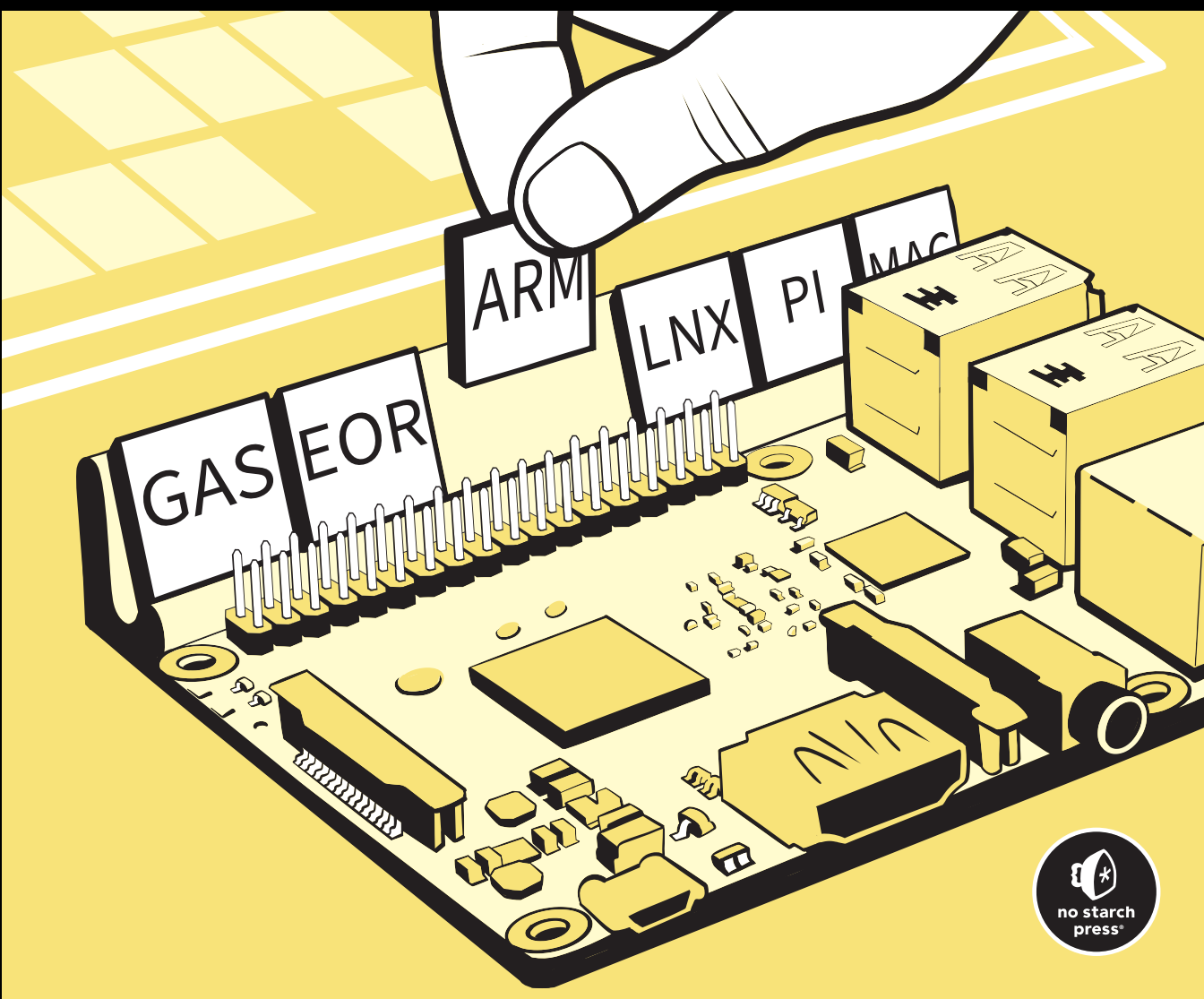
VOLUME 1

THE ART OF ARM ASSEMBLY

64-BIT ARM MACHINE ORGANIZATION AND PROGRAMMING

RANDALL HYDE



no starch press®

THE ART OF ARM ASSEMBLY, VOLUME 1

THE ART OF ARM ASSEMBLY

Volume 1

64-Bit ARM Machine Organization and Programming

by Randall Hyde

no starch press®

San Francisco

First printing

ISBN-13: 978-1-7185-0282-6 (print)
ISBN-13: 978-1-7185-0283-3 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Abigail Schott-Rosenfield
Cover Illustrator: James L. Barry
Interior Design: Octopod Studios
Technical Reviewer: Tony Tribelli
Copyeditor: Sharon Wilkey
Proofreader: Scout Festa

Library of Congress Control Number: 2024009591

```
Name: Hyde, Randall, author.
Title: The Art of ARM Assembly / by Randall Hyde.
Description: San Francisco : No Starch Press, 2025. | Includes index.
Identifiers: LCCN 2024009591 (print) | LCCN 2024009592 (ebook) |
  ISBN 9781718502826 (print) | ISBN 9781718502833 (ebook)
Subjects: LCSH: Assembly languages (Electronic computers) |
  ARM microprocessors--Programming.
Classification: LCC QA76.73.A8 H9698 2025  (print) | LCC QA76.73.A8 (ebook) |
  DDC 005.2--dc23/eng/20241007
LC record available at https://lccn.loc.gov/2024009591
LC ebook record available at https://lccn.loc.gov/2024009592
```

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

This book is dedicated to Apple, Inc., whose introduction of the M-series-based Mac computers made this book possible.

About the Author

Randall Hyde is the author of *The Art of Assembly Language*, *The Art of 64-Bit Assembly*, *The Book of I^2C*, and *Write Great Code*, Volumes 1, 2, and 3 (all from No Starch Press), as well as *Using 6502 Assembly Language and P-Source* (Datamost). He is also the co-author of *Microsoft Macro Assembler 6.0 Bible* (The Waite Group). For more than 46 years, Hyde has worked as an embedded software and hardware engineer developing instrumentation for nuclear reactors, traffic-control systems, and other consumer electronics devices. He has also taught computer science at California State Polytechnic University, Pomona, and at the University of California, Riverside. His website is *https://www.randallhyde.com*.

About the Technical Reviewer

Tony Tribelli has more than 35 years of experience in software development. This experience ranges from embedded device kernels to molecular modeling and visualization to video games. The last includes 10 years at Blizzard Entertainment. He is currently a software development consultant and is privately developing applications utilizing computer vision.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

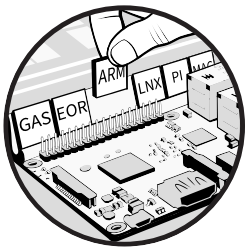
The Art of ARM Assembly has quite a bit of history behind it. This book, while a separate project from the original *Art of Assembly Language* and *The Art of 64-Bit Assembly*, does take some material from those earlier books. Therefore, I should begin my list of acknowledgments by mentioning the people I thanked in the first and second editions of *The Art of Assembly Language*: Bill Pollock, Alison Peterson, Ansel Staton, Riley Hoffman, Megan Dunchak, Linda Recktenwald, Susan Glinert Stevens, Nancy Bell, and Nathan Baker.

And in *The Art of 64-Bit Assembly*: Bill Pollock, Barbara Yien, Katrina Taylor, Miles Bond, Athabasca Witschi, Nathan Heidelberger, Natalie Gleason, Morgan Vega Gomez, Sharon Wilkey, Sadie Barry, and Jeff Lytle.

Finally, here are the people who have greatly contributed to *The Art of ARM Assembly*: Bill Pollock, Jill Franklin, Abigail Schott-Rosenfield, Sabrina Plomitallo-González, Sydney Cromwell, Sharon Wilkey, and Scout Festa.

Special thanks go to Tony Tribelli, this book's technical reviewer, who went above and beyond in tracking down issues with the book.

INTRODUCTION



Welcome to *The Art of ARM Assembly*. This book will teach you how to program 64-bit ARM CPUs, such as those found in modern Apple macOS machines, ARM-based Linux systems (including the Raspberry Pi with a 64-bit version of Raspberry Pi OS, previously known as Raspbian, which I'll just call Pi OS), and even mobile devices such as iPhones, iPads, and some Android devices. With the arrival of the ARM-based Apple macOS systems, the need to learn and understand 64-bit ARM assembly language increased dramatically, leading me to write this book. However, I've made the source code and other information in this book as portable as possible so that it applies to all 64-bit ARM machines.

This book is a sister volume to *The Art of 64-Bit Assembly*, which was, itself, a rewrite of *The Art of Assembly Language Programming* (*AoA*). *AoA* was a project I began way back in 1989 as a tool for teaching 80x86 (x86) assembly-language programming to students at California State Polytechnic University, Pomona, and the University of California, Riverside. For over 25 years, *AoA* served as a guide for learning x86 assembly language programming. During that time, other processors came and went, but x86 remained king of the hill in personal computers and high-end workstations, and x86 assembly language remained the de facto assembly language to learn. However, ARM-based PCs became mainstream with the introduction of the Apple M1-based systems (and later Apple machines), so the need to learn ARM assembly language programming is increasing.

This book was written using *The Art of 64-Bit Assembly* as a model for the material to cover. Anyone who has read my earlier books will find this book to be very familiar at a high level. Of course, the ARM instructions and assemblers—either the GNU assembler (Gas) or Apple's Clang assembler (largely compatible with Gas)—are quite different from the x86 instructions and the Microsoft Macro Assembler (MASM). The low-level presentation and programming techniques are therefore also somewhat different.

0.1 A Brief History of the ARM CPU

The ARM CPU has a long and storied history. It was first developed by Acorn Computers Ltd. in late 1983 as a replacement for the venerable 8-bit 6502 CPU used in its BBC Micro system. *ARM* originally stood for *Acorn RISC Machine*, though this was later changed to *Advanced RISC Machine* (*RISC* stands for *reduced instruction set computer*). That original design was largely a mind meld between the design described in the early University of California, Berkeley, RISC design and the 6502 CPU. For this reason, many would argue that the ARM initially wasn't a pure RISC design. We might think of the ARM as the spiritual successor to the 6502, inheriting many of the 6502's features.

In many respects, the ARM CPU is modeled on the 6502 CPU's notion of a reduced instruction set computer. In the original RISC design, each instruction was designed to do as little work as possible so it would require less hardware support and could run faster. Pure RISC architectures, for example, generally don't use condition code bits (as setting condition codes after the execution of an instruction would require the CPU do extra work), and use fixed-size machine instruction encodings (typically 32 bits). The 6502, on the other hand, attempted to reduce the total *number* of machine instructions as much as possible.

Additionally, the original ARM supported both 16-bit and 32-bit instruction encodings. While pure RISC CPUs try to maximize the number of general-purpose registers (generally 32), the original ARM design supported only 16. Furthermore, the ARM used one of the general-purpose registers as the program counter, which allows for all kinds of

programming tricks but creates problems for pure RISC designs (such as handling exceptions). Finally, the ARM partially supported a hardware stack, something you don't see on pure RISC machines. Nevertheless, "pure" or not, the ARM design outlasted all the other RISC CPUs of that era.

Over the years, the ARM CPU variants have largely been used in mobile and embedded applications, with the vast majority of ARM CPUs winding up in mobile phones and tablets. However, one notable use is in the Raspberry Pi computer system (with over 61 million units sold as of this writing). In addition to the Pi, millions of ARM-based Arduino-compatible and other single-board computers (such as the Teensy series) have been sold. At the time of writing, the Raspberry Pi Foundation released the Raspberry Pi Pico, an ARM-based microcontroller board for $4 (US), selling more than 4 million of these devices by January 2024.

0.2 Why Learn ARM Assembly?

RISC CPUs were designed to be programmed using high-level languages (especially C/C++). Very few programs of note have been written in RISC assembly language (though the original ARM Basic is a good counterexample). The main reason assembly language is taught in colleges and universities is to teach *machine organization* (an introduction to the machine's architecture). In addition, some applications (or, at the very least, portions of some applications) can benefit from an assembly language implementation. Speed and space are the two main reasons for using assembly language, though it is also true that certain algorithms are more easily written in assembly language (particularly bit-handling operations).

Finally, learning assembly language can help you write much better high-level language code. After all, a compiler for a language like C/C++ translates that high-level source code into assembly language. Understanding the underlying machine language will help you write better high-level language (HLL) code because you can avoid inefficient HLL constructs. This understanding can also be helpful when debugging or optimizing HLL code. Sometimes you must look at the code that the compiler generated to understand a bug or inefficiency.

So why a book on ARM assembly language in particular? Until the Apple Silicon M1 CPU came along, the only common personal computer using an ARM CPU was the Raspberry Pi. While the Pi was popular, it generally wasn't being used in schools to teach machine organization and assembly language programming. A few hobbyists were probably interested in picking up ARM assembly language on their own, but most Pi programmers were using Scratch or Python, with the hard-core types programming in C/C++. While mobile devices such as iPhones, iPads, and Android phones and tablets are also popular, developers rarely consider switching from Objective-C, Swift, or Java into assembly language for applications on those devices.

However, once Apple released M1-based Mac minis, MacBooks, and iMacs, the situation changed. Interest in low-level programming on ARMs

spiked, because now ARM assembly could be taught in colleges and universities on "normal" machines. Apple has sold more A-series (iPad and iPhone) and M-series (iPad and Mac) systems than Raspberry Pi since they were introduced. It is conceivable that Apple will have sold around a billion ARM-based personal computers and mobile devices by the time you're reading this.

Given these developments, a lot more people are going to be interested in assembly language programming on ARM CPUs. If you want to be able to write high-performance, efficient, and small code on this new crop of devices, learning ARM assembly language is the place to start.

0.3 Why Learn 64-Bit ARM?

Although the original ARM was a 32-bit CPU, Arm Holdings—the outfit that licenses the ARM design—introduced a 64-bit version in 2011. Apple introduced its 32-bit iPhone 5 a few years after that. Since then, most mobile and personal computer devices (including the Raspberry Pi 3, 4, and 400) have used 64-bit CPUs, while embedded devices have largely stuck with the 32-bit CPU variants. Code written for 32-bit CPUs is generally more memory efficient than that for 64-bit CPUs; unless an application requires more than 4GB, using a 32-bit instruction set is usually better.

Nevertheless, for high-performance computing, 64 bits is definitely the future. Why is this the case? Can't 64-bit ARM CPUs run the older 32-bit code? The answer is a qualified yes. For example, the Raspberry Pi provides a 32-bit OS that runs only 32-bit code, even when running on a 64-bit CPU such as on the Pi 3, 4, or 400. However, the 64-bit ARM CPUs (ARMv8 or AARCH64, informally abbreviated to ARM64) operate in one of two modes: 32-bit or 64-bit. When in 32-bit mode, they execute the 32-bit instruction set; when in 64-bit mode, they execute the 64-bit instruction set. Though these instruction sets have some similarities, they are not the same. Thus, when operating in one of these modes, you cannot execute the instructions from the other mode.

Given the incompatibility of the two instruction sets, this book focuses on 64-bit ARM assembly language. Since you can't program the Apple M1 (and later) in 32-bit ARM assembly language, teaching 32-bit alone would be a nonstarter. Why not teach both? While knowing 32-bit assembly language would help readers who want to write code for the 32-bit Pi OS and other embedded single-board microcontrollers, this book aims to teach fundamentals. Teaching two different instruction sets complicates the educational experience; better to do one thing well (64-bit assembly) rather than two things poorly. Teaching both 32-bit and 64-bit assembly is almost like trying to teach x86-64 and ARM in the same book; it's just too much to take in all at once. Moreover, the 32-bit operating modes will likely fade away entirely over time. As I write this, ARM has already introduced a variant that supports only 64-bit code; I expect all future desktop-class processors will head in this direction.

NOTE *Although concentrating on 64-bit ARM assembly language for desktop-class and mobile machines (such as iPhones) makes sense, some will want to learn 32-bit ARM assembly language to work with embedded devices. Arduino-based single-board computers (SBCs), Raspberry Pi Pico SBCs, and many other classes of ARM-based embedded systems use 32-bit ARM variants. Furthermore, if you're operating a Raspberry Pi using a 32-bit version of Pi OS, you'll need to use 32-bit ARM assembly language. For that reason,* The Art of ARM Assembly, Volume 2, *will cover 32-bit ARM assembly language on those systems.*

0.4 Expectations and Prerequisites

This book assumes that you are already comfortable programming in an HLL such as C/C++ (preferred), Python, Swift, Java, Pascal, Ruby, BASIC, or another object-oriented or imperative (procedural) programming language. Although many programmers have successfully learned assembly language as their very first programming language, I recommend that you learn to *program* first, then learn assembly language programming. This book makes use of several HLL examples (typically in C/C++ or Pascal). The examples are generally simple, so you should be able to understand them if you know a different HLL.

This book also assumes you're comfortable with the edit/compile/test/debug cycle during program development. You should be familiar with source code editors and using standard software development tools, as I won't explain how to edit source files.

A wide variety of 64-bit ARM systems are out there, and I aimed to make this book applicable to as many of them as possible. To that end, every example program in this book has been tested on each of the following systems:

- Apple M1-based Mac systems such as the Mac mini M1 and Mac mini M2. The book's example code was tested on the mini M1 but should work on any of the ARM-based MacBooks or iMacs, as well as future M*x* systems.
- Raspberry Pi 3, 4, 400, and 5 systems (and future 64-bit-capable Pi systems) running the 64-bit version of Pi OS.
- PINE64 system including the Pinebook, Pinebook Pro, and ROCKPro 64.
- Almost any 64-bit ARM-based Linux system.
- NVIDIA Jetson Nano systems.

In theory, it should be possible to apply the information in this book to ARM-based Windows machines (such as the Surface Laptop Copilot+). Unfortunately, Microsoft's software development tools, particularly its assembler, are based on the original ARM assembly syntax defined by Arm (the company), not Gas. While Microsoft's *armasm64* is a better tool in many respects (as it uses standard ARM assembly language syntax), everyone else uses Gas syntax. The machine instructions are more or less the same

between the two sets of assemblers, but the other statements (known as *assembler directives* or *pseudo-opcodes*) are completely different. Therefore, example programs written in Gas will not assemble under *armasm64*, and vice versa. Since trying to present both syntax forms in example programs would be just as confusing as trying to teach 32- and 64-bit programming simultaneously, I stick to Gas syntax in my examples.

0.5 Source Code

This book contains considerable ARM assembly language (and some C/C++) source code that typically comes in one of three forms: code snippets, single assembly language procedures or functions (modules), or full-blown programs.

Code snippets are fragments of a program; they are not stand-alone, and you cannot compile them by using an ARM assembler (or a C++ compiler, in the case of C/C++ source code). They exist to make a point or provide a small example of a particular programming technique. Here is a typical example:

```
     .data
i64  .quad   0
      .
      .
      .
     ldr     x1, i64
```

The vertical ellipses denote arbitrary code that could appear in their place.

Modules are small blocks of code that can be compiled but won't run on their own. Modules typically contain a function that will be called by another program. Here is a typical example:

```
someFunc:
        add x0, x1, x2
        sub x0, x0, x3
        ret
```

Full-blown programs are called *listings* in this book, and I refer to them by listing number or filename. A typical filename usually takes the form *Listing*C-N.*S*, where C is the chapter number and N is a listing number within that chapter. For example, the following *Listing1-1.S* is the first listing that appears in Chapter 1:

```
// Listing1-1.S
//
// Comments consist of all text from a //
// sequence to the end of the line.
// The .text directive tells MASM that the
// statements following this directive go in
```

```
// the section of memory reserved for machine
// instructions (code).

        .text

// Here is the main function.
// (This example assumes that the
// assembly language program is a
// stand-alone program with its own
// main function.)
//
// Under macOS, the main program
// must have the name _main
// beginning with an underscore.
// Linux systems generally don't
// require the underscore.
//
// The .global _main statement
// makes the _main procedure's name
// visible outside this source file
// (needed by the linker to produce
// an executable).

        .global _main

// The .align 2 statement tells the
// assembler to align the following code
// on a 4-byte boundary (required by the
// ARM CPU). The 2 operand specifies
// 2 raised to this power (2), which
// is 4.

        .align 2

// Here's the actual main program. It
// consists of a single ret (return)
// instruction that simply returns
// control to the operating system.

_main:
        ret
```

Although most listings take the form *Listing*C-N.*S*, some (especially those from external sources) simply consist of a descriptive filename, such as the *aoaa.inc* header file used by most of the sample programs in this book.

All listings are available in electronic form at *https://artofarm.randallhyde.com*, either individually or as a ZIP file containing all the listings found in this book. That page also contains support information for this book, including errata and PowerPoint slides for instructors.

Most of the programs in this book run from a command line. These examples typically use the bash shell interpreter. Therefore, every `build` command and sample output will typically have the text prefix `$` or `%` before any command you would type from the keyboard on the command line.

Under macOS, the default shell (command line) program is zsh. It prints a percent sign (%) rather than $ as the prompt character. If you are completely unfamiliar with the Linux or macOS command line, please see Appendix D for a quick introduction to the command line interpreter.

Unless otherwise noted, all source code appearing in this book is covered under the Creative Commons 4.0 license. You may freely use that code in your own projects as per the Creative Commons license. See *https://creativecommons.org/licenses/by/4.0/* for more details.

0.6 Typography and Pedantry

Computer books have a habit of abusing the English language. This book is no exception. Whenever source code snippets appear in the middle of an English sentence, a conflict often arises between the grammar rules of the programming language and English. This section describes my choices for differentiating syntactical rules in English versus programming languages, in addition to a few other conventions.

First, this book uses a monospaced font to denote any text that appears as part of a program source file. This includes variable and procedure functions, program output, and user input to a program. Therefore, when you see something like `get`, you know that the book is describing an identifier in a program, not commanding you to get something.

A few logic operations have names that also have common English meanings: AND, OR, and NOT. When using these terms as logic functions, this book uses all caps to help differentiate otherwise-confusing English statements. When using these terms as English, this book uses the standard typeset font. The fourth logic operator, exclusive or (XOR), doesn't normally appear in English statements, but this book still capitalizes it.

In general, I always try to define any acronym or abbreviation the first time I use it. If I haven't used the term in a while, I will often redefine it on that usage. The glossary in Appendix B also includes most of the acronyms appearing in this book.

0.7 Organization

This book is organized into 4 parts comprising 16 chapters and 6 appendixes.

Part I, Machine Organization, covers data types and machine architecture for the ARM processor:

Chapter 1: Hello, World of Assembly Language Teaches you a small handful of instructions so you can experiment with the software development tools and write simple little programs.

Chapter 2: Data Representation and Operations Discusses the internal representation of simple data types such as integers, characters, and Boolean values. It also discusses the various arithmetic and logical operations possible on these data types. This chapter also introduces some basic ARM assembly language operand formats.

Chapter 3: Memory Access and Organization Discusses how the ARM organizes main memory. It explains the layout of memory and how to declare and access memory variables. It also introduces the ARM's methods for accessing memory and the stack (a place to store temporary values).

Chapter 4: Constants, Variables, and Data Types Describes how to declare named constants in assembly language, how to declare and use pointers, and the use of composite data structures such as strings, arrays, structs (records), and unions.

Part II, Basic Assembly Language, provides the basic tools and instructions you need to write assembly language programs.

Chapter 5: Procedures Covers the instructions and syntax you need to write your own assembly language functions (procedures). This chapter describes how to pass arguments (parameters) to functions and return function results. It also describes how to declare (and use) local or automatic variables that you allocate on the stack.

Chapter 6: Arithmetic Explains the basic integer arithmetic and logical operations in ARM assembly language. It also describes how to convert arithmetic expressions from an HLL into ARM assembly language. Finally, this chapter covers floating-point arithmetic using the hardware-based floating-point instructions.

Chapter 7: Low-Level Control Structures Describes how to implement HLL-like control structures such as `if`, `elseif`, `else`, `while`, `do...while` (`repeat...until`), `for`, and `switch` in ARM assembly language. This chapter also touches on optimizing loops and other code in assembly language.

Part III, Advanced Assembly Language, covers more advanced assembly language operations.

Chapter 8: Advanced Arithmetic Explores extended-precision arithmetic, mixed-mode arithmetic, and other advanced arithmetic operations.

Chapter 9: Numeric Conversion Provides a very useful set of library functions you can use to convert numeric values to string format and convert string values to numeric format.

Chapter 10: Table Lookups Describes how to use memory-based lookup tables (arrays) to accelerate certain computations.

Chapter 11: Neon and SIMD Programming Discusses the ARM Advanced SIMD instruction set that allows you to speed up certain applications by operating on multiple pieces of data at once.

Chapter 12: Bit Manipulation Describes various operations and functions that allow you to manipulate data at the bit level in ARM assembly language.

Chapter 13: Macros and the Gas Compile-Time Language Covers the Gas macro facilities. Macros are powerful constructs enabling you to

design your own assembly language statements that expand to a large number of individual ARM assembly language instructions.

Chapter 14: String Operations Explains the use and creation of various character string library functions in ARM assembly language.

Chapter 15: Managing Complex Projects Describes how to create libraries of assembly language code, and build those libraries by using makefiles (along with a discussion of the `make` language).

Chapter 16: Stand-Alone Assembly Language Programs Shows how to write assembly language applications that don't use the C/C++ standard library for I/O and other operations. This chapter includes system call examples for both Linux (Pi OS) and macOS.

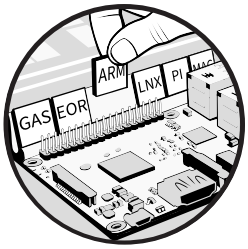
Part IV, Reference Materials, provides reference information, including a table listing the full ASCII character set, a glossary, instructions for installing and using Gas on your system, an introduction to the bash shell interpreter, useful C/C++ functions you can call from your assembly language programs, and answers to the questions at the end of each chapter.

PART I

MACHINE ORGANIZATION

1

HELLO, WORLD OF ASSEMBLY LANGUAGE



This "quick-start" chapter gets you writing basic assembly language programs as rapidly as possible, giving you the skills you need to learn new assembly language features in the following chapters. You'll learn the foundations of 64-bit ARM architecture and the basic syntax of the GNU assembler (Gas) program, a compiler for assembly language.

You'll also learn to set aside memory for variables, control the CPU by using machine instructions, and link a Gas program with C/C++ code so that you can call routines in the C standard library (C stdlib). Gas running under Linux and macOS is by far the most common assembler for writing real-world ARM assembly language programs. Vendors (especially Apple) have produced variants of Gas with slightly different syntax; for instance, under macOS, Gas is known as the *Clang* or *Mach-O* assembler. To make the source code in this book portable between macOS and Linux, this chapter

also introduces a header file, *aoaa.inc*, that eliminates the differences between Gas and the Clang assembler.

1.1 What You'll Need

To learn assembly language programming with Gas, you'll need a version of the assembler for your platform, plus a text editor for creating and modifying Gas source files, a linker, various library files, and a C++ compiler. You'll learn to set up the Gas assembler and text editor in this section, and the other tools later in this chapter.

1.1.1 Setting Up Gas

The GNU Compiler Collection (GCC) emits Gas source code as its output (which Gas then converts to object code). Therefore, if you have the compiler suite running on your system, you also have Gas. Apple macOS uses a compiler based on the LLVM compiler suite rather than GCC, so if you have a macOS, you'll need to install its Xcode integrated development environment (IDE) to gain access to the assembler (see Appendix C). Otherwise, if you don't have the GCC compiler, install it with the instructions in your operating system (OS) documentation.

NOTE *The GNU assembler and the Clang assembler's executable name is actually* as *(assembler). The examples in this book rarely invoke the assembler directly, so you won't use the* as *program often. Therefore, this book refers to the assembler by using the name* Gas *rather than* as *(or Clang assembler).*

1.1.2 Setting Up a Text Editor

To write ARM assembly language programs, you will need some sort of programmer's text editor to create assembly language source files. The choice of editor is dictated by personal tastes and editor availability for your OS or development suite.

The standard suffix for assembly language source files is *.s*, since GCC emits this suffix when it converts a C/C++ file into assembly language during compilation. For handwritten assembly language source files, the *.S* suffix is a better choice, since it tells the assembler to route the source file through the C preprocessor (CPP) before assembly. Since this allows the use of CPP macros (`#define` statements), conditional compilation, and other facilities, all example files in this book use *.S*.

GCC always produces assembly language output files, which are then processed by Gas. GCC automatically invokes the assembler and then deletes the assembly source file after the assembly is complete.

1.1.3 Understanding C/C++ Examples

Today's software engineers drop into assembly language only when their C/C++, C#, Java, Swift, or Python code is running too slowly and they need

to improve the performance of certain modules or functions. The examples in this book use C/C++ because you'll typically interface assembly language with C/C++ or other high-level language (HLL) code in the real world.

The C/C++ standard library is another good reason to use this language. To make the C stdlib immediately accessible to Gas programs, I present examples with a short C++ main function that calls a single external function written in assembly language using Gas. Compiling the C++ main program along with the Gas source file produces a single executable file that you can run and test.

This book spoon-feeds you the C++ you'll need to run the example HLL programs, so you'll be able to follow even if you're not fluent in the language. However, you'll have an easier time if you have a little prior familiarity with C/C++. At minimum, this book assumes that you have some experience in a language such as Pascal (or Delphi), Java, Swift, Rust, BASIC, Python, or any other imperative or object-oriented programming language.

1.2 The Anatomy of an Assembly Language Program

A typical (stand-alone) Gas program takes the form shown in Listing 1-1.

```
// Listing1-1.S
//
// Comments consist of all text from a //
// sequence to the end of the line.
// The .text directive tells Gas that the
// statements following this directive go in the
// section of memory reserved for machine
// instructions (code).

     ❶ .text

// Here is the main function. (This example assumes
// that the assembly language program is a
// stand-alone program with its own main function.)
//
// Under macOS, the main program must have the name
// _main beginning with an underscore. Linux
// systems generally don't require the underscore.
//
// The .global _main statement makes the _main
// procedure's name visible outside this source file
// (needed by the linker to produce an executable).

        .global _main, main

// The .align 2 statement tells the assembler to
// align the following code on a 4-byte boundary
// (required by the ARM CPU). The 2 operand
// specifies 2 raised to this power (2), which is 4.
```

```
      ❷ .align 2

// Here's the actual main program. It consists of a
// single ret (return) instruction that simply
// returns control to the operating system.

_main:
main:
        ret
```

Assembly language programs are broken into *sections*. Some sections contain data, some contain constants, some contain machine instruction (executable statements), and so on. Listing 1-1 contains a single code section, called *text* in macOS and Linux. The `.text` statement ❶ tells the assembler that the following statements are associated with the code section.

In assembly language source files, symbols are usually local or private to a source file. When creating an executable source file, you must pass one or more symbols to the system linker—at least the name of the main program. You can accomplish this by using the `.global` statement, specifying the global name as an operand: `_main` in the macOS case, `main` in the Linux case. Leaving out this statement gives you an error when you try to compile the source file.

The ARM instruction set requires all machine instructions to begin on a 32-bit (4-byte) boundary in memory. Therefore, before the first machine instruction in a `.text` section, tell the assembler to align the addresses on a 4-byte boundary. The `.align` statement ❷ raises 2 to the power specified by its operand and aligns the next instruction on that boundary. Since 2^2 is equal to 4, this statement aligns the next instruction on a 4-byte boundary.

A procedure, or function, in ARM assembly simply consists of the name of that function (`_main` or `main` in this case) followed by a colon. The machine instructions follow. The main program in this example consists of a single machine instruction: `ret` (return). This instruction immediately returns control to whatever called the main program—that is, the OS.

Identifiers in Gas are similar to identifiers in most HLLs. Gas identifiers may begin with a dollar sign ($), an underscore (_), or an alphabetic character and may be followed by zero or more alphanumeric, dollar sign, or underscore characters. Symbols are case sensitive.

LINUX VS. MACOS: GLOBAL NAMES

The C/C++ compiler treats global (`extern`) names differently in macOS and Linux programs. The Clang compiler (macOS) automatically prepends an underscore character (_) to the beginning of each external/global symbol, as in `_main` in Listing 1-1; the GCC compiler does not.

> I've written the source code appearing in this book to make it easy to port between the two OSes, by using equates for all the global symbols so that they have to be changed in only one spot. We'll discuss using equates to resolve external symbols in section 4.1, "Gas Constant Declarations," on page 170; also see section 1.12, "For More Information," on page 43 for details specific to macOS and Linux assembly language programming.

While the program in Listing 1-1 doesn't really *do* anything, you can use it to learn how to use the assembler, linker, and other tools necessary for writing ARM assembly language programs, as we'll do in the next section.

1.3 Running Your First Assembly Language Program

Once you have an assembly source file, you can compile and run that program. In theory, you could run the assembler (`as`) and then the linker (`ld`, supplying appropriate library files needed by the OS). Here's how that would look for macOS (where the `$` appearing at the beginning of each line is the OS's shell prompt):

```
$ as -arch arm64 Listing1-1.S -o Listing1-1.o
$ ld -o Listing1-1 Listing1-1.o -lSystem \
     -syslibroot `xcrun -sdk macosx --show-sdk-path` \
     -e _main -arch arm64
$ ./Listing1-1
```

However, the command lines differ depending on your OS, and producing an executable in this way takes a lot of typing. An easier way to compile the program and produce an executable is to use the GCC compiler (`g++`) by running this command:

```
$ g++ -o Listing1-1 Listing1-1.S
```

This command line even works on macOS, which uses the Clang compiler rather than GCC; macOS has an alias for Clang named `g++`. On macOS, you could also use the `clang -o Listing1-1 Listing1-1.S` command line. This book, however, will stick to the `g++` command line, as that works on macOS and Linux.

The `g++` command is smart enough to note that this is an assembly language source file and run Gas on it to produce an object file. GCC will then run the linker (`ld`) and supply all the default libraries the OS requires.

You can run the resulting executable file from the command line as follows:

```
$ ./Listing1-1
```

This program immediately returns without any output, since that's all Listing 1-1 does; it's simply intended to demonstrate how to compile and run ARM assembly language programs.

In addition to reducing the amount of typing required, using g++ to assemble your assembly language source files provides another advantage: it's the easiest way to run the CPP, which many of the example files in this book require. You can invoke the CPP (by itself) on an assembly source file by using a command like the following, to see the modifications the CPP makes to your assembly source files:

```
$ g++ -E Listing1-1.S
```

You can even pipe the output from the CPP to Gas, using the following command:

```
$ g++ -E Listing1-1.S | as -o Listing1-1.o
```

However, at that point, you may as well have typed

```
$ g++ -o Listing1-1.o Listing1-1.S
```

as it's shorter and easier to input.

1.4 Running Your First Gas/C++ Hybrid Program

This book commonly combines an assembly language module containing one or more functions written in assembly language with a C/C++ main program that calls those functions. Because the compilation and execution process is slightly different from a stand-alone Gas program, this section demonstrates how to create, compile, and run a hybrid assembly/C++ program. Listing 1-2 provides the main C++ program that calls the assembly language module.

```
// Listing1-2.S
//
// A simple C++ program that calls
// an assembly language function
//
// Need to include stdio.h so this
// program can call printf().

#include <stdio.h>

// extern "C" namespace prevents
// "name mangling" by the C++
// compiler.

extern "C"
{
    // Here's the external function,
```

```
    // written in assembly language,
    // that this program will call:

     void asmMain( void );
};

int main(void)
{
    printf( "Calling asmMain:\n" );
    asmMain();
    printf( "Returned from asmMain\n" );
}
```

Listing 1-3, a slight modification of the stand-alone Gas program, contains the `asmMain()` function that the C++ program calls. The main difference between Listing 1-3 and Listing 1-1 is that the function's name changes from `_main` to `_asmMain`. The C++ compiler and linker would get confused if we continued to use the name `_main`, as that's also the name of the C++ main function.

```
// Listing1-3.S
//
// A simple Gas module that contains
// an empty function to be called by
// the C++ code in Listing 1-2

        .text

// Here is the asmMain function:

        .global _asmMain, asmMain
        .align  2    // Guarantee 4-byte alignment.
_asmMain:
asmMain:

// Empty function just returns to C++ code.

        ret          // Returns to caller
```

Finally, to compile and run these source files, run the following commands:

```
$ g++ -o Listing1-2 Listing1-2.cpp Listing1-3.S
$ ./Listing1-2
Calling asmMain:
Returned from asmMain
$
```

Granted, this assembly language example doesn't accomplish much other than demonstrate how to compile and run some assembly code. To write real assembly code, you're going to need a lot of support code. The

next section describes the *aoaa.inc* header file that provides some of this support.

1.5 The aoaa.inc Include File

The example code in this book was written to be as portable between macOS and Linux assemblers as possible, a difficult task requiring considerable advanced behind-the-scenes trickery. Many of those tricks are a bit too advanced to easily explain to beginning ARM programmers, so I've incorporated all this magic code in a special header file, *aoaa.inc*, that I use in most of the example programs from this point forward.

This human-readable include file is little more than a typical advanced C/C++ header file; it just contains a bunch of macros (for example, C/C++ `#define` statements) that help smooth out some of the differences between the macOS and Linux versions of the assembler. By the time you get to the end of this book (especially by the time you read Chapter 13), most of the material in the header file will make perfect sense. For now, I won't distract you with advanced macros and conditional assembly information.

You can find *aoaa.inc* along with all the other example code at *https://artofarm.randallhyde.com*. If you're curious about this file's content and don't want to wait for Chapter 13, load it into a text editor and take a look.

To include this file in an assembly, use the following CPP statement in your assembly language source files:

```
#include "aoaa.inc"
```

Just as in C/C++, this statement will automatically insert the content of this file into the current source file during assembly (at the point of the `#include` statement).

Gas has its own include statement, used as follows:

```
.include "include_file_name"
```

However, don't use this statement to include *aoaa.inc* in your source files. The Gas `.include` directive executes after the CPP runs, but *aoaa.inc* contains CPP macros, conditional compilation statements, and other code that must be processed by the CPP. If you use the `.include` directive rather than `#include`, the CPP will never see the contents of the *aoaa.inc* file, and Gas will generate errors when it processes the file.

The *aoaa.inc* file must be present in the same directory as your assembly source file during the assembly process (or you must supply an appropriate path to the file in the `#include "aoaa.inc"` statement). If the header file isn't in the current directory, Gas will complain that it can't find the file and terminate the assembly. Also remember to use the *.S* suffix with your assembly source files when using `#include "aoaa.inc"`, or GCC won't run the CPP on those files.

1.6 The ARM64 CPU Architecture

Thus far, you've seen a pair of Gas programs that compile and run. However, the statements appearing in those programs do nothing more at this point than return control to the OS. Before you learn some real assembly language, you'll need to understand the basic structure of the ARM CPU family so you can follow the machine instructions.

The ARM CPU family is generally classified as a Von Neumann architecture machine. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/0) devices*. These three components are interconnected via the *system bus* (consisting of the address, data, and control buses). Figure 1-1 shows this relationship.

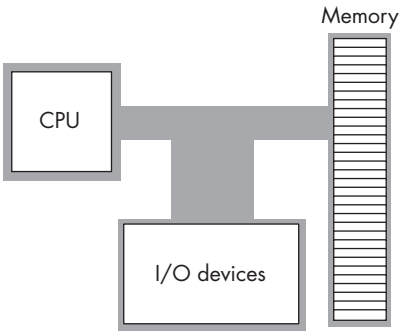


Figure 1-1: A Von Neumann computer system block diagram

The CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select one of the memory or I/O device port locations, each of which has a unique binary numeric *address*. Then the CPU, memory, and I/O devices pass data among themselves by placing the data on the data bus. The control bus contains signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).

1.6.1 ARM CPU Registers

There are two categories of ARM CPU registers: *general-purpose registers* and *special-purpose kernel-mode registers*. The special-purpose registers are intended for writing OSes, debuggers, and other system-level tools. Such software construction is well beyond the scope of this text.

The ARM64 supports 32 general-purpose 64-bit registers (named X0 through X31) and 32 general-purpose 32-bit registers (named W0 through W31). This doesn't imply there are 64 registers total; instead, the 32-bit registers overlay the low-order (LO) 32 bits of each of the 64-bit registers. (Chapter 2 discusses LO components in more depth.) Modifying one of the 32-bit registers also modifies the corresponding 64-bit register, and vice versa, as outlined in Figure 1-2.

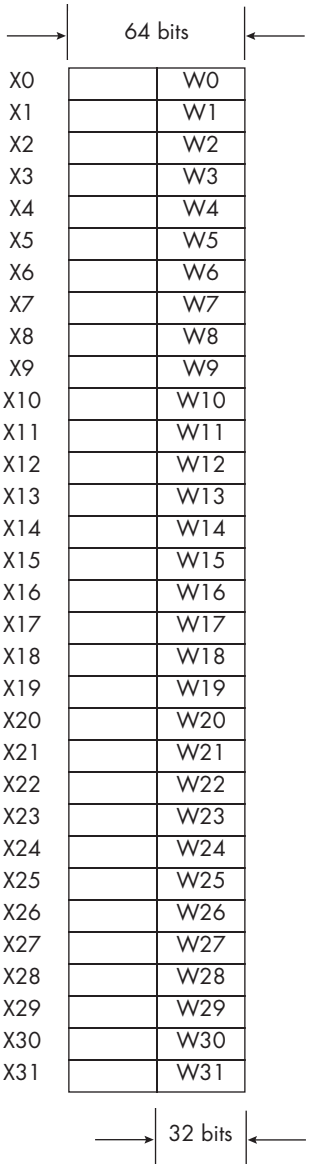


Figure 1-2: The 32- and 64-bit registers on the ARM

Those new to assembly language are often surprised that all calculations on the ARM64 involve a register. For example, to add two variables together, storing the sum into a third variable, you must load one of the variables into a register, add the second operand to the value in the register, and then store the register away in the destination variable. Registers are a middleman in nearly every calculation, so they're important in ARM64 assembly language programs.

Although these registers are known as *general-purpose* registers, a few of them have special purposes:

- X31, usually referred to as *SP* in code, is called the *stack pointer*, since it's used to maintain a *hardware stack* on the ARM (another non-RISC, or reduced instruction set computer, feature), always as a 64-bit register. Because it's used as the stack pointer, SP cannot be used for other purposes in most code. This register is accessible only by a few instructions.
- The XZR/WZR register (also treated as X31/W31 by the hardware) is called the *zero* register. It always returns 0 when read and is a convenient way to obtain the constant 0 in a program.
- Register X30 is the *link register*, referred to by the name *LR* rather than X30. The ARM CPU uses this register to hold *return addresses* when the code makes a function call. (Chapter 5 discusses the LR in greater detail.) This register is also always accessed in 64-bit mode. While you could theoretically use X30/W30 as a general-purpose register, you should avoid doing so, because function calls will wipe out the value in this register.
- Although this special purpose isn't enforced by the hardware, most software uses X29 as a 64-bit *frame pointer (FP)*. Software generally uses this register to provide access to function parameters and local variables. Technically, you could use any of the general-purpose registers for this purpose, but using X29/FP is conventional.
- Apple reserves X18 for its own internal purposes. Programs written for macOS, iOS, iPadOS, and so on must not use this register. Since 29 other registers are available, the examples in this book don't use X18, even for Linux examples.

In addition to the 32 general-purpose registers, the ARM64 CPUs have two additional special-purpose registers accessible to user programs: the 32-bit *processor state (PSTATE)* register and the 64-bit *program counter (PC)* register. The PC register always contains the address of the machine instruction being executed. Because instructions are always 32 bits long, the CPU will increment this register by 4 whenever it finishes the execution of one instruction and moves on to the next (more on this activity in Chapter 2).

NOTE *32-bit ARM CPUs refer to the PSTATE register as the* CPSR *or* PSR. *You may see references to those names in various documents.*

The PSTATE register is 32 bits wide (of which only 16 bits are used at the time of this writing) and is really just a collection of individual Boolean flags. Its layout appears in Figure 1-3.

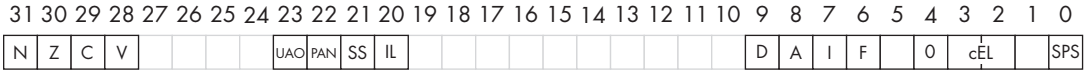


Figure 1-3: The PSTATE register layout

Most user applications use only the N, Z, C, and V bits in the PSTATE register. These bits, also known as the *condition codes*, have the following meanings:

N Negative (sign) flag, set when an instruction produces a negative result

Z Zero flag, set when an instruction produces a zero result

C Carry flag, set when an unsigned arithmetic overflow occurs

V Overflow flag, set when a signed arithmetic overflow occurs

Most of the remaining flags are inaccessible or of little use in user programs. UAO and PAN control CPU access features, allowing user programs to access kernel memory. SS is the single-step control bit for debugging. IL is the illegal instruction flag, set when the CPU executes an illegal instruction. D, A, I, and F are interrupt flags. cEL selects an exception level, usually 00 for user mode. SPS selects a stack pointer to use (kernel versus user mode).

In addition to the 32 general-purpose registers, the ARM64 provides 32 floating-point and vector registers to handle nonintegral arithmetic. Chapters 6 and 11 discuss these registers in greater detail when covering floating-point arithmetic and single-instruction/multiple data (SIMD) operations.

1.6.2 The Memory Subsystem

A typical ARM64 processor running a modern 64-bit OS can access a maximum of 2^{48} memory locations, or just over 256TB—probably far more than any of your programs will ever need. Since the ARM64 supports byte-addressable memory, the basic memory unit is a byte, which is sufficient to hold a single character or a very small integer value (discussed further in Chapter 2).

Because 2^{48} is a frightfully large number, the following discussion uses the 4GB address space of 32-bit ARM processors. Scaled up, the same discussion applies to 64-bit ARM processors.

NOTE *While the ARM64 supports 64 address bits in software, the hardware supports only 48 to 52 address bits for virtual memory operations. Most OSes limit this to 48 bits.*

Think of memory as a linear array of bytes. The address of the first byte is 0, and the address of the last byte is $2^{32} - 1$. For an ARM processor, the following pseudo-Pascal array declaration is a good approximation of memory:

```
Memory: array [0..4294967295] of byte;
```

C/C++ and Java users might prefer the following syntax:

```
byte Memory[4294967296];
```

To execute the equivalent of the Pascal statement `Memory [125] := 0;`, the CPU places the value 0 on the data bus, places the address 125 on the address bus, and asserts the write line (which generally involves setting that line to 0), as shown in Figure 1-4.

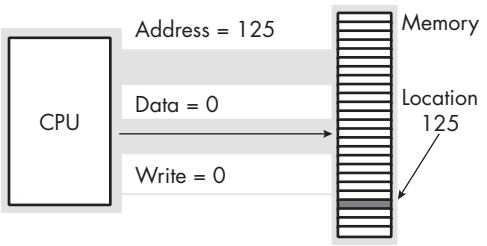


Figure 1-4: The memory write operation

To execute the equivalent of `CPU := Memory [125];`, the CPU places the address 125 on the address bus, asserts the read line (because the CPU is reading data from memory), and reads the resulting data from the data bus (see Figure 1-5).

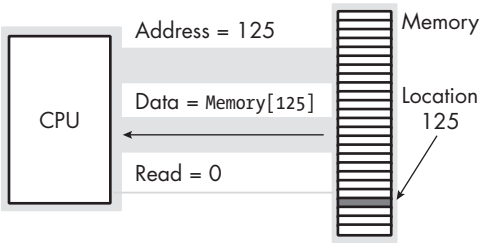


Figure 1-5: The memory read operation

This discussion applies only when accessing a single byte in memory. To store values larger than a single byte, like *half words* (2 bytes) and *words* (4 bytes), the ARM uses a sequence of consecutive memory locations, as shown in Figure 1-6. The memory address is the address of each object's first byte (that is, the lowest address).

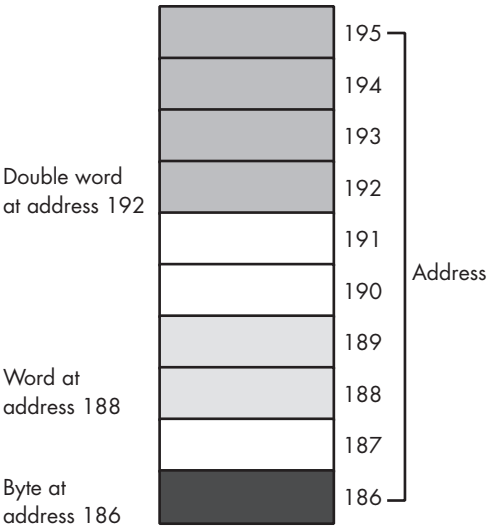


Figure 1-6: Byte, half-word, and word storage in memory

The ARM64 generally supports *unaligned memory access*, meaning the CPU can read or write an object of any size—byte, half word, word, or double word (dword)—at any address in memory. Certain instructions, however, require that memory access be aligned on the natural size of the transfer. Generally, this means that 16-, 32-, and 64-bit memory accesses must take place on addresses that are a multiple of 2, 4, or 8; otherwise, the CPU may raise an exception. Regardless of exceptions, the CPU can usually access memory locations aligned on a natural boundary faster.

Modern ARM processors don't connect directly to memory. Instead, a special memory buffer on the CPU known as the *cache* (pronounced "cash") acts as a high-speed intermediary between the CPU and main memory. You'll learn to set the alignment of memory objects and the effects of the cache on data alignment in Chapter 3.

1.7 Declaring Memory Variables in Gas

Referencing memory by using numeric addresses in assembly language is possible, but painful and error-prone. Rather than having your program state, "Give me the 32-bit value held in memory location 192 and the 16-bit value held in memory location 188," it's much nicer to state, "Give me the contents of `elementCount` and `portNumber`." Using variable names, rather than memory addresses, makes your program much easier to write, read, and maintain.

To create (writable) data variables, you have to put them in a data section of the Gas source file, defined using the `.data` directive. The `.data` directive tells Gas that all following statements (up to the next `.text` or other section-defining directive) will define data declarations to be grouped into a read/write section of memory.

Within a `.data` section, Gas allows you to declare variable objects by using a set of data declaration directives. The basic form of a data declaration directive is

```
label:  directive value(s)
```

where *label* is a legal Gas identifier and *directive* is one of the directives in the following list:

`.byte` Byte (8-bit) values. One or more comma-separated 8-bit expressions appear in the operand field (*values*).

`.hword, .short, .2byte` Half-word (16-bit) values. One or more comma-separated 16-bit expressions appear in the operand field.

`.word, .4byte` Word (32-bit) values. One or more comma-separated 32-bit expressions appear in the operand field.

`.quad, .8byte` Dword (64-bit) values. One or more comma-separated 64-bit expressions appear in the operand field. `.quad` is an unfortunate misnomer for ARM64, since a 64-bit value is actually a double word, not a quad word (on the ARM, a quad word is 128 bits). The term predates the ARM assembler, coming from "quad word" in the x86 and 68000 assembly language days. To avoid confusion, this book uses the `.dword` directive in place of `.quad`.

`.dword` The `.dword` macro appearing in the *aoaa.inc* include file is a synonym for the `.quad` directive that emits 8 bytes (64 bits) for each operand. Using `.dword` is preferable to `.quad`. You must include the *aoaa.inc* file in order to use this directive.

`.octa` Octaword (oword, 128-bit/16-byte) values. One or more comma-separated 128-bit expressions appear in the operand field. `.octa` is an unfortunate misnomer for ARM64, since a 128-bit value is actually a quad word, not an "octa" word (on the ARM, an octaword is 256 bits). To avoid confusion, this book avoids the `.octa` directive and uses `.qword` instead.

`.qword` This is a macro appearing the *aoaa.inc* include file. It is a synonym for the `.octa` directive and emits 16 bytes for each operand. You must include the *aoaa.inc* file in order to use this directive.

`.ascii` String values. A single string constant (surrounded by quotation marks) appears in the operand field. Note that Gas does not terminate this string with a 0 byte.

`.asciz` Zero-terminated string values. A single string constant (surrounded by quotation marks) appears in the operand field. Gas will emit a 0 after the last character in the string operand.

`.float` Single-precision floating-point values. One or more comma-separated 32-bit single-precision floating-point expressions appear in the operand field.

`.double` Double-precision floating-point values. One or more comma-separated 64-bit double-precision floating-point expressions appear in the operand field.

Gas provides additional synonyms for some of the directives in this list; see the link to the Gas documentation in section 1.12, "For More Information," on page 43.

Here are some examples of valid Gas data declarations:

```
byteVar:    .byte   0
halfVar:    .hword  1,2  // Actually reserves 2 half-words
wordVar:    .word   -1
dwordVar:   .dword  123456789012345
str1:       .ascii  "Hello, world!\n"  // Uses C-style escape for newline
str2:       .asciz  "How are you?\n"   // Sequences are legal.
pi:         .float  3.14159
doubleVar:  .double 1.23456e-2
```

Whenever you declare a variable in this manner, Gas will associate the current location in the output object-code file with the label at the beginning of the line. It will then emit the appropriate-sized data value into memory at that location, adjusting the assembler's *location counter* (which tracks the current location) by the size of each operand it emits.

The `label` field in these data declaration directives is optional. If you do not include the label, Gas simply emits the data in the operand field, starting at the current location counter and incrementing the location counter afterward. This is useful, for example, when you want to insert a control character or special Unicode character into a string:

```
longStr:   .ascii "A bell character follows this string"
           .byte  7, 0   // Bell (7) and zero termination
```

Gas allows C-style escape sequences within quoted strings. Although Gas doesn't support the full set of escape characters, it does support the following:

- **\b** Backspace character (0x08)
- **\n** Newline character/line feed (0x0A)
- **\r** Carriage return (0x0D)
- **\t** Tab (0x09)
- **\f** Form feed character (0x0C)
- **** Backslash character
- ***nnn*** Where *nnn* is a three-digit octal value; emit the value to the code stream
- **\x*hh*** Where *hh* is a two-digit hexadecimal value; emit the value to the code stream

Gas does not support \a, \e, \f, \v, \', \", \?, \u*hhhh*, or \U*hhhh* escape sequences.

1.7.1 Associating Memory Addresses with Variables

With an assembler like Gas, you don't have to worry about numeric memory addresses. Once you declare a variable in Gas, the assembler associates that variable with a unique set of memory addresses. For example, say you have the following declaration section:

```
      .data
i8:   .byte   0
i16:  .hword  0
i32:  .word   0
i64:  .dword  0
```

Gas will find an unused 8-bit byte in memory and associate it with the `i8` variable; it will likewise associate a pair of consecutive unused bytes with `i16`, 4 consecutive unused bytes with `i32`, and 8 consecutive unused bytes with `i64`. You'll always refer to these variables by their names and generally don't have to concern yourself with their numeric addresses. Still, be aware that Gas is doing this for you.

When Gas is processing declarations in a `.data` section, it assigns consecutive memory locations to each variable. Assuming `i8` (in the previous declarations) as a memory address of 101, Gas will assign the addresses appearing in Table 1-1 to `i8`, `i16`, `i32`, and `i64`.

Table 1-1: Variable Address Assignments

Variable	Memory address
`i8`	101
`i16`	102 (address of `i8` plus 1)
`i32`	104 (address of `i16` plus 2)
`i64`	108 (address of `i32` plus 4)

Technically, Gas assigns offsets into the *`.data`* section to variables. Linux/macOS converts these offsets to physical memory addresses when it loads the program into memory at runtime.

Whenever you have multiple operands in a data declaration statement, Gas will emit the values to sequential memory locations in the order in which they appear in the operand field. The label associated with the data declaration (if one is present) is associated with the address of the first (leftmost) operand's value. See Chapter 4 for more details.

1.7.2 Aligning Variables

As noted already, your programs may run faster if your variables are aligned on a *natural boundary* (alignment to the size of the object). Alignment is accomplished with the `.align` directive, which you saw in Listing 1-1.

Byte variables don't require any alignment. Use the `.align 1` directive to put half words at an even address (2-byte boundary); remember, Gas will

align the next statement on a boundary that is equal to 2^n, where n is the `.align` statement's operand. For words, use the `.align 2` directive. For double words (`.dword`), use the `.align 3` directive.

For example, let's return to the declaration given earlier:

```
     .data
i8:  .byte   0
i16: .hword  0
i32: .word   0
i64: .dword  0
```

Sticking `.align` directives in front of every declaration (except `i8`) will start to clutter up your code and make it harder to read:

```
     .data
i8:  .byte   0  // No alignment necessary for bytes
     .align  1
i16: .hword  0
     .align  2
i32: .word   0
     .align  3
i64: .dword  0
```

If your variables don't have to be declared in a particular order, you can clean this up by declaring the largest variables first and the remaining variables sorted by decreasing size. If you do this, you have to align only the first variable in your declaration list:

```
           .data
     .align  3
i64: .dword  0
i32: .word   0
i16: .hword  0
i8:  .byte   0  // No alignment necessary for bytes
```

Because the `i64` declaration appears immediately after the `.align 3` statement in this code, the `i64` address will be aligned on an 8-byte boundary. As `i32` immediately follows `i64` in memory, it will also be aligned on an 8-byte boundary (which, of course, is also a 4-byte boundary). This is because `i64` is aligned on an 8-byte boundary and consumes 8 bytes; therefore, the address following `i64` (the address of `i32`) will also be 8-byte aligned.

Meanwhile, because `i16` immediately follows `i32` in memory, it will be aligned on a 4-byte boundary (which is also an even address). The alignment of `i8` doesn't matter, but it happens to be at an even address, as it follows `i16`, which was aligned on a 4-byte boundary and consumes 2 bytes.

NOTE *Gas also provides a* `.balign` *directive whose operand must be a power of 2 (1, 2, 4, 8, 16, . . .) to specify the alignment value directly, rather than as a power of 2. While this book uses* `.align` *because it's the original directive, feel free to use* `.balign` *if you prefer.*

Strings are sequences of bytes, so their alignment usually doesn't matter. However, it is possible to write very high-performance string functions in assembly language that process strings eight or more characters at a time. If you have access to such library code, it might run faster if your strings are aligned on an 8-byte boundary.

Of course, floats and doubles should be aligned on 4-byte and 8-byte boundaries for the highest performance. In fact, as you'll see in Chapter 11, 16-byte alignment is also sometimes better.

1.7.3 Declaring Named Constants in Gas

Gas allows you to declare manifest constants by using the `.equ` directive. A *manifest constant* is a symbolic name (identifier) that Gas associates with a value. Everywhere the symbol appears in the program, Gas will directly substitute its value.

A manifest constant declaration takes the following form:

```
.equ label, expression
```

Here, *`label`* is a legal Gas identifier, and *`expression`* is a constant arithmetic expression (typically a single literal constant value). The following example defines the symbol `dataSize` to be equal to 256:

```
.equ dataSize, 256
```

Constant declarations, or *equates* in Gas terminology, may appear anywhere in your Gas source file prior to their first use: in a `.data` section, in a `.text` section, or even outside any sections.

Once you define a constant symbol with `.equ`, it cannot be further modified in the source file during assembly. If you need to reassign the value associated with a label during assembly (see Chapter 13 for reasons you'd want to do this), use the `.set` directive:

```
    .set valueCanChange, 6
      .
      .  // valueCanChange has the value 6 here.
      .
    .set valueCanChange, 7

// From this point forward, valueCanChange has the value 7.
```

Equates can specify textual arguments as well as numeric constants.

Because Gas will run your source files through the CPP if the filename suffix is *.S*, you can also use the CPP `#define` macro definition to create

named constants. Although the `.equ` directive is probably the better choice, the C macro form offers a few advantages, like allowing arbitrary textual substitution, not just numeric expression substitution. For more on this, see Chapter 13.

1.7.4 Creating Register Aliases in Gas and Substituting Text

As you begin to write more complex ARM assembly language programs, you'll discover that the 32 general-purpose register names (X0 to X30 and SP) obscure the meaning of their values in the program. It's been decades since BASIC supported only variable names like A0, A1, B2, and Z3. To avoid returning to those days by using meaningless two-character names, Gas provides a way to create more meaningful aliases of register names in your programs: the `.req` directive.

The syntax of the `.req` directive is

```
symbolicName  .req  register
```

where *`symbolicName`* is any valid Gas identifier and *`register`* is one of the 32- or 64-bit register names. After this statement in the source file, if you use *`symbolicName`* in place of *`register`*, Gas will automatically substitute that register for the name.

Sadly, the `.req` directive works only for creating register aliases; you can't use it as a general-purpose text-substitution facility. However, if you name your assembly language source files with *.S*, Gas/GCC will first run your source file through the CPP. This allows you to embed C/C++ `#define` statements in your assembly source file, and the CPP will happily expand any symbols you define in such statements throughout your source file. The following example demonstrates using `#define`:

```
#define arrayPtr X0

// From this point forward, you can use arrayPtr in place of X0.
```

Typically, you'll use `.req` for register aliases and `#define` for any other textual substitutions in the source file, though my personal preference is to use the `#define` statement for both purposes in this book. Since `#define` also accepts parameters, it's flexible. Gas also supports textual substitution via *macros*; see Chapter 13 for more on this.

1.8 Basic ARM Assembly Language Instructions

Thus far, the programming examples in this chapter have consisted of functions that use only the `ret` instruction. This section describes a few more instructions to get you started writing more meaningful assembly language programs.

1.8.1 ldr, str, adr, and adrp

One solidly RISC feature of ARM is its use of *load/store architecture*. All computational activity takes place in the ARM's registers; the only instructions that access main memory are those that load a value from memory or store a value into memory.

Although the ARM64 has many general-purpose registers for holding variable values (and thus can avoid using memory), most applications use more variable data than can fit in all the registers. This is especially true for larger objects like arrays, structs, and strings. Furthermore, programming conventions—known as the *application binary interface (ABI)*, discussed later in this chapter—often reserve many of ARM's registers so they cannot be used to hold application variables for any length of time. So variables must be placed in main memory and accessed via these `ldr` (load) and `str` (store) instructions.

This is the generic syntax for the load and store instructions

```
ldr{size}  reg, mem
str{size}  reg, mem
```

where *size* is either absent or one of the character sequences `b`, `h`, `sb`, `sh`, or `sw`; *reg* is one of the ARM's 32- or 64-bit registers; and *mem* is a memory addressing mode that specifies where to fetch the data from in memory. The `ldr` instruction loads the register specified by *reg* from the memory location specified by *mem*. The `str` instruction stores the value held in the register operand into the memory location.

Chapter 2 discusses the *size* operand in greater depth, but this chapter largely ignores the size suffixes on the `ldr` and `str` instructions. Without a size prefix, the *reg* operand determines the operation's size. If *reg* is X*n*, the instruction transfers 64 bits; if it's W*n*, then the instruction transfers 32 bits.

The *mem* operand is either the name of a variable in your program, typically in the `.data` section (Linux only), or a register name surrounded by square brackets ([]). In this latter case, the register holds the numeric memory address of the memory location to access. See Chapter 3 for more on *mem*.

LINUX VS. MACOS:
POSITION-INDEPENDENT EXECUTABLES

One major OS policy difference between macOS and Linux ARM assembly language is that macOS requires the use of *position-independent executables (PIE)*, while Linux only encourages them. PIE allows the system to load the various sections of a program into different memory locations at runtime. This is important for two reasons: it enables the use of shared libraries and addresses security concerns.

(continued)

Shared libraries contain code shared among applications. An OS will load only one copy of a library's code into physical memory and share that single copy among multiple running applications. However, the library code has to sit at an address in a given application's memory space in order for that application to call functions within the library, yet the address used for a library function in one application may already be in use when a second application attempts to load the library. Therefore, the second application will need to call that function at a different address in its own virtual memory address space.

If two separate copies of the function were made in real memory, calling the function at different addresses in memory wouldn't be an issue; the first application could locate the function at an address completely independent of the second. However, one major reason to use shared libraries is to share the exact same code in real (physical) memory.

The OS resolves the virtual memory address conflicts by programming the memory management unit (MMU) to map that physical memory to two separate virtual memory addresses in the two applications. However, for this to work, the library code must not access any *absolute* (fixed) memory addresses; if it does, the second application maps the function to a different address in the virtual memory address space, and the sharing concept fails. For example, if the library code transfers control from location 0x12_3456 to location 0x12_3500 in memory, this transfer will fail if the code is moved to a different location; the application will still want to transfer to location 0x12_3500, even though the code has moved elsewhere.

For machine instructions, this is not a problem. The ARM instructions that transfer control typically use *program-counter-relative (PC-relative)* addressing. Rather than transfer control to a fixed location (like 0x12_3500), they transfer control to a location relative to the current location. That is, they transfer to a location a certain number of bytes before or after the current value in the PC register. If the code moves to a different fixed address in memory, the instruction will still transfer to the correct place, because the destination location moved along with the current instruction.

Unfortunately, this scheme doesn't work for data. If a shared library accesses global data, the OS has to create a separate block of data for each application that uses the shared library; you typically don't want one application to affect the data in another application. That means data addresses must be relocatable as well.

The ARM CPU can also access data at locations relative to the PC, so in theory, the OS can remap the data to a different location for each application, as it does for the code. However, using PIE is still wise for security reasons. In the past, various hacks have taken advantage of the fact that the data for a shared library sits at a fixed offset from the code. To help prevent such exploits, macOS and Linux support *address space layout randomization (ASLR)*. With this feature, the OS randomly assigns a different address to the code and data sections of a program (or library) code when loading it into memory. This makes it more difficult for a hack to exploit the code.

> ASLR also makes it slightly more difficult (and less efficient) to access that data. Worse still, Linux and macOS provide completely different mechanisms for accessing position-independent data. This is transparent to HLL programmers, but it has to be handled explicitly when writing assembly language code. This creates problems in a book such as this one, where the goal is to provide example code that compiles and runs on different OSes. As for other Linux versus macOS issues, the *aoaa.inc* header file contains macros and other code to resolve these issues. I'll have more to say about PIE in Chapter 3.

Because macOS requires that your applications be written in a position-independent fashion (as we just discussed in "Linux vs. macOS: Position-Independent Executables"), you will not be able to use an `ldr` instruction of this form:

```
ldr x0, i64   // i64 is a 64-bit variable declared
              // in the .data section by using .dword.
```

To access the `i64` variable, you must first load its address into a 64-bit register, then access that data by using the *register-indirect addressing mode*, or X*n*. To do so, place the address of the variable you want to access in the register by using the `adr` and `adrp` instructions:

```
adr  reg64, mem
adrp reg64, mem
```

Here, *reg*$_{64}$ is the name of a 64-bit general-purpose register, and *mem* is a memory addressing mode, like the name of a global variable. The `adr` instruction loads *reg* with the address of the memory variable, which must be ±1MB from the `adr` instruction if the operand is just the name of a variable (like `i64` from the previous example). The `adrp` instruction loads the 64-bit destination register with the page (4,096-byte boundary) containing the memory object. That value will have the LO 12 bits containing all 0s.

Because of macOS's PIE requirements, it doesn't take kindly to instructions such as the following:

```
ldr x0, i64
```

On the Mac, you must use the register-indirect addressing mode to access a global variable. Unfortunately

```
adr x1, i64
```

fails for the same reason: you're not allowed to specify the name of a global variable.

In this book, to get the address of a global variable into a register under macOS, we'll use the following statement:

```
lea reg, mem
```

The `lea` (load effective address) macro, included in *aoaa.inc*, will expand into two instructions (different ones depending on your OS). These instructions will load the address of the second operand (*mem*) into the 64-bit register specified by the first operand (*reg*). You can use `lea` in any projects where you've included *aoaa.inc* at the beginning of your source file.

As noted, the *aoaa.inc* macros make the code in this book portable between OSes. However, you can choose to go with the appropriate OS-specific code, which can sometimes be more efficient, once you master basic ARM assembly language programming. See Chapter 7 for more details on `lea`.

To conclude this discussion of taking the address of a variable, let's recap how to load and store values by using `ldr` and `str`:

```
        .data
i64:    .dword  0  // This also requires the aoaa.inc file.
         .
         .
         .
// Load i64's value into X0:

        lea x0, i64
        ldr x0, [x0]
         .
         .
         .
// Store X0 into i64:

        lea x1, i64
        str x0, [x1]
```

When loading X0 with a variable's value, you can first load X0 with the address of the variable and then load X0 indirectly from the location held in X0. This winds up using only a single register. However, when storing data to memory, you need a second register to hold the address (X1 in this example).

If you are referencing a particular variable several times within a small section of code, it's more efficient to load its address into a register just once and reuse that register value multiple times, rather than constantly reloading the address:

```
lea x1, i64
ldr x0, [x1]
 .
 .
 .
str x0, [x1]
```

Of course, this means you can't use the register for any other purpose while it holds `i64`'s address. Fortunately, for just this reason, the ARM64 has lots of registers.

1.8.2 mov

Beyond the `ldr` and `str` instructions, the `mov` instruction handles two additional data movement operations: moving data between a pair of registers and copying a constant into a register. The generic syntax for `mov` is as follows:

```
mov reg_dest, reg_src
mov reg_dest, #constant
```

The first `mov` instruction copies the data in the source register (*reg$_{src}$*) into the destination register (*reg$_{dest}$*). This instruction is equivalent to the C/C++ statement *reg$_{dest}$* = *reg$_{src}$*;. The source and destination registers can be any of the general-purpose registers but must be the same size (32 or 64 bits).

The second `mov` instruction moves a small integer constant into the destination register. Constants encoded as part of the instruction are known as *immediate constants* and are generally preceded by a `#` character (though Gas often allows you to drop the `#` when specifying literal numeric constants). Chapter 2 discusses limitations on constants, but for now, assume any constant less than ±2,047 will work.

Here are two examples of the `mov` instruction:

```
mov x1, x0   // X1 = X0
mov x2, #10  // X2 = 10
```

There are many additional variants of `mov`, covered in depth in later chapters. For example, if you encounter a constant you cannot load into a register with a single `mov` instruction, other variants of `mov` let you load any arbitrary 32- or 64-bit constant by using two to three instructions. In the meantime, this variant of the `ldr` instruction will load any constant into a register:

```
ldr reg, =veryLargeConstant
```

The assembler will simply store *veryLargeConstant* in a memory location somewhere and then load the contents of that memory location to the specified register. Use this handy pseudo-instruction when you need to load a large constant into a register with a single instruction.

1.8.3 add and sub

The `add` and `sub` instructions handle simple arithmetic on the ARM CPU. These instructions take many forms covered more thoroughly in the next couple of chapters. Their basic forms are the following:

```
add  reg_dest, reg_lsrc, reg_rsrc  // reg_dest = reg_lsrc + reg_rsrc
add  reg_dest, reg_lsrc, #const // reg_dest = reg_lsrc + const
adds reg_dest, reg_lsrc, reg_rsrc  // reg_dest = reg_lsrc + reg_rsrc
adds reg_dest, reg_lsrc, #const // reg_dest = reg_lsrc + const
sub  reg_dest, reg_lsrc, reg_rsrc  // reg_dest = reg_lsrc - reg_rsrc
sub  reg_dest, reg_lsrc, #const // reg_dest = reg_lsrc - const
subs reg_dest, reg_lsrc, reg_rsrc  // reg_dest = reg_lsrc - reg_rsrc
subs reg_dest, reg_lsrc, #const // reg_dest = reg_lsrc - const
```

Here, *reg*$_{dest}$, *reg*$_{lsrc}$, and *reg*$_{rsrc}$ are 32- or 64-bit registers (which must all be the same size for a given instruction), and *const* is an immediate constant in the range 0 to 4,095. You'll learn to specify larger constants later, but these forms are sufficient for the example programs in the next few chapters.

NOTE *Some assemblers allow a range of –4,095 to +4,095 and swap the* `add` *and* `sub` *instructions if the immediate constant is negative.*

The instructions with the `s` suffix affect the condition code flags. They set the flags according to the conditions specified in the following list:

- **N** Set if the arithmetic operation produces a negative result (high-order, or HO, bit is set); clear if it produces a nonnegative result (HO bit is clear).
- **Z** Set if the arithmetic operation produces a 0 result; clear if it produces a nonzero result.
- **C** Set if the addition operation produces an unsigned overflow (carry out of the HO bit). Clear if a subtraction operation produces a borrow (unsigned underflow), and set otherwise.
- **V** Set if the arithmetic operation produces a signed overflow (carry out of the next-to-HO bit).

The following instructions negate their source operands, because they subtract the source register from 0 (remember that WZR and XZR are the zero registers and return 0 when read):

```
sub reg_dest32, wzr, reg_src32 // reg_dest32 = - reg_src32
sub reg_dest64, xzr, reg_src64 // reg_dest64 = - reg_src64
```

Gas provides synonyms for these instructions:

```
neg  reg_dest32, reg_src32 // Negate instruction, no flags
negs reg_dest32, reg_src32 // Negate instruction, w/flags
```

```
neg  reg_dest64, reg_src64 // Negate instruction, no flags
negs reg_dest64, reg_src64 // Negate instruction, w/flags
```

These forms are a little easier to read.

1.8.4 bl, blr, and ret

Calling procedures and functions is handled by the `bl` (branch and link) and `blr` (branch and link through register) instructions. Here's their syntax

```
bl  label
blr Xn
```

where *label* is a statement label preceding code in the `.text` section, and `Xn` represents one of the 64-bit registers. These two instructions copy the address of the next instruction (following the `bl` or `blr` instruction) into the link register (LR/X30), then transfer control either to the target *label* or to the address specified by the contents of X*n*.

The `bl` instruction does have a minor limitation: it can transfer control only to a statement label within ±128MB of the current instruction. This is generally far more than enough for any function you'll write. In theory, if the OS loads code into another section (besides `.text`), it could be placed sufficiently far away that it would exceed this range. The OS linker will probably complain if this occurs. This book generally places all code within the `.text` section, as it would be rare for such programs to exceed this limitation.

The `blr` instruction copies the full 64-bit address from X*n* into the PC (after copying the address of the next instruction into LR). Therefore, `blr` does not have the range limitation of the `bl` instruction. If you ever do encounter the range limitation when using `bl`, overcome it by using the following sequence:

```
lea x0, farAwayProcedure
blr x0
```

This will load the address of *farAwayProcedure* into X0 (no matter where it appears in memory), then transfer control to that procedure via `blr`.

The `ret` instruction has appeared in several examples up to this point. It copies the contents of the LR (X30) register into the PC. Assuming that LR was loaded with a value as a result of executing the `bl` or `blr` instruction, this returns control to the instruction following the `bl/blr`.

The `bl`, `blr`, and `ret` instructions have one issue: the ARM architecture tracks only a single subroutine call with the LR register. Consider the following code fragment:

```
someFunc:
        ret
         .
         .
         .
```

```
main:
        bl someFunc
        ret
```

When the OS calls the main program, it loads the LR register with the return address back to the OS. Normally, when the main program completes execution, its `ret` instruction transfers control to this location. However, that's not the case in this example: when the main program begins execution, it immediately calls `someFunc` with the `bl` instruction. This instruction copies its return address (the address of the main program's `ret` instruction) into the LR register, wiping out the OS's return address currently residing there. When `someFunc` executes the return instruction, it returns control back to the main program.

Upon return from `someFunc`, the main program executes the `ret` instruction. However, the LR register now contains the return address of the `someFunc` call, which is the address of the `ret` instruction in the main program, so control transfers there, re-executing `ret`. The LR register's value hasn't changed; it still points at that `ret` instruction, meaning this code enters an infinite loop continuously executing the return and transferring control back to the return (where LR continues to point).

Chapter 3 discusses the high-level solution to this problem. For the time being, we must save the LR register value in the main program before calling `someFunc`. One quick-and-dirty way to do this is to copy it into another (unused by main) register and restore LR before the final return:

```
someFunc:
        ret
         .
         .
         .
main:
        mov x1, lr
        bl someFunc
        mov lr, x1
        ret
```

This code saves the return address (in LR) in the X1 register and restores it after returning from `someFunc` (the call to `someFunc` overwrote the value in LR).

In general, saving the return address in the X1 register is a bad idea, because the ARM's designers reserve X1 for passing parameters. (Using X1 worked in this example because `someFunc` doesn't have any parameters, as it just returns to its caller.) The next section covers in greater depth which registers are reserved for various purposes.

1.9 The ARM64 Application Binary Interface

A CPU's *application binary interface (ABI)* describes how programs should use registers, pass parameters between functions, represent data, and many

other conventions. Its primary purpose is to provide interoperability among programming languages and systems. The ARM64's ABI, for example, describes the conventions that allow C/C++ programs to call functions written in Swift, Pascal, and other languages. Since the GCC (and Clang) compilers follow these rules, you must also follow them to pass information between your assembly language code and code written in an HLL such as C/C++.

An ABI is a *convention*, not an absolute rule. It is a contract between the code being called and the code making the call. When writing your own assembly language functions to be called by your own assembly language code, you are under no obligation to use the ABI and can use whatever inter-code communication scheme you like. However, if you call C/C++ code from your assembly functions, or if your assembly code is being called from C/C++, you must follow the ARM64 ABI. Since this book uses a considerable mixture of C/C++ and assembly code, understanding the ARM64 ABI is critical for our purposes.

1.9.1 Register Usage

The ARM64 ABI reserves some of its 32 general-purpose registers for specific uses and defines whether registers are *volatile* (meaning you don't have to preserve their values) or *nonvolatile* (meaning that you must preserve their values within a function). Table 1-2 describes the special purposes and volatility of the 32 ARM registers.

Table 1-2: ARM64 ABI Register Conventions

Register	Volatile	Special meaning
X0/W0	Yes	Pass parameter 1 here, return function results here. Registers X0 through X7 can also be used as a scratchpad/temporary/local variable if not used as a parameter.
X1/W1	Yes	Pass parameter 2 here, return function results here.
X2/W2	Yes	Pass parameter 3 here, return function results here.
X3/W3	Yes	Pass parameter 4 here, return function results here.
X4/W4	Yes	Pass parameter 5 here, return function results here.
X5/W5	Yes	Pass parameter 6 here, return function results here.
X6/W6	Yes	Pass parameter 7 here, return function results here.
X7/W7	Yes	Pass parameter 8 here, return function results here.
X8/W8	Yes	Pointer to large function return results (for example, a large C structure returned by value).
X9/W9 X10/W10 X11/W11 X12/W12 X13/W13 X14/W14 X15/W15	Yes Yes Yes Yes Yes Yes Yes	Can be used as a scratchpad/temporary/local variable.

(continued)

Table 1-2: ARM64 ABI Register Conventions *(continued)*

Register	Volatile	Special meaning
X16/W16/IP0	Yes, but . . .	You can use this register as a temporary variable, but its value may change across the execution of a control-transfer instruction; the system linker/loader may use this register to create a *veneer*, also known as a *trampoline* (more on this in Chapter 7).
X17/W17/IP1	Yes, but . . .	You can use this register as a temporary variable, but its value may change across the execution of a control-transfer instruction; the system linker/loader may use this register to create a *veneer*, also known as a *trampoline* (more on this in Chapter 7).
X18/W18/Plat	No access	This register is reserved for use by the OS, and application programs must not modify its value. Under macOS, you definitely must not modify this register; under Linux, you may get away with using this register if you preserve its value, but the safe choice is to avoid using this register.
X19/W19 X20/W20 X21/W21 X22/W22 X23/W23 X24/W24 X25/W25 X26/W26 X27/W27 X28/W28	No No No No No No No No No No	A function using this register must save and restore the register's value so that it contains its original value when the function returns.
X29/W29/FP	N/A	Reserved for use as the system frame pointer.
X30/W30/LR	N/A	Reserved for holding function return addresses.
SP /X31/W31	N/A	Reserved for use as the system stack pointer.

Conveniently, when using volatile registers in a function, you don't have to preserve (save and restore) their values within the function. However, this means that you also cannot expect them to maintain their values across any functions you call via `bl` or `blr`. Nonvolatile registers will maintain their values across function calls you make, but you must explicitly preserve their values if you modify them within your functions.

1.9.2 Parameter Passing and Function Result Conventions

Chapter 5 provides a complete discussion of parameter passing and function results in assembly language. However, when calling functions written in a different languages (particularly HLLs), you must adhere to the conventions that language uses. Most HLLs use the ARM ABI as the convention for passing parameters.

The ARM ABI uses registers X0 through X7 to pass up to eight integer parameters to a function. These parameters can be 8-, 16-, 32-, or 64-bit entities. You pass the first parameter in X0, the second in X1, and so on. To pass fewer than eight parameters, simply ignore the additional registers in this set. Chapter 5 discusses how to pass more than eight parameters and how to pass data types larger than 64 bits, including arrays and structs. Chapter 6 covers how to pass floating-point values to a function.

You can also return function results in these registers. Most functions return integer results in X0. If you're returning a large object by value, like a structure, array, or string, you typically use X8 to return a pointer to that data object. Chapter 6 discusses returning floating-point function results.

Registers X0 through X7 are volatile, meaning you can't expect a called function to preserve the original register values on return. This is true even if you don't use all eight registers to pass parameter values. If you want to preserve a value across function calls, use a nonvolatile register.

1.10 Calling C Library Functions

All the coding examples in this book so far have immediately returned to the OS, apparently without accomplishing anything. While it is theoretically possible for a pure assembly language program to produce its own output, it takes a lot of work and is largely beyond the scope of this book. Instead, this book calls prewritten C/C++ library code to do the I/O. This section discusses how this is done.

Most other books on assembly language that use libraries in this way call the OS by using available *application programming interfaces (APIs)*. This is a reasonable approach, but such code is tied to the particular OS for which the calls are made (see Chapter 16 for examples). This book instead relies on library functions written in the C stdlib, since it's available on many OSes.

In most introductory programming books, the first programming example provided is the venerable "Hello, world!" program. Here's that program written in C:

```
#include <stdio.h>

int main( int argc, char **argv )
{
   printf( "Hello, world!\n" );
}
```

Except for an actual `printf()` statement, the assembly language source files given thus far have fulfilled the purpose of the "Hello, world!" example: learning how to edit, compile, and run a simple program.

Most of this book uses the C `printf()` function to handle program output to the console. This function requires one or more arguments—that is, a *variable-length* parameter list. The first argument is the address of a *format string*. If that string requires it, additional parameters provide data to convert to string form. For the "Hello, world!" program, the format string (`"Hello, world!\n"`) is the only argument.

The C stdlib—and all C functions, for that matter—adheres to the ARM ABI. Therefore, `printf()` expects its first argument, the format string, in the X0 register. Instead of trying to pass a string (with 14 characters, including the newline) in a 64-bit register, we pass the address of that string in memory. If we put the string `"Hello, world!\n"` in the `.text` section along with the

program (out of the way, so the CPU doesn't try to execute it as code), then we can compute the address of that string by using the `lea` macro:

```
hwStr:   .asciz  "Hello, world!\n"
           .
           .
           .
         lea    x0, hwStr
```

Once we have this string address in X0, calling `printf()` prints that string to the standard output device:

```
lea x0, hwStr
bl  printf
```

To run, this program must be linked against the C stdlib and a small C/C++ program like the one in Listing 1-2. Rather than grabbing that program, I'll create a slightly better version in Listing 1-4 to use with almost every example program in the rest of this book.

```
// Listing1.4.cpp
//
// Generic C++ driver program to call AoAA example programs
// Also includes a "readLine" function that reads a string
// from the user and passes it on to the assembly language
// code
//
// Need to include stdio.h so this program can call "printf"
// and stdio.h so this program can call strlen.

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Extern "C" namespace prevents "name mangling" by the C++
// compiler:

extern "C"
{
    // asmMain is the assembly language code's "main program":

  ❶ void asmMain( void );

    // getTitle returns a pointer to a string of characters
    // from the assembly code that specifies the title of that
    // program (that makes this program generic and usable
    // with a large number of sample programs in "The Art of
    // ARM Assembly Language"):

  ❷ char *getTitle( void );

    // C++ function that the assembly
    // language program can call:
```

```
❸ int readLine( char *dest, int maxLen );

};

// readLine reads a line of text from the user (from the
// console device) and stores that string into the destination
// buffer the first argument specifies. Strings are limited in
// length to the value specified by the second argument
// (minus 1).
//
// This function returns the number of characters actually
// read, or -1 if there was an error.
//
// If the user enters too many characters (maxLen or
// more), this function returns only the first maxLen - 1
// characters. This is not considered an error.

int readLine( char *dest, int maxLen )
{
    // Note: fgets returns NULL if there was an error, else
    // it returns a pointer to the string data read (which
    // will be the value of the dest pointer):

    char *result = fgets( dest, maxLen, stdin );
    if( result != NULL )
    {
        // Wipe out the newline character at the
        // end of the string:

        int len = strlen( result );
        if( len > 0 )
        {
            dest[ len - 1 ] = 0;
        }
        return len;
    }
    return -1; // If there was an error
}

int main(void)
{
    // Get the assembly language program's title:

    char *title = getTitle();

    printf( "Calling %s:\n", title );
    asmMain();
    printf( "%s terminated\n", title );

}
```

This program contains a few additional features over Listing 1-2. First, the name of the assembly language function has changed to `asmMain()` ❶, the assembly language main program. This code also requires a second

assembly function, `getTitle()` ❷. This function, provided by the assembly language source code, returns a pointer to a zero-terminated string containing the program's title. The program displays this title before and after calling `asmMain()`.

The `readLine()` function appears in the C program that reads a line of text from the user and stores that text into a buffer specified by the caller ❸. You can call this function from the example assembly code, sparing you from having to write the function in assembly (it's grunt work better done in C). You'll see examples of this function call in later chapters.

This file (appearing as *Listing1-4.cpp* or *c.cpp* in the online code) requires the assembly code to provide a `getTitle()` function that returns the address of a string so the C program can display the name. This string is embedded in the assembly language source file, since most of the programs in this book use only one version of *c.cpp*. The `getTitle()` function is the same in every program

```
getTitle:
            lea     x0, title  // C expects pointer in X0.
            ret
```

where `title` is a zero-terminated string appearing elsewhere in your program (usually in the `.data` section). That declaration will usually take this form:

```
title:      .asciz "Listing1-5"
```

The `getTitle` function returns the address of this string to the *c.cpp* program. The string following the `.asciz` directive will typically be the name of the assembly language source file (I used *Listing1-5* in this example).

1.10.1 Assembling Programs Under Multiple OSes

We could easily bang out a "Hello, world!" program for Linux or macOS at this point, but the programs would be slightly different for each OS. So that we don't need to use a different include file for each OS, I've modified *aoaa.inc* to look for a couple of symbol definitions: `isMacOS` and `isLinux`. Both symbols must be defined with the CPP `#define` declaration, and one must be true (1) while the other is false (0). The *aoaa.inc* file uses these symbols to adjust the definitions present in the file for the appropriate OS.

In theory, we could use code like the following to define these symbols:

```
#define isMacOS (1)
#define isLinux (0)
#include "aoaa.inc"
```

However, this would force every example program to have two versions, one for macOS (the example just given) and one for Linux, containing the following statements:

```
#define isMacOS (0)
#define isLinux (1)
#include "aoaa.inc"
```

GCC has a preferable command line option that lets you define a preprocessor symbol and give it a value:

```
-D name=value
```

This way, the following commands will automatically define the symbol prior to assembling the *source.S* file:

```
g++ -D isMacOS=1 source.S
g++ -D isLinux=1 source.S
```

We can specify the OS from the command line in this way so that the source files (*source.S* and *aoaa.inc*) don't require any changes under either macOS or Linux. To avoid any extra typing required to assemble the program, we'll use a command line program known as a *shell script*.

While writing a shell script for this purpose, I also further automated the build process. The script, named *build*, accepts the base name of an example file without a suffix and automatically deletes any existing object or executable files with that base name (a *clean* operation, in Unix terminology). It then determines which OS *build* is running on and then automatically generates the appropriate GCC command line to build the example.

SHELL SCRIPTS VS. MAKEFILES

If you have experience developing software by using the command line, you may wonder why I haven't built the examples with a makefile. I discuss makefiles further in Chapter 15, but I've chosen not to use them here for a couple of reasons:

- If you don't already know the Make language, I'd prefer to put off teaching that until you've mastered a little more assembly language.
- Using Make would mean writing a separate makefile for each example program. However, the *build* shell script this section describes works for nearly all the example programs in this book.

For example, to build a file named *example.S*, you'd execute the following command:

```
./build example
```

Under Linux, this would generate the following command:

```
g++ -D isLinux=1 -o example c.cpp example.S
```

Under macOS, it would generate the following:

```
g++ -D isMacOS=1 -o example c.cpp example.S
```

The *build* script also supports a couple of command line options: `-c` and `-pie`. The `-c` (compile-only) option generates the following command line, which only assembles the assembly file to an object file; it does not compile *c.cpp*, nor does it produce an executable:

```
./build -c example
```

This executes the following command as appropriate:

```
g++ -c -D isMacOS=1 -o example.o example.S
```

or

```
g++ -c -D isLinux=1 -o example.o example.S
```

The `-pie` option applies only to Linux. It issues the appropriate commands to tell Linux to produce a position-independent executable file (by default, Linux produces a non-position-independent executable). Because macOS's assembler always produces PIE code, this option is ignored under macOS.

For the curious, I've provided the text for this shell script in the file *build* without further comment, as writing shell scripts is beyond the scope of this book:

```
#!/bin/bash
#
# build
#
# Automatically builds an Art of ARM Assembly
# example program from the command line
#
# Usage:
#
#    build {options} fileName
#
# (no suffix on the filename.)
#
# options:
#
#    -c: Assemble .S file to object code only.
#    -pie: On Linux, generate a PIE executable.

fileName=""
compileOnly=" "
```

```
pie="-no-pie"
cFile="c.cpp"
lib=" "
while [[ $# -gt 0 ]]
do

    key="$1"
    case $key in

        -c)
        compileOnly='-c'
        shift
        ;;

        -pie)
        pie='-pie'
        shift
        ;;

        -math)
        math='-lm'
        shift
        ;;

        *)
        fileName="$1"
        shift
        ;;
    esac
done

# If -c option was provided, only assemble the .S
# file and produce an .o output file.
#
# If -c not specified, compile both c.cpp and the .S
# file and produce an executable:

if [ "$compileOnly" = '-c' ]; then
    objectFile="-o $fileName".o
    cFile=" "
else
    objectFile="-o $fileName"
fi

# If the executable already exists, delete it:

if test -e "$fileName"; then
    rm "$fileName"
fi

# If the object file already exists, delete it:

if test -e "$fileName".o; then
    rm "$fileName".o
fi
```

```
# Determine what OS you're running under (Linux or Darwin [macOS]) and
# issue the appropriate GCC command to compile/assemble the files.

unamestr=$(uname)
if [ "$unamestr" = 'Linux' ]; then
    g++ -D isLinux=1 $pie $compileOnly $objectFile  $cFile $fileName.S $math
elif [ "$unamestr" = 'Darwin' ]; then
    g++ -D isMacOS=1  $compileOnly $objectFile $cFile  $fileName.S -lSystem $math
fi
```

Check out a book on GNU's bash shell interpreter if you want to learn how this works (see section 1.12, "For More Information," on page 43).

The *build* shell script is available in electronic form at *https://artofarm.randallhyde.com*. Execute the following command to make this file executable from the bash command line on your Linux or macOS system:

```
chmod u+x build
```

This makes the *build* script executable. See Appendix D for more information about the `chmod` command.

1.10.2 Writing a "Hello, World!" Program

You finally have the pieces in place to write a complete "Hello, world!" program, as shown in Listing 1-5.

```
// Listing1-5.S
//
// The venerable "Hello, world!" program, written
// in ARM assembly by calling the C stdlib printf
// function
//
// aoaa.inc is the Art of ARM Assembly include file.
//
// This makes asmMain global and
// automatically converts it to _asmMain
// if this program is being assembled under macOS.
// It also converts printf to _printf for macOS.

            #include "aoaa.inc"

            .data

❶ title:    .asciz  "Listing 1-5"
  saveLR:   .dword  0       // Save LR here.
  hwStr:    .asciz  "Hello, world!\n"

            .text

  // getTitle function, required by c.cpp, returns the
  // name of this program. The title string must
  // appear in the .text section:
```

```
            .align  2       // Code must be 4-byte aligned.

❷ getTitle:
            lea     x0, title
            ret


// Here's the main function called by the c.cpp function:

asmMain:

// LR is *highly* volatile and will be wiped
// out when this code calls the printf() function.
// We need to save LR in memory somewhere so we
// can return back to the OS using its value.
// For now, save it in the saveLR global
// variable:

            lea     x0, saveLR
            str     lr, [x0]

// Set up printf parameter (format string)
// and call printf():

          ❸ lea     x0, hwStr   // hwStr must be in .text.
            bl      printf      // Print the string.

// Back from printf(), restore LR with its original
// value so we can return to the OS:

          ❹ lea     x0, saveLR
            ldr     lr, [x0]

// Return to the OS:

            ret
```

The `title` string ❶ holds the program's title (`"Listing 1-5"` in this example). The `hwStr` variable holds the `Hello, world!` string that the main program will pass to the `printf()` function. The `getTitle()` function ❷ returns the address of the `title` string to the *c.cpp* program. As per the ARM ABI, this function returns the function result in the X0 register.

Upon entry into the `asmMain()` function (the assembly language main program), the code must preserve the contents of the LR register because the call to `printf()` will overwrite its value. This code saves the LR register (which holds the return address to the *c.cpp* main function) in the `saveLR` global variable in the `.data` section ❶.

NOTE *Saving the LR register value in this fashion is not good practice. In Chapter 3 you'll learn about the ARM stack and discover a much better place to save return addresses held in LR.*

The code that actually prints `Hello, world!` ❸ loads X0 with the `printf()` format string as per the ARM ABI, then calls `printf()` by using the `bl` instruction. Before returning to *c.cpp*, the assembly code must reload LR with the returned address held in `saveLR` ❹.

Here are the commands to build and run the program in Listing 1-5, along with the program's output:

```
$ ./build Listing1-5
$ ./Listing1-5
Calling Listing1-5:
Hello, world!
Listing1-5 terminated
```

You now have a functioning "Hello, world!" program in assembly language.

LINUX VS. MACOS: VARIADIC PARAMETERS

Passing parameters to functions with a variable number of parameters, such as `printf()`, works differently in the standard ARM ABI and the macOS variant. Linux, using the standard ABI, passes the first eight parameters in registers X0 through X7, as Table 1-2 describes. However, macOS unfortunately passes only the first parameter of a *variadic function* (a function with a variable number of parameters) in register X0. It passes all remaining parameters on the *stack* (described in Chapters 3 and 5).

To allow us to write code that will assemble and run on both OSes, the *aoaa.inc* include file comes to the rescue once again. This file contains six macros with the following names:

```
vparm2, vparm3, ..., vparm7
```

Each macro takes a single argument: the name of a variable in the `.data` section. These macros will load the specified variable into the appropriate location (a register or on the stack) for that parameter. For this to work under macOS, the following statement must appear at the very beginning of your `asmMain()` function:

```
sub sp, sp, #64
```

You must also include the following statement before the `ret` instruction at the end of your `asmMain()` function:

```
add sp, sp, #64
```

Chapter 5 fully explains the purpose of these instructions; just trust them for now (they are required for macOS and do no harm under Linux).

If you have two variables, `i` and `j`, declared as words in your `.data` section, here's how to print them by using `printf()`

```
lea x0, fmtStr // Parameter 1 is still passed in X0.
vparm2 i
vparm3 j
bl printf
```

where `fmtStr` is something like this:

```
fmtStr: .asciz "i=%d, j=%d\n"
```

We use `vparm2`, `vparm3`, and so on only for variadic functions. Functions with a fixed number of parameters use registers X0 through X7 for the first eight parameters on both Linux and macOS.

1.11 Moving On

This chapter equipped you with the prerequisites to start learning new assembly language features in the chapters that follow. You learned the basic syntax of a Gas program and the basic 64-bit ARM architecture, and how to use the *aoaa.inc* header file to make source files portable between macOS and Linux. You also learned how to declare some simple global variables, use a few machine instructions, and assemble a Gas program with C/C++ code so you can call routines in the C stdlib (using the *build* script file). Finally, you ran that program from the command line.

The next chapter introduces you to *data representation*, one of the main reasons for learning assembly language in the first place.

1.12 For More Information

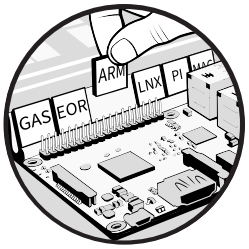
- For more information about the bash shell interpreter, visit the reference manual at *https://www.gnu.org/software/bash/manual/bash.html*.
- For more information about the GNU assembler, visit the reference manual at *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html*.
- You can find an online guide to 64-bit ARM assembly language at *https://modexp.wordpress.com/2018/10/30/arm64-assembly/*.
- If you're interested in programming ARM assembly language on Apple platforms, see *https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms*.
- The ARM developer portal at *https://developer.arm.com* provides generic information about ARM CPUs and ARM assembly language programming.

TEST YOURSELF

1. What is the name of the Gas executable program file?
2. What are the names of the three main system buses?
3. Which register holds the condition code bits?
4. How many bytes are consumed by the following data types?
   a. Word
   b. Dword
   c. Oword
   d. Double
5. What is the destination (register) operand size for the `lea` macro?
6. What is the name of the assembly language instruction you use to call a procedure or function?
7. What is the name of the assembly language instruction you use to return from a procedure or function?
8. What does *ABI* stand for?
9. In the ARM ABI, where do you return the following function return results?
   a. 8-bit byte values
   b. 16-bit word values
   c. 32-bit integer values
   d. 64-bit integer values
   e. 64-bit pointer values
10. Where do you pass the first, second, third, and fourth parameters to an ARM ABI-compatible function?

2

DATA REPRESENTATION AND OPERATIONS



A major stumbling block many beginners encounter when learning assembly language is the common use of the binary and hexadecimal numbering systems. However, the advantages of these systems far outweigh their disadvantages: they greatly simplify the discussion of other topics, including bit operations, signed numeric representation, character codes, and packed data.

This chapter discusses the following:

- The binary and hexadecimal numbering systems
- Binary data organization (bits, nibbles, bytes, half words, words, and double words)
- Signed and unsigned numbering systems
- Arithmetic, logical, shift, and rotate operations on binary values
- Bit fields and packed data

- Floating-point and binary-code decimal formats
- Character data

The remainder of this book depends on your understanding of these basic concepts. If you are already familiar with these terms from other courses or study, you should still skim this material to be sure you're not missing anything, and to learn the instructions this chapter introduces, before proceeding to the next one. If you are unfamiliar or only partly familiar with this material, study it carefully before proceeding. Don't skip any sections: *all the material in this chapter is important*!

2.1 Numbering Systems

Most modern computer systems do not use the decimal (base-10) system to represent numeric values. Instead, they typically use a binary numbering system. This is because the binary (base-2) numbering system more closely matches the electronic circuitry used to represent numeric values in a computer system.

2.1.1 Decimal

You've been using the decimal numbering system for so long that you probably take it for granted. When you see a number like 123, you don't think about the value 123; rather, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents the following:

$$(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0)$$

or

$$100 + 20 + 3$$

In a decimal *positional numbering system*, each digit appearing to the left of the decimal point represents a value from 0 to 9 multiplied by an increasing power of 10. Digits appearing to the right of the decimal point represent a value from 0 to 9 multiplied by an increasing negative power of 10. For example, the value 123.456 means this:

$$(1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (4 \times 10^{-1}) + (5 \times 10^{-2}) + (6 \times 10^{-3})$$

or

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

2.1.2 Binary

Most modern computer systems operate using *binary* logic. The computer uses two voltage levels (usually 0 V and 2.4 to 5 V) to represent values. These two levels can represent exactly two unique values. These could be

any two values, but they typically represent the values 0 and 1, the two digits in the binary numbering system.

The binary numbering system works just like the decimal numbering system, except binary allows only the digits 0 and 1 (rather than 0 to 9) and uses powers of 2 rather than powers of 10. Therefore, converting a binary number to decimal is easy. For each 1 in a binary string, add 2^n, where n is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents the following:

$$(1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$$

$$= 128_{10} + 64_{10} + 8_{10} + 2_{10}$$

$$= 202_{10}$$

Converting decimal to binary is slightly more difficult. You must find those powers of 2 that, when added together, produce the decimal result.

A simple way to convert decimal to binary is the *even/odd, divide-by-2* algorithm, comprising the following steps:

1. If the number is even, emit a 0. If the number is odd, emit a 1.
2. Divide the number by 2 and throw away any fractional component or remainder.
3. If the quotient is 0, the algorithm is complete.
4. If the quotient is not 0 and is odd, insert a 1 before the current string; if the number is even, prefix your binary string with 0.
5. Go back to step 2 and repeat.

Binary numbers, although they have little importance in HLLs, appear everywhere in assembly language programs, so make sure you're comfortable with them.

In the purest sense, every binary number contains an infinite number of digits (or *bits*, which is short for *binary digits*). For example, you can represent the number 5 with any of the following:

```
101
00000101
0000000000101
. . . 000000000000101
```

Any number of leading-zero digits may precede the binary number without changing its value. Because the ARM typically works with groups of 8 bits, this book will zero-extend all binary numbers to a multiple of 4 or 8 bits. Following this convention, you'd represent the number 5 as 0101_2 or 00000101_2.

To make larger numbers easier to read, I will often separate each group of 4 binary bits with an underscore. For example, I will write the binary value 1010111110110010 as 1010_1111_1011_0010. (Gas does not actually

allow you to insert underscores into the middle of a binary number; I use this convention just for readability purposes.)

The usual convention is to number each bit as follows: the rightmost bit in a binary number is bit position 0, and each bit to the left is given the next successive bit number. An 8-bit binary value uses bits 0 to 7:

$$X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

A 16-bit binary value uses bit positions 0 to 15:

$$X_{15}\ X_{14}\ X_{13}\ X_{12}\ X_{11}\ X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

A 32-bit binary value uses bit positions 0 to 31, and so on.

Bit 0 is the *low-order (LO)* bit; some refer to this as the *least significant bit*. The leftmost bit is called the *high-order (HO)* bit, or the *most significant bit*. I'll refer to the intermediate bits by their respective bit numbers.

In Gas, you can specify binary values as a string of 0 or 1 digits beginning with the sequence 0b—for example, 0b10111111.

2.1.3 Hexadecimal

Unfortunately, binary numbers are verbose: representing the value 202_{10} requires eight binary digits but only three decimal digits. When dealing with large values, binary numbers quickly become unwieldy. Since the computer "thinks" in binary, however, using the binary numbering system is convenient when creating values for the computer to use. Although you can convert between decimal (which humans tend to be most comfortable with) and binary, the conversion is not a trivial task. Additionally, many assembly language constants are easier to read and understand when written in binary (rather than decimal), so it's often a better idea to use binary.

The *hexadecimal* (base-16) numbering system solves many of the problems inherent in the binary system: hexadecimal numbers are compact, and it's simple to convert them to binary, and vice versa. For this reason, most engineers use the hexadecimal numbering system rather than binary.

Because the *radix* (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents a certain value multiplied by a successive power of 16. For example, the number $1{,}234_{16}$ is equal to this:

$$(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$$

or

$$4{,}096_{10} + 512_{10} + 48_{10} + 4_{10} = 4{,}660_{10}$$

Each hexadecimal digit can represent one of 16 values from 0 to 15_{10}. Because there are only 10 decimal digits, you need 6 additional digits to represent the values in the range 10_{10} to 15_{10}. Rather than create new symbols for these digits, the convention is to use the letters A to F. The following are examples of valid hexadecimal numbers:

1234_{16}

$DEAD_{16}$

$BEEF_{16}$

$0AFB_{16}$

$F001_{16}$

$D8B4_{16}$

Because you'll often need to enter hexadecimal numbers into the computer system, and on most computer systems you cannot enter a subscript to denote the radix of the associated value, you'll need a different mechanism for representing hexadecimal numbers. In this book, I use the following Gas conventions:

- All hexadecimal values have a 0x prefix (for example, 0x123A4 and 0xDEAD).
- All binary values begin with a 0b sequence (for example, 0b10010).
- Decimal numbers do not have a prefix character.
- If the radix is clear from the context, I may drop the 0x or 0b prefix characters.

Gas also allows the use of octal (base-8) numbers that begin with a leading 0 and contain only the digits 0 through 7. This book, however, does not use octal numbers.

Here are examples of valid hexadecimal numbers using Gas notation:

```
0x1234
0xDEAD
0xBEEF
0xAFB
0xF001
0xD8B4
```

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Table 2-1 provides all the information you need to convert any hexadecimal number into a binary number, or vice versa.

To convert a hexadecimal number into a binary number, substitute the corresponding 4 bits for each hexadecimal digit in the number. For example, to convert 0xABCD into a binary value, convert each hexadecimal digit according to Table 2-1: A becomes 1010, B becomes 1011, C becomes 1100, and D becomes 1101, giving you the binary value 1010_1011_1100_1101.

Table 2-1: Binary/Hexadecimal Conversion

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Converting a binary number into hexadecimal format is almost as easy:

1. Pad the binary number with 0s to make sure that the number contains a multiple of 4 bits. For example, given the binary number 1011001010, add 2 bits to the left of the number so that it contains 12 bits: 001011001010.
2. Separate the binary value into groups of 4 bits. In this example, you'd get 0010_1100_1010.
3. Look up these binary values in Table 2-1 and substitute the appropriate hexadecimal digits: 0x2CA.

Contrast this with the difficulty of conversion between decimal and binary, or decimal and hexadecimal!

Because you'll need to convert between hexadecimal and binary over and over again, take a few minutes to memorize the conversion table. Even if you have a calculator that can do the conversion for you, manual conversion is much faster and more convenient once you get the hang of it.

2.2 Numbers vs. Representation

Many people confuse numbers and their representation. Beginning assembly language students often ask, "I have a binary number in the W0 register; how do I convert that to a hexadecimal number in the W0 register?" The answer is, "You don't." Although one could make a strong argument

that numbers in memory or in registers are represented in binary, it's best to view values in memory or in a register as abstract numeric quantities. Strings of symbols like 128, 0x80, or 0b10000000 are not different numbers; they are simply different representations for the quantity that people refer to as "one hundred twenty-eight." Inside the computer, a number is a number regardless of representation; the only time representation matters is when you input or output the value in a human-readable form.

Pure assembly language has no generic print or write functions you can call to display numeric quantities as strings on your console. Chapter 9 demonstrates how to write your own procedures to handle this process. For the time being, the Gas code in this book relies on the C stdlib `printf()` function to display numeric values. Consider Listing 2-1, which converts various decimal values to their hexadecimal equivalents.

```
// Listing2-1.S
//
// Displays some numeric values on the console

#include "aoaa.inc"

            .data

// Program title, required by C++ code:

titleStr:   .asciz      "Listing 2-1"

// Format strings for three calls to printf():

fmtStrI:    .asciz      "i=%d, converted to hex=%x\n"
fmtStrJ:    .asciz      "j=%d, converted to hex=%x\n"
fmtStrK:    .asciz      "k=%d, converted to hex=%x\n"

// Some values to print in decimal and hexadecimal form:

            .align      2  // Be nice and word-align.
i:          .dword      1
j:          .dword      123
k:          .dword      456789
saveLR:     .dword      0

            .text
            .align      2      // Code must be word-aligned.
            .extern     printf // printf is outside this code.

// Return program title to C++ program:

getTitle:

// Load address of "titleStr" into the X0 register (X0 holds
```

```
// the function return result) and return back to the caller:

            lea x0, titleStr
            ret

// Here is the asmMain function:

            .global     asmMain
asmMain:
            sub         sp, sp, #64 // Magic instruction

// Save LR so we can return to C++ program:

            lea         x0, saveLR
            str         lr, [x0]

// Call printf three times to print the three values
// i, j, and k:
//
// printf( "i=%d, converted to hex=%x\n", i, i );

          ❶ lea         x0, fmtStrI
            vparm2      i           // Get parameter 2
            vparm3      i           // Get parameter 3
            bl          printf

// printf( "j=%d, converted to hex=%x\n", j, j );

          ❷ lea         x0, fmtStrJ
            vparm2      j
            vparm3      j
            bl          printf

// printf( "k=%d, converted to hex=%x\n", k, k );

          ❸ lea         x0, fmtStrK
            vparm2      k
            vparm3      k
            bl          printf

// Restore LR so we can return to C++ program:

            lea         x0, saveLR
            ldr         lr, [x0]

            add         sp, sp, #64  // Magic instruction
            ret                      // Returns to caller
```

To simulate the C statement

```
printf( "i=%d, converted to hex=%x\n", i, i );
```

the code must load three parameters ❶ into X0, X1, and X2: the address of the format string (`fmtStrI`) and the current value held in variable `i` (passed

twice, in X1 and X2). Note that the `vparm2` and `vparm3` macros will load their argument (`i`) into X1 and X2, respectively. In a similar vein, the code sets up X0, X1, and X2 to print the values held in the `j` and `k` variables ❷ ❸.

This decimal-to-hexadecimal conversion program uses the generic *c.cpp* program from Chapter 1, along with the generic *build* shell script. You can compile and run this program by using the following commands at the command line:

```
$ ./build Listing2-1
$ ./Listing2-1
Calling Listing2-1:
i=1, converted to hex=1
j=123, converted to hex=7b
k=456789, converted to hex=6f855
Listing2-1 terminated
```

As you can see, this program displays the initialized values of `i`, `j`, and `k` in decimal and hexadecimal form.

2.3 Data Organization

In pure mathematics, a value's representation may require an arbitrary number of bits. Computers, on the other hand, generally work with a specific number of bits. Common collections are single bits, groups of 4 bits (called *nibbles*), 8 bits (*bytes*), 16 bits (*half words*, or *hwords*), 32 bits (*words*), 64 bits (*double words*, or *dwords*), 128 bits (*quad words*, or *qwords*), and more. The following subsections describe how the ARM CPU organizes these groups of bits and the typical values you can represent with them.

2.3.1 Bits

The smallest unit of data on a binary computer is a single *bit*. With one bit, you can represent any two distinct items, such as 0 or 1, true or false, and right or wrong. However, you are *not* limited to representing binary data types; you could use a single bit to represent the numbers 723 and 1,245 or, perhaps, the colors red and blue, or even the color red and the number 3,256. You can represent *any* two values with a single bit, but *only* two values with a single bit.

Different bits can represent different things. For example, you could use one bit to represent the values 0 and 1, while a different bit could represent the values true and false, and another bit could represent the two colors red and blue. You can't tell what a bit represents just by looking at it, though.

This illustrates the whole idea behind computer data structures: *data is what you define it to be.* If you use a bit to represent a Boolean (true/false) value, then that bit, by your definition, represents true or false. However, you must be consistent. If you're using a bit to represent true or false at one point in your program, you shouldn't use that value to represent red or blue later.

2.3.2 Nibbles

A *nibble* is a collection of 4 bits. With a nibble, you can represent up to 16 distinct values, using the 16 possible unique combinations of those 4 bits:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

A nibble takes 4 bits to represent a single digit in *binary-coded decimal (BCD)* numbers and hexadecimal numbers. In the case of hexadecimal numbers, each of the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F is represented with 4 bits. BCD uses 4 binary bits to represent each of the 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) used in decimal numbers.

BCD requires 4 bits because you can represent only 8 different values with 3 bits, and representing 10 values takes at least 4 bits. (The additional 6 values you can represent with 4 bits are never used in BCD representation.) In fact, any 16 distinct values can be represented with a nibble, though hexadecimal and BCD digits are the primary items you'll represent with a single nibble.

2.3.3 Bytes

Without question, the most important data structure used by the ARM microprocessor is the *byte*, which consists of 8 bits. Main memory and I/O addresses on the ARM are all byte addresses. This means that the smallest item that can be individually accessed by an ARM program is an 8-bit value. To access anything smaller requires that you read the byte containing the data and eliminate the unwanted bits. The bits in a byte are normally numbered from 0 to 7, as shown in Figure 2-1.

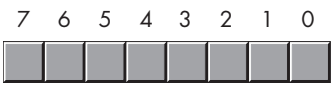


Figure 2-1: Bit numbering

Bit 0 is the *LO bit*, or *least significant bit*, and bit 7 is the *HO bit*, or *most significant bit*, of the byte. I'll refer to any other bit by its number.

A byte contains exactly 2 nibbles, as shown in Figure 2-2.

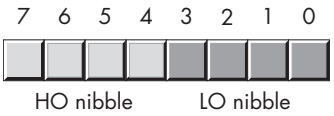


Figure 2-2: The 2 nibbles in a byte

Bits 0 to 3 compose the *LO nibble*, and bits 4 to 7 form the *HO nibble*. Because a byte contains exactly 2 nibbles, byte values require two hexadecimal digits.

Because a byte contains 8 bits, it can represent 2^8 (256) values. Generally, assembly programmers use a byte to represent numeric values in the range 0 through 255, signed numbers in the range –128 through +127 (see section 2.6, "Signed and Unsigned Numbers," on page 65), character codes, and other special data types requiring no more than 256 values. Many data types have fewer than 256 items, so 8 bits is often sufficient.

Because the ARM is a byte-addressable machine, it's more efficient to manipulate a whole byte than an individual bit or nibble. That means it's more efficient to use a whole byte to represent data types that require 2 to 256 items, even if fewer than 8 bits would suffice.

Probably the most important use for a byte is holding a character value. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To communicate with the rest of the world, PCs typically use a variant of the American Standard Code for Information Interchange (ASCII) character set or the Unicode character set. The ASCII character set has 128 defined codes. (Because the Unicode character set has far more than 256 characters, a single byte is insufficient to represent all the Unicode characters; see section 2.17, "Gas Support for the Unicode Character Set," on page 102 for more.)

Bytes are also the smallest variable you can create in a Gas program. To create an arbitrary byte variable, use the `.byte` data type, as follows:

```
        .data
byteVar: .byte   0
```

The `byte` data type holds any 8-bit value: small signed integers, small unsigned integers, characters, and the like. It's up to you to keep track of the type of object you've put into a byte variable.

2.3.4 Half Words

A *half word* is a group of 16 bits. The bits in a half word are numbered from 0 to 15, as Figure 2-3 shows. As with the byte, bit 0 is the LO bit. For half words, bit 15 is the HO bit. When referencing any other bit in a half word, I'll use its bit position number.

Figure 2-3: The bit numbers in a half word

A half word contains exactly 2 bytes, as shown in Figure 2-4. Bits 0 to 7 form the LO byte, and bits 8 to 15 form the HO byte.

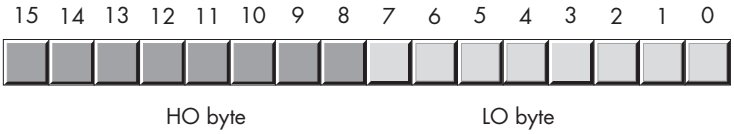


Figure 2-4: The 2 bytes in a half word

A half word also contains 4 nibbles, as shown in Figure 2-5.

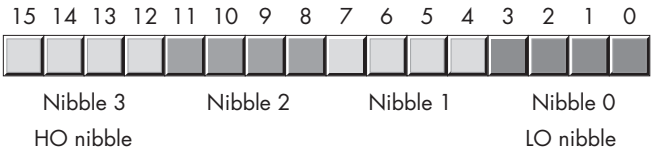


Figure 2-5: The nibbles in a half word

With 16 bits, you can represent 2^{16} (65,536) values. These could be the values in the range 0 to 65,535 or, as is usually the case, the signed values –32,768 to +32,767, or any other data type with no more than 65,536 values.

The two major uses for half words are short signed integer values and short unsigned integer values. Unsigned numeric values are represented by the binary value corresponding to the bits in the half word. Signed numeric values use the two's complement form for numeric values (see section 2.6, "Signed and Unsigned Numbers," on page 65).

As with bytes, you can also create half-word variables in a Gas program. To create an arbitrary half-word variable, just use the `.hword` data type, as follows:

```
        .data
hw:     .hword  0
```

This defines a 16-bit variable (`hw`) initialized with 0.

2.3.5 Words

A word quantity is 32 bits long, as shown in Figure 2-6.

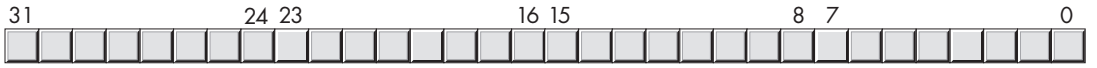


Figure 2-6: The bit numbers in a word

Naturally, this word can be divided into a HO half word and a LO half word, 4 bytes, or 8 nibbles, as shown in Figure 2-7.

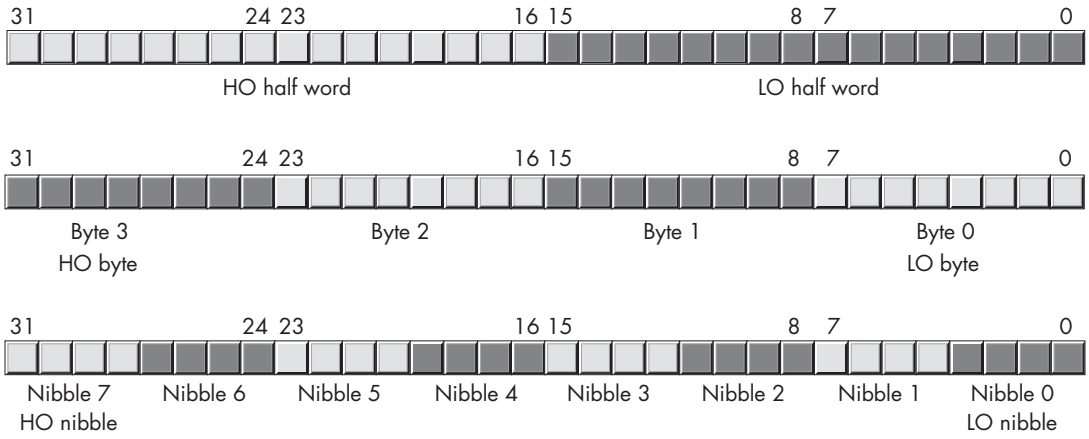


Figure 2-7: The nibbles, bytes, and half words in a word

Words can represent all kinds of things. You'll commonly use them to represent 32-bit integer values (which allow unsigned numbers in the range 0 to 4,294,967,295 or signed numbers in the range –2,147,483,648 to +2,147,483,647); 32-bit floating-point values also fit into a word.

You can create an arbitrary word variable by using the `.word` declaration, as in the following example:

```
      .data
w:    .word   0
```

This defines a 32-bit variable (`w`) initialized with 0.

2.3.6 Double Words and Quad Words

Double-word (64-bit) values are also important because 64-bit integers, pointers, and certain floating-point data types require 64 bits. In a similar vein, *quad-word* (128-bit) values are important because the ARM Neon instruction set can manipulate 128-bit values. The *aoaa.inc* include file includes the `.dword` and `.qword` macros, which allow Gas to declare 64- and 128-bit values by using the `dword` and `qword` types:

```
      .data
dw:   .dword 0
qw:   .qword 0
```

Without *aoaa.inc*, the standard Gas directives are `.quad` (for dwords) and `.octa` (for qwords). This book uses `.dword` and `.qword` because they are more descriptive.

NOTE *Technically, Gas does support* `.dword`*. It's the macOS assembler (Clang assembler) that doesn't support* `.dword` *and requires the macro in the* aoaa.inc *header file.*

You cannot directly manipulate 128-bit integer objects by using standard instructions like `mov`, `add`, and `sub` because the standard ARM integer registers process only 64 bits at a time. In Chapter 8, you'll see how to manipulate these *extended-precision* values; Chapter 11 describes how to directly manipulate qword values by using SIMD instructions.

2.4 Logical Operations on Bits

Although you can represent numeric values with bytes, half words, words, and so on, these are also groups of bits that you can manipulate at the bit level. This section describes the operations on individual bits and how to operate on these bits in larger data structures. You will typically do four logical operations (Boolean functions) on hexadecimal and binary numbers: AND, OR, XOR (exclusive-OR), and NOT.

2.4.1 AND

The AND operation is *dyadic*, meaning it accepts exactly two operands of individual binary bits, as shown here:

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

Many texts call the AND operation a *binary operation*. The term *dyadic* means the same thing and avoids confusion with the binary numbering system.

A *truth table*, which takes the form shown in Table 2-2, is a compact way to represent the AND operation.

Table 2-2: AND Truth Table

AND	0	1
0	0	0
1	0	1

Truth tables work just like the multiplication tables you may have encountered in school. The values in the left column correspond to the left operand of the AND operation. The values in the first row correspond to the right operand of the AND operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result of ANDing those two values together.

In English, the AND operation is, "If the first operand is 1 and the second operand is 1, the result is 1; otherwise, the result is 0." You could also state this as, "If either or both operands are 0, the result is 0."

You can use the AND operation to force a 0 result: if one of the operands is 0, the result is always 0 regardless of the other operand. In Table 2-2, for example, the row labeled with a 0 input contains only 0s, and the column labeled with a 0 contains only 0s. Conversely, if one operand contains a 1, the result is exactly the value of the second operand. These results of the AND operation are important, particularly when you want to force bits to 0. This chapter investigates these uses of the AND operation in section 2.5, "Logical Operations on Binary Numbers and Bit Strings," on the next page.

2.4.2 OR

The OR operation, which is also dyadic, is defined as follows:

```
0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

Table 2-3 shows the truth table for the OR operation.

Table 2-3: OR Truth Table

OR	0	1
0	0	1
1	1	1

Colloquially, the OR operation is, "If the first operand or the second operand (or both) is 1, the result is 1; otherwise, the result is 0." This is also known as the *inclusive-OR* operation.

If one of the operands to the OR operation is a 1, the result is always 1 regardless of the second operand's value. If one operand is 0, the result is always the value of the second operand. As with the AND operation, this is an important side effect of the OR operation that will prove quite useful.

There is a difference between this form of the inclusive-OR operation and the standard English meaning. Consider the sentence "I am going to the store, *or* I am going to the park." Such a statement implies that the speaker is going to the store or to the park, but not to both places. This colloquial use of *or* is analogous not to the inclusive-OR but to the *exclusive-OR* operation.

2.4.3 XOR

The XOR (exclusive-OR) operation is also dyadic. Its definition is as follows:

```
0 xor 0 = 0
0 xor 1 = 1
```

```
1 xor 0 = 1
1 xor 1 = 0
```

Table 2-4 shows the truth table for the XOR operation.

Table 2-4: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English, the XOR operation is, "If the first operand or the second operand, but not both, is 1, the result is 1; otherwise, the result is 0."

If one of the operands to the exclusive-OR operation is a 1, the result is always the *inverse* of the other operand; that is, if one operand is 1, the result is 0 if the other operand is 1, and the result is 1 if the other operand is 0. If the first operand contains a 0, the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

2.4.4 NOT

The NOT operation is *monadic*, meaning it accepts only one operand:

```
not 0 = 1
not 1 = 0
```

Table 2-5 shows the truth table for the NOT operation.

Table 2-5: NOT Truth Table

NOT	0	1
	1	0

The NOT operation inverts the value of the input bit.

2.5 Logical Operations on Binary Numbers and Bit Strings

The previous section defined the logical functions for single-bit operands. Because the ARM uses groups of 8, 16, 32, 64, or more bits, this section extends the definition of these functions to deal with more than 2 bits.

Logical functions on the ARM operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, these functions operate on bit 0 of each value, producing bit 0 of the result; then they operate on bit 1 of the input values, producing bit 1 of the result, and so on. For example, if you want to compute the AND of the following two 8-bit numbers, you would perform the AND operation on each column independently of the others:

```
0b1011_0101
0b1110_1110
-----------
0b1010_0100
```

You may apply this bit-by-bit calculation to the other logical functions as well. To perform a logical operation on two hexadecimal numbers, first convert them to binary.

The ability to force bits to 0 or 1 by using the AND or OR operations and the ability to invert bits by using the XOR operation are very important when working with strings of bits (for example, binary numbers). These operations let you selectively manipulate certain bits within a bit string while leaving other bits unaffected.

For example, if you have an 8-bit binary value *X* and want to guarantee that bits 4 to 7 contain 0s, you could AND the value *X* with the binary value 0000_1111. This bitwise AND operation would force the HO 4 bits to 0 and pass the LO 4 bits of *X* unchanged. Likewise, you could force the LO bit of *X* to 1 and invert bit 2 of *X* by ORing *X* with 0000_0001 and then XORing *X* with 0000_0100.

Using the AND, OR, and XOR operations to manipulate bit strings in this fashion is known as *masking* bit strings, because you can use certain values (1 for AND, 0 for OR/XOR) to mask out or mask in certain bits from the operation when forcing bits to 0, 1, or their inverse. The term *masking* comes from painting. Painters use tape (masking tape) and paper to cover (*mask out*) those portions of an object they want to protect while painting. In a similar sense, programmers use 1s (with the AND operation) in bit positions they want to protect when forcing bits to 0, and they use 0s (with the OR operation) to mask bit positions they want to protect when forcing bits to 1.

The ARM-64 CPUs support five instructions that apply these bitwise logical operations to their operands: `and`, `ands`, `orr`, `eor`, and `mvn`. The `and`, `ands`, `orr`, and `eor` instructions use the same syntax as the `add` and `sub` instructions you learned about in Chapter 1:

```
and   dest, source_left, source_right
ands  dest, source_left, source_right  // Affects the flags
orr   dest, source_left, source_right
eor   dest, source_left, source_right  // XOR operation
```

These operands have the same limitations as the `add` operands. Specifically, the $source_{left}$ operand has to be a register operand, the $source_{right}$ operand must be a register or a constant, and the *dest* operand must be a register. The operands must also be the same size. You'll see extensions to this syntax in section 2.19, "Operand2," on page 106.

The `orr` and `eor` instructions do not have versions with the `s` suffix. You'll have to work around this bizarre limitation in the instruction set if you would like to test the flags after these instructions.

The immediate constant (*source*$_{right}$ operand) has a completely different set of restrictions than the immediate constants for `add` and `sub`. For more information on what constitutes legal immediate constants, see section 2.19, "Operand2," on page 106.

These instructions compute the obvious bitwise logical operation via the following equation:

```
dest = source_left operator source_right
```

The ARM doesn't have an actual `not` instruction. Instead, a variant of the `mov` instruction does the honors: `mvn` (move and not). This instruction takes the following form:

```
mvn  dest, source
```

Note that this instruction does not provide a form with an `s` suffix that updates the condition code flags after its execution.

This instruction computes the following result:

```
dest = not( source )
```

The operands must both be registers.

The program in Listing 2-2 inputs two hexadecimal values from the user and calculates their logical AND, OR, XOR, and NOT.

```
// Listing2-2.S
//
// Demonstrate AND, OR, XOR, and NOT operations.

#include "aoaa.inc"

            .data
leftOp:     .dword    0xf0f0f0f
rightOp1:   .dword    0xf0f0f0f0
rightOp2:   .dword    0x12345678
result:     .dword    0
saveLR:     .dword    0

titleStr:   .asciz   "Listing 2-2"

fmtStr1:    .asciz   "%lx AND %lx = %lx\n"
fmtStr2:    .asciz   "%lx OR  %lx = %lx\n"
fmtStr3:    .asciz   "%lx XOR %lx = %lx\n"
fmtStr4:    .asciz   "NOT %lx = %lx\n"

            .text
            .align    2   // Make code word-aligned.

            .extern   printf

// Return program title to C++ program:
```

```
            .global     getTitle
getTitle:

// Load address of "titleStr" into the X0 register (X0 holds the
// function return result) and return back to the caller:

            lea     x0, titleStr
            ret

// Here is the "asmMain" function.

            .global asmMain
asmMain:

// "Magic" instruction offered without explanation at this point:

            sub     sp, sp, 64

// Save LR so we can return to C++ code:

            lea     x0, saveLR
            str     lr, [x0]

// Demonstrate the AND operation:

          ❶ lea     x0, leftOp
            ldr     x1, [x0]
            lea     x0, rightOp1
            ldr     x2, [x0]
            and     x3, x1, x2  // Compute left AND right.
            lea     x0, result
            str     x3, [x0]

            lea     x0, fmtStr1 // Print result.
            vparm2  leftOp
            vparm3  rightOp1
            vparm4  result
            bl      printf

// Demonstrate the OR operation:

          ❷ lea     x0, leftOp
            ldr     x1, [x0]
            lea     x0, rightOp1
            ldr     x2, [x0]
            orr     x3, x1, x2  // Compute left OR right.
            lea     x0, result
            str     x3, [x0]

            lea     x0, fmtStr2 // Print result.
            vparm2  leftOp
            vparm3  rightOp1
            vparm4  result
            bl      printf
```

```
// Demonstrate the XOR operation:

        ❸ lea     x0, leftOp
          ldr     x1, [x0]
          lea     x0, rightOp1
          ldr     x2, [x0]
          eor     x3, x1, x2  // Compute left XOR right.
          lea     x0, result
          str     x3, [x0]

          lea     x0, fmtStr3 // Print result.
          vparm2  leftOp
          vparm3  rightOp1
          vparm4  result
          bl      printf

// Demonstrate the NOT instruction:

        ❹ lea     x0, leftOp
          ldr     x1, [x0]
          mvn     w1, w1      // W1 = not W1 (32 bits)
          lea     x0, result
          str     x1, [x0]

          lea     x0, fmtStr4 // Print result.
          vparm2  leftOp
          vparm3  result
          bl      printf

        ❺ lea     x0, rightOp1
          ldr     x1, [x0]
          mvn     w1, w1      // W1 = not W1 (32 bits)
          lea     x0, result
          str     x1, [x0]

          lea     x0, fmtStr4 // Print result.
          vparm2  rightOp1
          vparm3  result
          bl      printf

        ❻ lea     x0, rightOp2
          ldr     x1, [x0]
          mvn     w1, w1      // W1 = not W1
          lea     x0, result
          str     x1, [x0]

          lea     x0, fmtStr4 // Print result.
          vparm2  rightOp2
          vparm3  result
          bl      printf

// Another "magic" instruction that undoes the effect of
// the previous one before this procedure returns to its
// caller:
```

```
            add     sp, sp, #64

// Restore LR so we can return to C++ code:

            lea     x0, saveLR
            ldr     lr, [x0]
            ret     // Returns to caller
```

The code computes the logical AND ❶, OR ❷, and XOR ❸ of `leftOp` and `rightOp1`. It then prints the result. The code next computes the NOT of `leftOp` ❹, `rightOp1` ❺, and `rightOp2` ❻ and prints their results.

Here's the `build` command and output for the program in Listing 2-2:

```
$ ./build Listing2-2
$ ./Listing2-2
Calling Listing2-2:
f0f0f0f AND f0f0f0f0 = 0
f0f0f0f OR  f0f0f0f0 = ffffffff
f0f0f0f XOR f0f0f0f0 = ffffffff
NOT f0f0f0f = f0f0f0f0
NOT f0f0f0f0 = f0f0f0f
NOT 12345678 = edcba987
Listing2-2 terminated
```

As you can see, the AND operation clears bits, the OR operation sets bits, and the XOR and NOT operations invert bits.

2.6 Signed and Unsigned Numbers

Thus far, this chapter has treated binary numbers as unsigned values. The binary number 0 . . . 00000 represents 0, 0 . . . 00001 represents 1, 0 . . . 00010 represents 2, and so on toward infinity. With n bits, you can represent 2^n unsigned numbers.

What about negative numbers? If you assign half of the possible combinations to the negative values, and half to the positive values and 0, with n bits you can represent the signed values in the range -2^{n-1} to $+2^{n-1} - 1$. This means you can represent the negative values –128 to –1 and the nonnegative values 0 to 127 with a single 8-bit byte. With a 16-bit half word, you can represent values in the range –32,768 to +32,767. With a 32-bit word, you can represent values in the range –2,147,483,648 to +2,147,483,647.

In mathematics and computer science, the *complement method* encodes negative and nonnegative (positive plus zero) numbers into two equal sets in such a way that they can use the same algorithm or hardware to perform addition and produce the correct result regardless of the sign.

The ARM microprocessor uses *two's complement* notation to represent signed integers. In this system, the HO bit of a number is a *sign bit*: the integers are divided into two equal sets. If the sign bit is 0, the number is positive (or zero); if the sign bit is 1, the number is negative (taking a complement form, which I'll describe in a moment).

Here are some examples of 16-bit positive and negative numbers:

0x8000 is negative because the HO bit is 1.

0x100 is positive because the HO bit is 0.

0x7FFF is positive.

0xFFFF is negative.

0xFFF is positive.

If the HO bit is 0, the number is positive (or zero) and uses the standard binary format. If the HO bit is 1, the number is negative and uses the two's complement form: the magic form that supports addition of negative and nonnegative numbers with no special hardware.

You convert a positive number to its negative two's complement form with the following algorithm steps:

1. Invert all the bits in the number; that is, apply the NOT function.
2. Add 1 to the inverted result and ignore any carry out of the HO bit.

This produces a bit pattern that satisfies the mathematical definition of the complement form. In particular, adding negative and nonnegative numbers using this form produces the expected result.

For example, to compute the 8-bit equivalent of –5:

1. Write 5 in binary: 0000_0101.
2. Invert all the bits: 1111_1010.
3. Add 1 to obtain the result: 1111_1011.

If you take –5 and perform the two's complement operation on it, you get your original value, 0000_0101, back again:

1. Take the two's complement for –5: 1111_1011.
2. Invert all the bits: 0000_0100.
3. Add 1 to obtain the result 0000_0101.

If you add +5 and –5 together (ignoring any carry out of the HO bit), you get the expected result of 0:

```
     0b1111_1011     Take the two's complement for -5.
   + 0b0000_0101     Invert all the bits and add 1.
     -----------
(1)  0b0000_0000     Sum is zero, if you ignore carry.
```

The following examples provide some positive and negative 16-bit signed values:

0x7FFF: +32,767, the largest 16-bit positive number

0x4000: +16,384

0x8000: –32,768, the smallest 16-bit negative number

To convert the preceding numbers to their negative counterpart (that is, to negate them), do the following:

```
0x7FFF: 0b0111_1111_1111_1111   +32,767
        0b1000_0000_0000_0000   Invert all the bits (8000h).
        0b1000_0000_0000_0001   Add 1 (8001h or -32,767).

x04000: 0b0100_0000_0000_0000   16,384
        0b1011_1111_1111_1111   Invert all the bits (0BFFFh).
        0b1100_0000_0000_0000   Add 1 (0C000h or -16,384).

0x8000: 0b1000_0000_0000_0000   -32,768
        0b0111_1111_1111_1111   Invert all the bits (7FFFh).
        0b1000_0000_0000_0000   Add 1 (8000h or -32,768).
```

0x8000 inverted becomes 0x7FFF. After adding 1, you obtain 0x8000! Wait, what's going on here? –(–32,768) is –32,768? Of course not. But the value +32,768 cannot be represented with a 16-bit signed number, so you cannot negate the smallest negative value.

Usually, you won't need to perform the two's complement operation by hand. The ARM microprocessor provides an instruction, `neg` (negate), that performs this operation for you:

```
neg  dest, source
negs dest, source  // Sets condition code flags
```

This instruction computes *dest* = -*source*, and the operands must be registers. Because this is a signed integer operation, it only makes sense to operate on signed integer values. Listing 2-3 demonstrates the two's complement operation and the `neg` instruction on signed 32-bit integer values.

```
// Listing2-3.S
//
// Demonstrates two's complement operation and input of
// numeric values

#include "aoaa.inc"

            .equ        maxLen, 256

            .data
titleStr:   .asciz      "Listing 2-3"

prompt1:    .asciz      "Enter an integer between 0 and 127:"
fmtStr1:    .asciz      "Value in hexadecimal: %x\n"
fmtStr2:    .asciz      "Invert all the bits (hexadecimal): %x\n"
fmtStr3:    .asciz      "Add 1 (hexadecimal): %x\n"
fmtStr4:    .asciz      "Output as signed integer: %d\n"
fmtStr5:    .ascii      "Negate again and output as signed integer:"
            .asciz      " %d\n"

fmtStr6:    .asciz      "Using neg instruction: %d\n"
```

```
intValue:   .dword      0
saveLR:     .dword      0


// The following reserves 256 bytes of storage to hold a string
// read from the user.

❶ input:      .space      maxLen, 0

            .text
            .align      2
            .extern     printf
            .extern     atoi
          ❷ .extern     readLine

// Return program title to C++ program:

            .global     getTitle
getTitle:
            lea         x0, titleStr
            ret

// Here is the asmMain function:

            .global     asmMain
asmMain:

// "Magic" instruction offered without explanation at this point:

            sub     sp, sp, #128

// Save LR so we can return to C++ program:

            lea     x0, saveLR
            str     lr, [x0]

// Read an unsigned integer from the user: this code will blindly
// assume that the user's input was correct. The atoi function
// returns zero if there was some sort of error on the user
// input. Later chapters in AoAA will describe how to check for
// errors from the user.

            lea     x0, prompt1
            bl      printf

            lea     x0, input
            mov     x1, #maxLen
            bl      readLine

// Call C stdlib strtol function:
//
// i = strtol( str, NULL, 10 )
```

```
❸ lea     x0, input
  mov     x1, xzr
  mov     x2, #10
  bl      strtol
  lea     x1, intValue
  str     x0, [x1]

// Print the input value (in decimal) as a hexadecimal number:

  lea     x0, fmtStr1
  vparm2  intValue
  bl      printf

// Perform the two's complement operation on the input number.
// Begin by inverting all the bits:

  lea     x1, intValue
  ldr     x0, [x1]
  mvn     x0, x0      // Not X0
  str     x0, [x1]    // Store back into intValue.
  lea     x0, fmtStr2
  vparm2  intValue
  bl      printf

// Invert all the bits and add 1 (inverted value is in intValue):

  lea     x0, intValue
  ldr     x1, [x0]
  add     x1, x1, #1
  str     x1, [x0]    // Store back into intValue.
  lea     x0, fmtStr3
  vparm2  intValue
  bl      printf

  lea     x0, fmtStr4 // Output as integer rather
  vparm2  intValue    // than hexadecimal.
  bl      printf

// Negate the value and print as a signed integer. Note that
// intValue already contains the negated value, so this code
// will print the original value:

  lea     x0, intValue
  ldr     x1, [x0]
  mvn     x1, x1
  add     x1, x1, #1
  str     x1, [x0]
  lea     x0, fmtStr5
  vparm2  intValue
  bl      printf

// Negate the value using the neg instruction:

  lea     x0, intValue
  ldr     x1, [x0]
```

```
            neg     x1, x1
            str     x1, [x0]
            lea     x0, fmtStr6
            vparm2  intValue
            bl      printf

// Another "magic" instruction that undoes the effect of the
// previous one before this procedure returns to its caller:

            lea     x0, saveLR
            ldr     lr, [x0]
            add     sp, sp, #128
            ret     // Returns to caller
```

The `.space` directive ❶ is new in this chapter. This directive reserves a buffer (array of bytes). The first operand specifies the number of bytes to reserve, and the second operand specifies the value to assign to each byte in the buffer. This particular directive sets aside 256 bytes to hold a line of text to be input by the user. We'll discuss arrays and memory allocation for arrays further in Chapter 4.

The `readLine` function ❷ is supplied by the C++ code in the *c.cpp* source file. This function expects two parameters: the address of a buffer in the X0 register and a maximum input count in the X1 register (including room for a zero-terminating byte). When called, this function will read a line of text from the standard input device and place those characters in the specified buffer (zero-terminating, and truncating if the input is greater than the value passed in X1).

The `strtol` function ❸ is a C stdlib function that will convert a string of characters, presumably containing numeric digits, into long integer form (64 bits). This function expects three arguments: X0 contains the address of a buffer (containing the string to convert); X1 points at the end of the numeric string, or is ignored if it contains NULL (0); and X2 contains the radix (base) for the conversion. The function returns the converted value in the X0 register.

Here's the `build` command and program output for Listing 2-3 (I supplied `123` as the input for this particular run of the program):

```
$ ./build Listing2-3
$ ./Listing2-3
Calling Listing2-3:
Enter an integer between 0 and 127:123
Value in hexadecimal: 7b
Invert all the bits (hexadecimal): ffffff84
Add 1 (hexadecimal): ffffff85
Output as signed integer: -123
Negate again and output as signed integer: 123
Using neg instruction: -123
Listing2-3 terminated
```

As you can see, this program reads an integer value in decimal format from the user, inverts the bits, adds 1 (the two's complement operation), and then displays the result.

2.7 Sign Extension and Zero Extension

Converting a small two's complement value to a larger number of bits can be accomplished via *sign extension* operations.

To extend a signed value from a certain number of bits to a greater number of bits, copy the sign bit into all the additional bits in the new format. For example, to sign-extend an 8-bit number to a 16-bit number, copy bit 7 of the 8-bit number into bits 8 to 15 of the 16-bit number. To sign-extend a 16-bit half word to a word, copy bit 15 into bits 16 to 31 of the word. Likewise, to sign-extend a 32-bit word into a 64-bit double word, copy bit 31 from the word through the upper 32 bits of the double word.

You must use sign extension when manipulating signed values of varying lengths. For example, to add a signed byte quantity to a word quantity, you must sign-extend the byte quantity to a word before adding the two values. Other operations (multiplication and division, in particular) may require a sign extension to 32 bits. Table 2-6 provides several examples of sign extension.

Table 2-6: Examples of Sign Extension

8 bits	16 bits	32 bits
0x80	0xFF80	0xFFFFFF80
0x28	0x0028	0x00000028
0x9A	0xFF9A	0xFFFFFF9A
0x7F	0x007F	0x0000007F
—	0x1020	0x00001020
—	0x8086	0xFFFF8086

To extend an unsigned value to a larger one, you must *zero-extend* the value. Zero extension is easy—just store a zero into the HO byte(s) of the larger operand. For example, to zero-extend the 8-bit value 0x82 to 16 bits, prepend a zero to the HO byte, yielding 0x0082. Table 2-7 provides several zero-extension examples.

Table 2-7: Examples of Zero Extension

8 bits	16 bits	32 bits
0x80	0x0080	0x00000080
0x28	0x0028	0x00000028
0x9A	0x009A	0x0000009A
0x7F	0x007F	0x0000007F
—	0x1020	0x00001020
—	0x8086	0x00008086

You can zero-extend to double or quad words by using this same approach.

2.8 Sign Contraction and Saturation

Sign contraction, converting a value with a certain number of bits to the identical value with a fewer number of bits, is a little more difficult. You cannot always convert a given *n*-bit number to an *m*-bit number if $m < n$. For example, consider the value –448. As a 16-bit signed number, its hexadecimal representation is 0xFE40. The magnitude of this number is too large for an 8-bit value, so you cannot sign-contract it to 8 bits; doing so would create an overflow condition.

To properly sign-contract a value, the HO bits to discard must all contain either 0 or 1, and the HO bit of your resulting value must match *every* bit you've removed from the number. Here are some examples (16 bits to 8 bits):

0xFF80 can be sign-contracted to 0x80.

0x0040 can be sign-contracted to 0x40.

0xFE40 cannot be sign-contracted to 8 bits.

0x0100 cannot be sign-contracted to 8 bits.

If you must convert a larger object to a smaller object, and you're willing to live with loss of precision, you can use *saturation*. To convert a value via saturation, copy the larger value to the smaller value if it is not outside the range of the smaller object. If the larger value is outside the range of the smaller value, *clip* the value by setting it to the largest (or smallest) value within the range of the smaller object.

For example, when converting a 16-bit signed integer to an 8-bit signed integer, if the 16-bit value is in the range –128 to +127, you copy the LO byte of the 16-bit object to the 8-bit object. If the 16-bit signed value is greater than +127, you clip the value to +127 and store +127 into the 8-bit object. Likewise, if the value is less than –128, you clip the final 8-bit object to –128.

Although clipping the value to the limits of the smaller object results in loss of precision, this is sometimes acceptable because the alternative is to raise an exception or otherwise reject the calculation. For many applications, such as audio or video processing, the clipped result is still recognizable, so the conversion is a reasonable choice.

2.9 Loading and Storing Byte and Half-Word Values

Memory on the ARM is byte-addressable. Up to this point, however, all loads and stores in this book have been either word or dword operations (determined by the `ldr/str` register's size). Fear not: the ARM CPU provides instructions for loading and storing bytes, half words, words, double words, and even quad words.

The generic `ldr` instruction takes the following forms:

```
ldr    reg, mem
ldrb   reg32, mem
ldrsb  reg, mem
ldrh   reg32, mem
ldrsh  reg, mem
ldrsw  reg64, mem
```

The reg_{32} operands can be only 32-bit registers, and the reg_{64} operand can be only a 64-bit register. The *reg* (no subscript) operands can be 32- or 64-bit registers.

The `ldrb` and `ldrsb` instructions load a byte from memory into the destination register. Since the register is always 32 or 64 bits wide, the byte from memory must be extended in some fashion when it is loaded into the register. The `ldrb` instruction zero-extends the byte from memory into the register. The `ldrsb` instruction sign-extends the memory byte into the register. Zero extension works only with 32-bit registers, but the `ldrb` and `ldrh` instructions will automatically zero out the HO 32 bits of the corresponding 64-bit register. If you sign-extend a byte or half word into a 32-bit register, this will zero out the HO 32 bits of the corresponding 64-bit register. Specify a 64-bit register if you want to sign-extend the byte or half word throughout the 64-bit register.

The `ldrh` and `ldrsh` instructions similarly load and extend a half-word value from memory by using zero extension (`ldrh`) and sign extension (`ldrsh`). As before, the `ldrh` instruction accepts a 32-bit register, but it will automatically zero-extend throughout the full 64 bits of the register.

The `ldrsw` instruction will fetch a 32-bit signed integer from memory and sign-extend it into the 64-bit register specified as the destination. No explicit instruction zero-extends from 32 to 64 bits; the standard `ldr` instruction, with a 32-bit register operand, will automatically do this.

Note that *mem* operands consisting only of a label (PC-relative addressing) are valid only for the `ldr` and `ldrsw` instructions. The other instructions allow only register-based addressing modes (for example, `[X0]`).

The `ldr{`*size*`}` instructions work well for loading and extending byte, half-word, and word values from memory. If the value to extend is sitting in another register, you don't want to have to store that register in memory, so you can extend the value into a different register. Fortunately, the ARM provides a set of instructions, `sxtb`, `sxth`, and `sxtw`, specifically for this situation:

```
sxtb regdest, regsrc  // Sign-extends LO byte of regsrc
sxth regdest, regsrc  // Sign-extends LO half word of regsrc
sxtw regdest, regsrc  // Sign-extends LO word of regsrc
```

The `sxtw` instruction requires a 64-bit destination register. The `sxtb`, `sxth`, and `sxtw` instructions require 32-bit source registers, regardless of the destination register size.

The ARM does not provide any explicit instructions for zero-extending one register into another. However, you can use some tricks to achieve the same result. Whenever you move data from one register into a 32-bit register, the ARM automatically zeros out the HO 32 bits of the corresponding destination 64-bit register. You can use this behavior to zero-extend any smaller value to a larger value.

The following instruction copies W*m* into W*n* and clears the HO 32 bits of X*n* in the process:

```
mov  wn, wm  // Zero-extends 32-bit Wm into Xn
```

The following instruction ANDs the value in W*m* with 0xFFFF and then stores the result into W*n*, zero-extending throughout the HO bits of X*n*:

```
and  wn, wm, #0xFFFF  // Zero-extends 16 bits to 64
```

And, finally, the following instruction zero-extends the LO byte of W*m* through X*n*:

```
and  wn, wm, #0xFF  // Zero-extends 8 bits to 64
```

Storing bytes and half words to memory is much simpler than loading. The ARM doesn't support contraction or saturation while storing to memory. Therefore, the byte and half-word store instructions take the following two forms:

```
strb reg32, mem
strh reg32, mem
```

The `strb` instruction stores the LO byte of the specified register to memory. The `strh` instruction stores the LO half word of the register to memory. The register must be a 32-bit register (if you want to store the LO byte or half word of a 64-bit register, simply specify the 32-bit register instead; this does the same thing). Note that *mem* must be a register-based addressing mode (these instructions do not allow the PC-relative addressing mode).

2.10 Control-Transfer Instructions

The assembly language examples thus far have limped along without making use of *conditional execution*, or the ability to make decisions while executing code. Indeed, except for the `bl` and `ret` instructions, I haven't covered any ways to affect the straight-line execution of assembly code. However, to provide meaningful examples for the remainder of this book, you'll soon need the ability to conditionally execute sections of code. Taking a brief detour from load and store instructions, this section provides a brief introduction to the subject of conditional execution and transferring control to other sections of your program.

2.10.1 Branch

Perhaps the best place to start is with a discussion of the ARM unconditional control-transfer instruction: the `b` instruction. The `b` instruction takes the form

```
b statementLabel
```

where *`statementLabel`* is an identifier attached to a machine instruction in your `.text` section. The `b` instruction immediately transfers control to the statement prefaced by the label. This is semantically equivalent to a `goto` statement in an HLL.

Here is an example of a statement label in front of a `mov` instruction:

```
stmtLbl: mov x0, #55
```

Like all Gas symbols, statement labels have an address associated with them: the memory address of the machine instruction following the label.

Statement labels don't have to be on the same physical source line as a machine instruction. Consider the following example:

```
anotherLabel:
   mov x0, #55
```

This example is semantically equivalent to the previous one. The value (address) bound to `anotherLabel` is the address of the machine instruction following the label. In this case, it's still the `mov` instruction, even though that `mov` instruction appears on the next line (it still follows the label without any other Gas statements that would generate code occurring between the label and the `mov` statement).

B IS FOR BAD

The letter `b` is an incredibly bad choice for an instruction mnemonic. The fact that it looks like a single-letter variable name, most of the time, makes reading both code and this book more difficult. For a while, I considered creating the following CPP macro to allow me to use `bra` rather than `b` in this book:

```
#define bra b
```

Ultimately, my concern that some people would prefer to use the "official" mnemonic prevented me from doing this. However, you can use this trick on your own to write more readable code.

The ARM supports a special version of the conditional branch: `b.al` (branch always). This instruction is an unconditional branch to the target location. The main drawback to using `b.al` is that it is limited to a ±1MB range (like the other

(continued)

conditional branches), while the b instruction supports a ±128MB range. However, the ±1MB range is sufficient for most cases. This book favors using the b.al mnemonic because it is more readable. If you prefer, feel free to substitute b.al for b in your own code (or, better yet, use the bra macro I described).

Technically, you could also jump to a procedure label instead of a statement label. However, the b instruction does not set up a return address; if the procedure executes a ret instruction, the return location may be undefined. Chapter 5 explores return addresses in greater detail.

Because b is a poor name for an instruction mnemonic (as we just discussed in "B Is for Bad"), this book will use the b.al instruction when branching to code within the current source file and reserve b for those rare instances when branching to code outside a ±1MB range.

2.10.2 Instructions That Affect the Condition Code Flags

When presenting the add, sub, and, orr, eor, and neg instructions, I pointed out that they typically take two forms:

```
instr   operands
instrs  operands  // Only adds, subs, ands, and negs
```

The form with the s suffix (adds, for example) will update the condition code flags in the PSTATE register after the instruction completes. For example, the adds and subs instructions will do the following:

- Set the carry flag if an unsigned overflow occurs during the arithmetic operation and clear it otherwise.
- Set the overflow flag if a signed overflow occurs.
- Set the zero flag if the operation produces a zero result.
- Set the negative (sign) flag if the operation produces a negative result (HO bit is set).

While not all instructions support the s suffix, many that perform some sort of calculation will allow this suffix. By allowing you to select which instructions affect the flags, the ARM CPU allows you to preserve the condition codes across the execution of some instructions whose effect on the flags you want to ignore.

As their name suggests, these condition codes allow you to test for certain conditions and conditionally execute code based on those tests. The next section describes how you can test the condition code flags and make decisions based on their settings.

2.10.3 Conditional Branch

Although the `b.al/b` instruction is indispensable in assembly language programs, it doesn't provide any ability to conditionally execute sections of code—hence the name *unconditional branch*. Fortunately, the ARM CPUs provide a wide array of *conditional branch instructions* that allow conditional execution of code.

These instructions test the condition code bits in the PSTATE register to determine whether a branch should be taken. There are four condition code bits in the PSTATE register that these conditional jump instructions test: the carry, sign, overflow, and zero flags.

The ARM CPUs provide eight instructions that test each of these four flags, as shown in Table 2-8. The basic operation of the conditional jump instructions is to test a flag to see whether it is set (1) or clear (0) and branch to a target label if the test succeeds. If the test fails, the program continues execution with the next instruction following the conditional jump instruction.

Table 2-8: Conditional Branch Instructions That Test the Condition Code Flags

Instruction	Explanation
`bcs` *`label`*	Branch if carry is set. Jump to *`label`* if the carry flag is set (1); control falls through to the next instruction if the carry is clear (0).
`bcc` *`label`*	Branch if carry is clear. Jump to *`label`* if the carry flag is clear (0); fall through if the carry is set (1).
`bvs` *`label`*	Branch if overflow set. Jump to *`label`* if the overflow flag is set (1); fall through if the overflow is clear (0).
`bvc` *`label`*	Branch if overflow clear. Jump to *`label`* if the overflow flag is clear (0); fall through if the overflow is set (1).
`bmi` *`label`*	Branch if minus. Jump to *`label`* if the negative (sign) flag is set (1); fall through if the sign is clear (0).
`bpl` *`label`*	Branch if positive (or zero). Jump to *`label`* if the negative flag is clear (0); fall through if the sign is set (1).
`beq` *`label`*	Branch if equal. Jump to *`label`* if the zero flag is set (1); fall through if zero is clear (0).
`bne` *`label`*	Branch if not equal. Jump to *`label`* if the zero flag is clear (0); fall through if zero is set (1).

For historical reasons, Gas also allows conditional branch mnemonics of the form `b.`*`condition`* (for example, `b.cs`, `b.cc`, `b.vs`, and `b.vc`). This form is based on the 32-bit ARM instruction set that allowed conditional execution of most data-processing instructions by using a "dot condition" suffix. While the 64-bit ARM instruction set no longer supports these conditional instructions, it does allow the dot condition syntax for the branch instruction. Since it's easier to type the conditional branches without the period, most people use that form when writing 64-bit ARM assembly language with Gas. Gas under Linux does not seem to support `bal` but does support

b.al, and the macOS assembler seems to support b.al just fine. That's why this book uses b.al for unconditional branches.

To use a conditional branch instruction, you must first execute an instruction that affects one or more of the condition code flags. For example, an unsigned arithmetic overflow will set the carry flag; if overflow does not occur, the carry flag will be clear. Therefore, you could use the bcs and bcc instructions after an adds instruction to see whether an unsigned overflow occurred during the calculation. For example, the following code checks for unsigned overflow by using bcs:

```
    lea  x0, int32Var
    ldr  w0, [x0]
    lea  x1, anotherVar
    ldr  w1, [x1]
    adds w0, w0, w1
    bcs  overflowOccured

// Continue down here if the addition did not
// produce an overflow.

    .
    .
    .

overflowOccured:

// Execute this code if the sum of int32Var and anotherVar
// does not fit into 32 bits.
```

As noted earlier, adds (and subs/negs) sets the condition codes based on signed/unsigned overflow, a zero result, or a negative result. The ands instruction copies the HO bit of its result into the negative flag and sets/clears the zero flag if it produces a zero/nonzero result.

2.10.4 cmp and Corresponding Conditional Branches

The ARM cmp instruction is extremely useful in conjunction with the conditional branches. The syntax for cmp is

```
cmp left, right
```

where *left* is a register (32 or 64 bits) and *right* is either a register or a small immediate constant. The instruction compares the *left* operand to the *right* operand and sets the flags based on the comparison. You can then use the conditional branch instructions to transfer control based on the result of the comparison.

Although cmp does not have an s suffix, it will set the condition code flags; indeed, that's why cmp exists. Technically, cmp isn't a real instruction, but rather an alias (synonym) for the subs instruction with a destination operand of WZR or XZR.

PSEUDO-INSTRUCTIONS (ALIASES) AND BUILT-IN MACROS

You'll often discover that two ARM64 assembly language instructions do exactly the same operation. Consider, for example, the following two instructions:

```
cmp x0, x1
subs xzr, x0, x1
```

The first instruction compares X0 to X1 and sets the condition code flags. The second instruction subtracts X1 from X0, sets the condition code flags, and then throws the result away (whenever you store a value into the zero register—WZR or XZR—the result is lost). The comparison operation is exactly equivalent to subtraction if you don't keep the difference (which is exactly what the `subs` instruction is doing), meaning these two instructions do exactly the same thing.

The ARM's designers noticed this semantic equivalence between many of their instructions and decided, "This is a RISC machine; we should not include extra hardware to handle redundant instructions." As they already had the `subs` instruction, they basically threw out the `cmp` instruction.

You may be thinking, "Didn't you discuss the `cmp` instruction already?" Yes, I did. But I lied: this isn't actually a `cmp` machine instruction. When the assembler accepts and executes a mnemonic named `cmp` that does everything you'd expect a `cmp` instruction to do, under the covers Gas has actually translated that `cmp` instruction into a `subs` instruction.

The `cmp` instruction is an example of a *pseudo-instruction*, a macro built into Gas (and most other ARM assemblers) that automatically translates `cmp` into the corresponding `subs` instruction. In fact, a fair number of ARM instructions fall into this same category.

In this book, we won't worry about whether an instruction is real or pseudo. The semantics are the important aspect, not the particular assembly language syntax. If the assembler contains (standardized) built-in macros to help you write clearer code, so much the better. This section exists just to let you know what is going on if you read about pseudo-instructions elsewhere or if you look at a disassembled listing of your code and the instructions have changed from what you actually wrote.

After executing a compare instruction, you might ask these reasonable questions:

- Is the *`leftOperand`* equal to the *`rightOperand`*?
- Is the *`leftOperand`* not equal to the *`rightOperand`*?
- Is the *`leftOperand`* less than the *`rightOperand`*?
- Is the *`leftOperand`* less than or equal to the *`rightOperand`*?

- Is the *leftOperand* greater than the *rightOperand*?
- Is the *leftOperand* greater than or equal to the *rightOperand*?

For less-than and greater-than comparisons, you might also ask, "Are these signed or unsigned comparisons?"

The ARM provides conditional branches to use after executing `cmp` that answer these questions. Table 2-9 lists these instructions for unsigned comparisons.

Table 2-9: Unsigned Conditional Branches

Instruction	Flag(s) tested	Description
`beq`	Z = 1	Branch if equal; fall through if not equal. After a comparison, this branch will be taken if the first `cmp` operand is equal to the second operand.
`bne`	Z = 0	Branch if not equal; fall through if equal. After a comparison, this branch will be taken if the first `cmp` operand is not equal to the second operand.
`bhi`	C = 1 and Z = 0	Branch if higher; fall through if not higher. After a comparison, this branch will be taken if the first `cmp` operand is greater than the second operand.
`bhs`	C = 1	Branch if higher or same; fall through if not higher or same. After a comparison, this branch will be taken if the first `cmp` operand is greater than or equal the second operand.
`blo`	C = 0	Branch if lower; fall through if not lower. After a comparison, this branch will be taken if the first `cmp` operand is less than the second operand.
`bls`	C = 0 or Z = 1	Branch if lower or same; fall through if not lower or same. After a comparison, this branch will be taken if the first `cmp` operand is less than or equal to the second operand.

If the left and right operands contain signed integer values, use the signed branches in Table 2-10.

Table 2-10: Signed Conditional Branches

Instruction	Flag(s) tested	Description
`beq`	Z = 1	Branch if equal; fall through if not equal. After a comparison, this branch will be taken if the first `cmp` operand is equal to the second operand.
`bne`	Z = 0	Branch if not equal; fall through if equal. After a comparison, this branch will be taken if the first `cmp` operand is not equal to the second operand.
`bgt`	Z = 0 and N = V	Branch if greater than; fall through if less than or equal. After a comparison, this branch will be taken if the first `cmp` operand is greater than the second operand.
`bge`	N = V	Branch if greater than or equal; fall through if less than. After a comparison, this branch will be taken if the first `cmp` operand is greater than or equal to the second operand.
`blt`	N ≠ V	Branch if less than; fall through if greater than or equal. After a comparison, this branch will be taken if the first `cmp` operand is less than the second operand.
`ble`	N ≠ V or Z = 1	Branch if less than or equal; fall through if greater than. After a comparison, this branch will be taken if the first `cmp` operand is less than or equal to the second operand.

As for the earlier branches based on condition codes, Gas allows branches of the form `b.condition` in addition to the forms in Tables 2-9 and 2-10. As it turns out, as shown in the "Flag(s) tested" columns, the `bcs` and `bhs` instructions are synonyms, as are the `bcc` and `blo` instructions.

Importantly, the `cmp` instruction sets the flags only for integer comparisons, which will also cover characters and other types you can encode with an integer value. Specifically, the instruction does not compare floating-point values and set the flags as appropriate for a floating-point comparison.

Sometimes it's convenient to branch on an opposite condition. For example, you might have the following logic:

```
    cmp x0, x1
    // Branch to geLbl if X0 is not less than X1.

    // Fall through to this code if X0 < X1.
     .
     .
     .
// Branch here if NOT(X0 < X1) (that is, X0 >= X1).
geLbl:
```

Of course, the opposite of *less than* is *greater than or equal*, so this pseudocode could be written as follows:

```
    cmp x0, x1
    bge geLbl

    // Fall through to this code if X0 < X1.
     .
     .
     .
// Branch here if NOT(X0 < X1) (that is, X0 >= X1).
geLbl:
```

However, using opposite branches to skip around the code you want to execute on a condition (such as less than) can make your code harder to read. People generally read the `bge` instruction as "branch to a label because the comparison produced greater than or equal," not as "fall through if the comparison result was less than."

To help make such logic clearer, the *aoaa.inc* include file contains macros for several *opposite branches*. Table 2-11 lists these macros and their meanings.

Table 2-11: Opposite Branches

Opposite branch	Equivalent to	Meaning
`bnhs`	`blo`	Branch if not higher or the same. After a comparison, this branch will be taken if the first cmp operand is not higher or the same (not greater than or equal to, unsigned) the second operand.
`bnhi`	`bls`	Branch if not higher. After a comparison, this branch will be taken if the first cmp operand is not higher (not greater than, unsigned) the second operand.
`bnls`	`bhi`	Branch if not lower or the same. After a comparison, this branch will be taken if the first cmp operand is not lower or the same (not less than or equal to, unsigned) the second operand.
`bnlo`	`bhs`	Branch if not lower. After a comparison, this branch will be taken if the first cmp operand is not lower (not less than, unsigned) the second operand.
`bngt`	`ble`	Branch if not greater than. After a comparison, this branch will be taken if the first cmp operand is not greater than (signed) the second operand.
`bnge`	`blt`	Branch if not greater than or equal. After a comparison, this branch will be taken if the first cmp operand is not greater than or equal to (signed) the second operand.
`bnlt`	`bge`	Branch if not less than. After a comparison, this branch will be taken if the first cmp operand is not less than (signed) the second operand.
`bnle`	`bgt`	Branch if not less than or equal. After a comparison, this branch will be taken if the first cmp operand is not less than or equal to (signed) the second operand.

You should read each of these opposite-branch mnemonics as "fall through on condition" (ignoring the *not*).

2.11 Shifts and Rotates

The *shift* and *rotate* operations are another set of logical operations that apply to bit strings. These two categories can be further broken into left shifts, left rotates, right shifts, and right rotates.

The *shift-left operation* moves each bit in a bit string one position to the left, as shown in Figure 2-8.

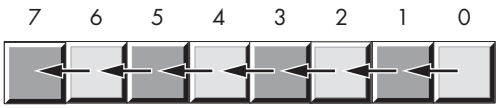


Figure 2-8: The shift-left operation

Bit 0 moves into bit position 1, the previous value in bit position 1 moves into bit position 2, and so on. You'll shift a 0 into bit 0, and the previous value of the HO bit will be lost.

The ARM provides a logical shift-left instruction, `lsl`, that performs this useful operation. This is the syntax for `lsl`:

```
lsl  dest, source, count // Does not affect any flags
```

The *count* operand is either a register or an immediate constant in the range 0 to *n*, where *n* is one less than the number of bits in the destination operand (for example, *n* = 31 for 32-bit operands and *n* = 63 for 64-bit operands). The *dest* and *source* operands are registers.

When the *count* operand is the value 1 (either an immediate constant or in a register), the `lsl` instruction performs the operation shown in Figure 2-9.

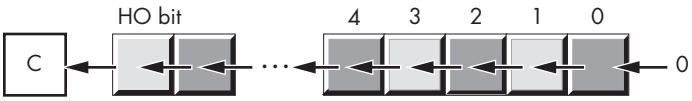


Figure 2-9: Shift-left operation

If the *count* value is 0, no shift occurs and the value remains unchanged. If the *count* value is greater than 1, the `lsl` instruction shifts the specified number of bits (shifting 0s into the LO position). Note that the `lsl` instruction does not affect any flags.

Shifting a value to the left by one digit is the same thing as multiplying it by its radix (base). For example, shifting a decimal number one position to the left (adding a 0 to the right of the number) effectively multiplies it by 10 (the radix):

```
1234 shl 1 = 12340
// (shl 1 means shift one digit position to the left.)
```

Because the radix of a binary number is 2, shifting it left multiplies it by 2. If you shift a value to the left *n* times, you multiply that value by 2^n.

A *shift-right* operation works the same way, except you're moving the data in the opposite direction. For a byte value, bit 7 moves into bit 6, bit 6 moves into bit 5, bit 5 moves into bit 4, and so on. During a shift right, you'll move a 0 into bit 7 (see Figure 2-10).

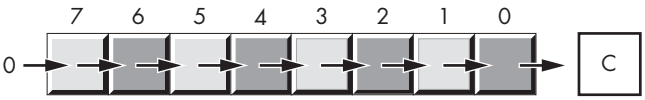


Figure 2-10: The shift-right operation

As you'd expect, the ARM provides an `lsr` instruction that shifts the bits to the right in a destination operand. The syntax is similar to the `lsl` instruction:

```
lsr  dest, source, count // Does not affect any flags
```

This instruction shifts a 0 into the HO bit of the destination operand and shifts the other bits one place to the right (that is, from a higher bit number to a lower bit number).

Because a shift left is equivalent to a multiplication by 2, it should come as no surprise that a shift right is roughly comparable to a division by 2 (or, in general, a division by the radix of the number). If you perform *n* shift-right operations, you will divide that number by 2^n.

However, a shift right is equivalent to only an *unsigned* division by 2. For example, if you shift the unsigned representation of 254 (0xFE) one place to the right, you get 127 (0x7F), exactly what you would expect. However, if you shift the two's complement representation of –2 (0xFE) to the right one position, you get 127 (0x7F), which is *not* correct. This problem occurs because you're shifting a 0 into bit 7. If bit 7 previously contained a 1, you're changing it from a negative to a positive number—not a good thing to do when dividing by 2.

To use the shift right as a division operator, this chapter must define a third shift operation: *arithmetic shift right*. There is no need for an arithmetic shift left; the standard shift-left operation works for both signed and unsigned numbers, assuming no overflow occurs.

An arithmetic shift right works just like the normal shift-right operation (a logical shift right), except instead of shifting a 0 into the HO bit, an arithmetic shift-right operation copies the HO bit back into itself. That is, the shift operation does not modify the HO bit, as Figure 2-11 shows.

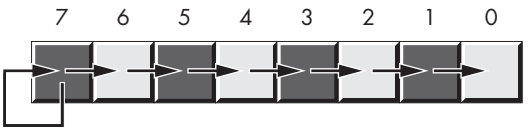


Figure 2-11: Arithmetic shift-right operation

An arithmetic shift right generally produces the signed integer result you expect. For example, if you perform the arithmetic shift-right operation on –2 (0xFE), you get –1 (0xFF). However, this operation always rounds the numbers to the closest integer that is *less than or equal to the actual result*. For example, if you apply the arithmetic shift-right operation on –1 (0xFF), the result is –1, not 0. Because –1 is less than 0, the arithmetic shift-right operation rounds toward –1. This is not a bug in the arithmetic shift-right operation; it just uses a different (though valid) definition of integer division.

The ARM-64 provides an arithmetic shift-right instruction, `asr` (arithmetic shift right). This instruction's syntax is nearly identical to `lsl`:

```
asr  dest, source, count // Does not affect any flags
```

The usual limitations on the operands apply. This instruction operates as shown in Figure 2-12 if the count is 1.

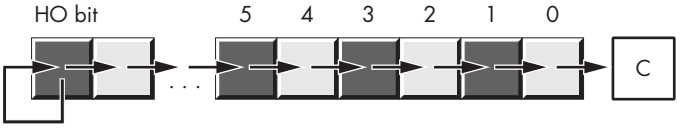


Figure 2-12: The `asr` `dest`, `source`, `#1` operation

If the *count* value is 0, no shift occurs and the value remains unchanged. If the *count* value is greater than 1, the `asr` instruction shifts the specified number of bits (shifting 0s into the LO position).

The *rotate-left* and *rotate-right* operations behave like the shift-left and shift-right operations, except the bit shifted out from one end is shifted back in at the other end. Figure 2-13 diagrams these operations.

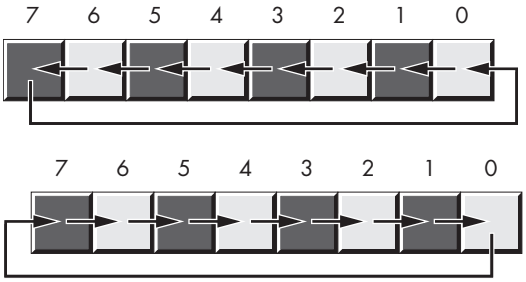


Figure 2-13: The rotate-left and rotate-right operations

The ARM provides a `ror` (rotate-right) instruction, but it does not have a rotate-left instruction. The syntax for the rotate right is similar to the shift instructions:

```
ror  dest, source, count // Does not affect any flags
```

Figure 2-14 shows the operation of this instruction on a register. Note that this instruction does not affect any flags. If the *count* value is 0, no rotate occurs and the value remains unchanged. If the *count* value is greater than 1, the rotate instructions rotate the specified number of bits (shifting 0s into the appropriate position).

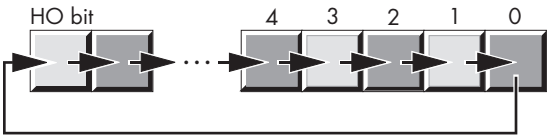


Figure 2-14: The `ror` `dest`, `source`, `#1` operation

If you absolutely need a `rol` operation, it can be (somewhat) synthesized using other instructions. Chapter 8 covers this in greater detail.

2.12 Bit Fields and Packed Data

Although the ARM operates most efficiently on byte, half-word, word, and dword data types, occasionally you'll need to work with a data type that uses a number of bits other than 8, 16, 32, or 64. You could zero-extend a nonstandard data size to the next larger power of 2 (such as extending a 22-bit value to a 32-bit value); this turns out to be fast, but if you have a

large array of such values, slightly more than 31 percent of the memory is going to waste (10 bits in every 32-bit value). However, suppose you were to repurpose those 10 bits for something else. By *packing* the separate 22-bit and 10-bit values into a single 32-bit value, you don't waste any space.

For example, consider a date of the form 04/02/01. Representing this date requires three numeric values: month, day, and year values. Months, of course, take on the values 1 to 12. At least 4 bits, a maximum of 16 values, are needed to represent the month. Days range from 1 to 31. This means it will take 5 bits, a maximum of 32 values, to represent the day entry. The year value, assuming that you're working with values in the range 0 to 99, requires 7 bits, which can be used to represent up to 128 values. This means we need 2 bytes to hold the whole date, since 4 + 5 + 7 = 16 bits.

In other words, you can pack the date data into 2 bytes rather than the 3 that would be required if you used a separate byte for each of the month, day, and year values. This saves 1 byte of memory for each date stored, which could make for significant savings if you need to store many dates. The bits could be arranged as shown in Figure 2-15.

Figure 2-15: Short packed-date format (2 bytes)

In the figure, *MMMM* represents the 4 bits making up the month value, *DDDDD* represents the 5 bits making up the day, and *YYYYYYY* represents the 7 bits composing the year. Each collection of bits representing a data item is a *bit field*. For example, April 2, 2001, would be represented as 0x4101:

```
0100      00010   0000001  = 0100_0001_0000_0001b or 0x4101
4         2       01
```

Although packed values are *space efficient* (that is, they make efficient use of memory), they are computationally *inefficient* (slow!). That's because unpacking the data packed into the various bit fields requires extra instructions. These take additional time to execute and additional bytes to hold the instructions; hence, you must carefully consider whether packed data fields will save you anything. The sample program in Listing 2-4 demonstrates the effort that goes into packing and unpacking this 16-bit date format.

```
// Listing2-4.S
//
// Demonstrate packed data types.

#include "aoaa.inc"

            .equ    NULL, 0         // Error code
            .equ    maxLen, 256     // Max input line size
```

```
            .data

saveLRMain:  .dword  0
saveLRRN:    .dword  0

ttlStr:      .asciz  "Listing 2-4"
moPrompt:    .asciz  "Enter current month: "
dayPrompt:   .asciz  "Enter current day: "

yearPrompt:  .ascii  "Enter current year "
             .asciz  "(last 2 digits only): "

packed:      .ascii  "Packed date is %04x = "
             .asciz  "%02d/%02d/%02d\n"

theDate:     .asciz  "The date is %02d/%02d/%02d\n"

badDayStr:   .ascii  "Bad day value was entered "
             .asciz  "(expected 1-31)\n"

badMonthStr: .ascii "Bad month value was entered "
             .asciz "(expected 1-12)\n"

badYearStr:  .ascii "Bad year value was entered "
             .asciz "(expected 00-99)\n"

// These need extra padding so they can be printed
// as integers. They're really byte (and word) values.

month:       .dword  0
day:         .dword  0
year:        .dword  0
date:        .dword  0

m:           .dword  0
d:           .dword  0
y:           .dword  0

input:       .fill    maxLen, 0

             .text
             .align  2       // Word-align code
             .extern printf
             .extern readLine
             .extern strtol

// Return program title to C++ program:

             .global getTitle
getTitle:
             lea     x0, ttlStr
             ret
```

```
// Here's a user-written function that reads a numeric value from
// the user:
//
// int readNum( char *prompt );
//
// A pointer to a string containing a prompt message is passed in
// the X0 register.
//
// This procedure prints the prompt, reads an input string from
// the user, then converts the input string to an integer and
// returns the integer value in X0.

readNum:
            lea     x1, saveLRRN
            str     lr, [x1]        // Save return address.

// Must set up stack properly (using this "magic" instruction)
// before you can call any C/C++ functions:

            sub     sp, sp, #64

// Print the prompt message. Note that the prompt message was
// passed to this procedure in X0; we're just passing it on to
// printf:

            bl      printf

// Set up arguments for readLine and read a line of text from
// the user. Note that readLine returns NULL (0) in RAX if there
// was an error.

            lea     x0, input
            mov     x1, #maxLen
            bl      readLine

// Test for a bad input string:

            cmp     x0, #NULL
            beq     badInput

// Okay, good input at this point. Try converting the string
// to an integer by calling strtol. The strtol function returns
// 0 if there was an error, but this is a perfectly fine
// return result, so we ignore errors.

            lea     x0, input       // Ptr to string
            mov     x1, #NULL       // No end string pointer
            mov     x2, #10         // Decimal conversion
            bl      strtol          // Convert to integer.

badInput:
            add     sp, sp, #64     // Undo stack setup.
            lea     x1, saveLRRN    // Restore return address.
            ldr     lr, [x1]
            ret
```

```
// Here is the "asmMain" function:


            .global asmMain
asmMain:
            sub     sp, sp, #64     // Magic instruction
            lea     x0, saveLRMain
            str     lr, [x0]

// Read the date from the user. Begin by reading the month:

            lea     x0, moPrompt
            bl      readNum

// Verify the month is in the range 1..12:

            cmp     x0, #1
            blo     badMonth
            cmp     x0, #12
            bhi     badMonth

// Good month, save it for now:

            lea     x1, month
            strb    w0, [x1]    // 1..12 fits in a byte.

// Read the day:

            lea     x0, dayPrompt
            bl      readNum

// We'll be lazy here and verify only that the day is in
// the range 1..31.

            cmp     x0, #1
            blo     badDay
            cmp     x0, #31
            bhi     badDay

// Good day, save it for now:

            lea     x1, day
            strb    w0, [x1]    // 1..31 fits in a byte.

// Read the year:

            lea     x0, yearPrompt
            bl      readNum

// Verify that the year is in the range 0..99:

            cmp     x0, #0
            blo     badYear
```

```
            cmp     x0, #99
            bhi     badYear

// Good year, save it for now:

            lea     x1, year
            strb    w0, [x1] // 0..99 fits in a byte.

// Pack the data into the following bits:
//
//  15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
//   m  m  m  m  d  d  d  d  d  y  y  y  y  y  y  y

            lea     x0, month
            ldrb    w1, [x0]
            lsl     w1, w1, #5

            lea     x0, day
            ldrb    w2, [x0]
            orr     w1, w1, w2
            lsl     w1, w1, #7

            lea     x0, year
            ldrb    w2, [x0]
            orr     w1, w1, w2

            lea     x0, date
            strh    w1, [x0]

// Print the packed date:

            lea     x0, packed
            vparm2  date
            vparm3  month
            vparm4  day
            vparm5  year
            bl      printf

// Unpack the date and print it:

            lea     x0, date
            ldrh    w1, [x0]

            // Extract month:

            lsr     w2, w1, #12
            lea     x0, m
            strb    w2, [x0]

            // Extract day:

            lsr     w3, w1, #7
            and     w3, w3, #0x1f
            lea     x0, d
            strb    w3, [x0]
```

```
            // Extract year:

            and     w1, w1, #0x7f
            lea     x0, y
            strb    w1, [x0]

            lea     x0, theDate
            vparm2  m
            vparm3  d
            vparm4  y
            bl      printf

            b.al    allDone

// Come down here if a bad day was entered:

badDay:
            lea     x0, badDayStr
            bl      printf
            b.al    allDone

// Come down here if a bad month was entered:

badMonth:
            lea     x0, badMonthStr
            bl      printf
            b.al    allDone

// Come here if a bad year was entered:

badYear:
            lea     x0, badYearStr
            bl      printf

allDone:
            add     sp, sp, #64
            lea     x0, saveLRMain
            ldr     lr, [x0]
            ret     // Returns to caller
```

Here's the result of building and running this program:

```
$ ./build Listing2-4
$ ./Listing2-4
Calling Listing2-4:
Enter current month: 2
Enter current day: 4
Enter current year (last 2 digits only): 56
Packed date is 2238 = 02/04/56
The date is 02/04/56
Listing2-4 terminated
```

The infamous problems with Y2K (year 2000) taught everyone that using a date format limited to 100 years (or even 127 years) would be quite

foolish. If you're too young to remember this fiasco, programmers in the middle to late 1900s used to encode only the last two digits of the year in their dates. When the year 2000 rolled around, these programs were incapable of distinguishing dates like 2024 and 1924.

To avoid this problem and future-proof the packed-date format in Listing 2-4, you can extend the format to 4 bytes packed into a double-word variable, as shown in Figure 2-16. (As you'll see in Chapters 3 and 4, you should always try to create data objects whose length is an even power of 2—that is, 1 byte, 2 bytes, 4 bytes, 8 bytes, and so on—or you will pay a performance penalty.)

Figure 2-16: The long packed-date format (4 bytes)

The month and day fields now consist of 8 bits each, so they can be extracted as a byte object from the word. This leaves 16 bits for the year, with a range of 65,536 years. By rearranging the bits so the year field is in the HO bit positions, the month field is in the middle bit positions, and the day field is in the LO bit positions, the long date format allows you to easily compare two dates to see whether one date is less than, equal to, or greater than another date. Consider the following code:

```
    lea x0, Date1  // Assume Date1 and Date2 are words.
    ldr x1, [x0]   // Using the long packed-date format
    lea x0, Date2
    ldr x2, [x0]
    cmp x1, x2
    ble d1LEd2

      // Do something if Date1 > Date2.

d1LEd2:
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare `Date1` and `Date2` as easily as for the short packed-date format. Therefore, this example demonstrates another reason for packing data even if you don't realize any space savings: it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Examples of practical packed data types abound. You could pack eight Boolean values into a single byte, two BCD digits into a byte, and so on. A classic example of packed data is the PSTATE register (see Figure 2-17). This register packs four important Boolean objects, along with 12 important system flags, into a single 32-bit register.

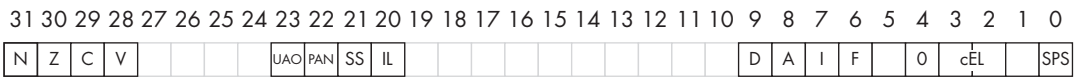


Figure 2-17: The PSTATE register as packed Boolean data

You'll commonly access the condition code flags by using the conditional jump instructions. Occasionally, you may need to manipulate the individual condition code bits in the PSTATE register. You can do this with the `msr` (move to system register) and `mrs` (move system register) instructions

```
msr  systemReg, reg
mrs  reg, systemReg
```

where *reg* is one of the ARM's 64-bit general-purpose registers and *systemReg* is a special *system register* name. The system register of interest here is NZCV, named after the condition code flags.

The following instruction copies bits 28 to 31 in the PSTATE register into the corresponding bits in X0 and copies 0s to all the other bits in X0:

```
mrs  x0, nzcv
```

This instruction copies bits 28 to 31 in X0 to the condition code bits in PSTATE (without affecting any other bits in PSTATE):

```
msr  nzcv, x0
```

If you want to explicitly set the carry flag, without affecting any other condition code flags, you could do that as follows:

```
mrs  x0, nzcv
orr  x0, x0, #0x20000000  // Carry is in bit 29; set it.
msr  nzcv, x0
```

This ORs a 1 bit into the carry flag in the PSTATE register.

2.13 IEEE Floating-Point Formats

Back in 1976, when Intel planned to introduce a floating-point coprocessor for its new 8086 microprocessor, it hired the best numerical analyst it could find to design a floating-point format. That person then hired two other experts in the field, and the three of them—William Kahan, Jerome Coonen, and Harold Stone—designed Intel's floating-point format. They did such a good job designing the KCS floating-point standard that the Institute of Electrical and Electronics Engineers (IEEE) adopted it for its floating-point format. That format has become the standard used by CPU vendors, including Arm.

The IEEE-754 standard single- and double-precision formats correspond to C's float and double types or FORTRAN's real and double-precision types. These same formats are available to ARM assembly language programmers.

2.13.1 Single-Precision Format

The *single-precision format* uses a one's complement 24-bit mantissa, an 8-bit excess-127 exponent, and a single sign bit. *One's complement* notation consists of a sign bit and an unsigned binary number, with the sign bit indicating the sign of the binary number. The *mantissa* (the part of the number that represents the significant digits) usually represents a value from 1.0 to just under 2.0. The HO bit of the mantissa is always assumed to be 1 and represents a value just to the left of the binary point. (A *binary point* is the same thing as a *decimal point*, except it appears in binary numbers rather than decimal numbers.) The remaining 23 mantissa bits (the fraction) appear to the right of the binary point.

Therefore, the mantissa represents the value:

```
1.mmmmmmm mmmmmmmm
```

The *mmmm* characters represent the 23 bits of the mantissa. Because the HO bit of the mantissa is always 1, the single-precision format doesn't actually store this bit within the 32 bits of the floating-point number. This HO bit is known as an *implied bit*.

Because you are working with binary numbers, each position to the right of the binary point represents a value (0 or 1) times a successive negative power of 2. The implied `1` bit is always multiplied by 2^0, which is 1. This is why the mantissa is always greater than or equal to 1.0. Even if the other mantissa bits are all 0s, the implied `1` bit always gives us the value 1.0. Of course, even if you had an almost infinite number of 1 bits after the binary point, they still would not add up to 2.0. This is why the mantissa can represent values in the range 1.0 to just under 2.0.

There is one exception to the implied bit always being 1: the IEEE floating-point format supports *denormalized* values, where the HO bit is not 0. However, this book generally ignores denormalized values.

Although there is an infinite number of values between 1.0 and 2.0, you can represent only 8 million of them because the format uses a 23-bit mantissa (with the implied 24th bit always being `1`). This is the reason for inaccuracy in floating-point arithmetic: you are limited to a fixed number of bits in computations involving single-precision floating-point values.

As noted, the mantissa uses a one's complement format rather than two's complement to represent signed values. This means that the 24-bit value of the mantissa is simply an unsigned binary number, and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for 0.0 (with the sign bit set or clear). Generally, this is important only to the person designing the floating-point software or hardware system. This book assumes that the value 0.0 always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating-point format comes into play. The floating-point format raises 2 to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is 8 bits and is stored in an excess-127 format. In *excess-127 format*, the exponent 0 is represented by the value 127 (0x7F), negative exponents are values in the range 1 to 126, and positive exponents are values in the range 128 to 254 (0 and 255 are reserved for special cases). To convert an exponent to excess-127 format, add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating-point values.

The single-precision floating-point format takes the form shown in Figure 2-18.

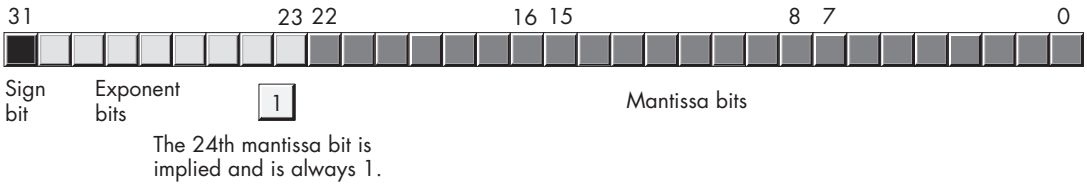


Figure 2-18: The single-precision (32-bit) floating-point format

With a 24-bit mantissa, you will get approximately six and a half (decimal) digits of precision (half a digit of precision means that the first six digits can all be in the range 0 to 9, but the seventh digit can only be in the range 0 to *x*, where $x < 9$ and is generally close to 5). Note, however, that only six digits are guaranteed. With an 8-bit excess-127 exponent, the *dynamic range* of single-precision floating-point numbers is approximately $2^{\pm 127}$, or about $10^{\pm 38}$. This dynamic range is the difference in size between the smallest and largest positive values.

Although single-precision floating-point numbers are perfectly suitable for many applications, the precision and dynamic range are somewhat limited and unsuitable for many financial, scientific, and other applications. Furthermore, during long chains of computations, the limited accuracy of the single-precision format may introduce serious errors.

2.13.2 Double-Precision Format

The *double-precision format* helps overcome the problems of single-precision floating-point. Using twice the space, the double-precision format has an 11-bit excess-1,023 exponent and a 53-bit mantissa (with an implied HO bit of 1), plus a sign bit. Double-precision floating-point values take the form shown in Figure 2-19.

Figure 2-19: The 64-bit double-precision floating-point format

The 53rd mantissa bit is implied and is always 1. The double-precision format provides a dynamic range of about $10^{\pm 308}$ and at least 15 digits of precision, sufficient for most applications.

2.14 Normalized Floating-Point Values

To maintain maximum precision during computation, most computations use normalized values. A *normalized floating-point value* is one whose HO mantissa bit contains 1. Almost any nonnormalized value can be normalized: shift the mantissa bits to the left and decrement the exponent until a 1 appears in the HO bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating-point value by 2. Likewise, whenever you decrement the exponent, you divide the floating-point value by 2. By the same token, shifting the mantissa to the left one bit-position multiplies the floating-point value by 2; likewise, shifting the mantissa to the right divides the floating-point value by 2. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating-point number at all.

Keeping floating-point numbers normalized maintains the maximum number of bits of precision for a computation. If the HO *n* bits of the mantissa are all 0s, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating-point computation will be more accurate if it involves only normalized values.

In two important cases, a floating-point number cannot be normalized. First, the floating-point value 0.0 can't be normalized, because the representation for 0.0 has no 1 bits in the mantissa. This, however, is not a problem because you can exactly represent the value 0.0 with only a single bit.

In the second case, you have some HO bits in the mantissa that are 0s, but the biased exponent is also 0 (and you cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose HO mantissa bits and biased exponent are 0 (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values. (The alternative would be to underflow the values to 0.) Although the use of denormalized values allows IEEE floating-point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer fewer bits of precision. Some texts use the term *subnormal* to describe denormalized values.

2.14.1 Nonnumeric Values

The IEEE floating-point standard recognizes four special nonnumeric values: –infinity, +infinity, and two special not-a-number (NaN) values. For each of these special numbers, the exponent field is filled with all 1 bits.

If the exponent is all 1 bits and the mantissa is all 0 bits (excluding the implied bit), then the value is infinity. The sign bit will be 0 for +infinity and 1 for –infinity.

If the exponent is all 1 bits and the mantissa is not all 0 bits, the value is an invalid number (known as a *NaN* in IEEE 754 terminology). NaNs represent illegal operations, such as trying to take the square root of a negative number.

Unordered comparisons occur whenever either operand (or both) is a NaN. As NaNs have an indeterminate value, they are incomparable. Any attempt to perform an unordered comparison typically results in an exception or some sort of error. Ordered comparisons, on the other hand, involve two operands, neither of which is a NaN.

2.14.2 Gas Support for Floating-Point Values

Gas provides a couple of data declarations to support the use of floating-point data in your assembly language programs. Gas floating-point constants allow the following syntax: the constant begins with an optional + or – symbol, denoting the sign of the mantissa (if this is not present, Gas assumes that the mantissa is positive). This is followed by one or more decimal digits, then a decimal point and zero or more decimal digits. These are optionally followed by an `e` or `E`, which is in turn optionally followed by a sign (+ or –) and one or more decimal digits.

The decimal point or the `e`/`E` must be present to differentiate a floating-point literal constant from an integer or unsigned literal constant. Here are some examples of legal floating-point literal constants:

```
1.234  3.75e2  -1.0  1.1e-1  1.e+4  0.1  -123.456e+300  +25.0e0
```

A floating-point literal constant must begin with a decimal digit, so you must use, for example, `0.1` rather than `.1` in your programs.

To declare a floating-point variable, use the `.single` or `.double` data types. Aside from using these types to declare floating-point variables rather than integers, their use is nearly identical to that of `.byte`, `.word`, `.dword`, and so on. The following examples demonstrate these declarations and their syntax:

```
          .data

fltVar1:  .single  0.0
fltVar1a: .single  2.7
pi:       .single  3.14159
DblVar:   .double  0.0
DblVar2:  .double  1.23456789e+10
DPVar:    .double  -1.0e-104
IntAsFP:  .double  -123
```

As usual, this book uses the C/C++ `printf()` function to print floating-point values to the console output. Certainly, an assembly language routine could be written to provide this same output, but the C stdlib provides a convenient way to avoid writing that complex code (at least until Chapter 9).

Floating-point arithmetic is different from integer arithmetic; you cannot use the ARM `add` and `sub` instructions to operate on floating-point values. This chapter presents only the floating-point formats; see Chapter 6 for more information on floating-point arithmetic and general floating-point operations.

In the meantime, let's consider some other data formats.

2.15 Binary-Coded Decimal Representation

Although the integer and floating-point formats cover most of the numeric needs of an average program, in some special cases other numeric representations are convenient. This section expands on the definition of the BCD format presented earlier. Although the ARM CPU doesn't provide hardware support for BCD, it's still a common format that some software uses, with BCD arithmetic provided by programmer-written software functions.

BCD values are a sequence of nibbles, with each nibble representing a value in the range 0 to 9. With a single byte, you can represent values containing two decimal digits, or values in the range 0 to 99. Figure 2-20 shows the two BCD digits, represented by 4 bits each, in a byte.

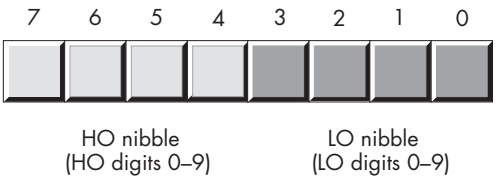


Figure 2-20: Two-digit BCD data representation in memory

As you can see, BCD storage isn't particularly memory efficient. For example, an 8-bit BCD variable can represent values in the range 0 to 99, while that same 8 bits, when holding a binary value, can represent values in the range 0 to 255. Likewise, a 16-bit binary value can represent values in the range 0 to 65,535, while a 16-bit BCD value can represent only about one-sixth of those values (0 to 9,999).

However, it's easy to convert BCD values between the internal numeric representation and their string representation, for example, using BCD to encode multidigit decimal values in hardware, using a thumb wheel or dial. For these two reasons, you're likely to see people using BCD in embedded systems (such as toaster ovens, calculators, alarm clocks, and nuclear reactors) but rarely in general-purpose computer software.

Unfortunately, all BCD operations on ARM have to be done using software functions, as BCD arithmetic is not built into the hardware on the ARM. As a result, computations involving BCD arithmetic can run slowly. Because the BCD data type is very specialized and used in only a few situations (for example, in embedded systems), this book won't spend any more time discussing it.

2.16 Characters

Perhaps the most important data type on a personal computer is the character data type. *Character* refers to a human or machine-readable symbol that is typically a nonnumeric entity. Specifically, a character is any symbol that you can typically type on a keyboard (including symbols that may require multiple keypresses to produce) or display on a video display.

Letters (*alphabetic characters*), punctuation symbols, numeric digits, spaces, tabs, carriage returns (ENTER), other control characters, and other special symbols are all characters. *Numeric characters* are distinct from numbers: the character `1` is different from the value 1. The computer (generally) uses two internal representations for numeric characters (`0`, `1`, . . . , `9`) versus the numeric values 0 to 9.

Most computer systems use a single- or multibyte sequence to encode the various characters in binary form. Linux and macOS use either the ASCII or Unicode encodings for characters. This section discusses the ASCII and Unicode character sets and the character declaration facilities that Gas provides.

2.16.1 The ASCII Character Encoding

The ASCII character set maps 128 textual characters to the unsigned integer values 0 to 127 (0 to 0x7F). Although the exact mapping of characters to numeric values is arbitrary and unimportant, you must use a standardized code for this mapping so that when you communicate with other programs and peripheral devices, you all speak the same "language." ASCII is a standardized code: if you use the ASCII code 65 to represent the character A, then you know that a peripheral device (such as a printer) will correctly interpret this value as the character A whenever you transmit data to that device.

Despite some major shortcomings, ASCII has become the standard for data interchange across computer systems and programs. Most programs can accept and produce ASCII data. Because you will be dealing with ASCII characters in assembly language, I recommend you study the layout of the character set and memorize a few key ASCII codes (for example, for `0`, `A`, `a`, and so on). See Appendix A for a list of all the ASCII character codes.

Today, Unicode (especially the UTF-8 encoding) is rapidly replacing ASCII, because the ASCII character set is insufficient for handling international alphabets and other special characters, as you'll see in Chapter 14. Nevertheless, most modern code still uses ASCII, so you should be familiar with it.

The ASCII character set is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 to 0x1F (31), form a special set of nonprinting characters, the control characters. They are called *control characters* because they perform various printer/display control operations rather than display symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters; *line feed*, which moves the cursor down one line on the output device; and *backspace*, which moves the cursor back one position to the left. (Historically, *carriage return* refers to the paper carriage used on typewriters: physically moving the carriage all the way to the right enabled the next character typed to appear at the left side of the paper.) Unfortunately, different control characters perform different operations on different output devices. Little standardization exists among output devices. To find out exactly how a control character affects a particular device, consult its manual.

The second group of 32 ASCII character codes contains various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code 0x20) and the numeric digits (ASCII codes 0x30 to 0x39).

The third group of 32 ASCII characters contains the uppercase alphabetic characters. The ASCII codes for the characters A through Z lie in the range 0x41 to 0x5A (65 to 90). Because there are only 26 alphabetic characters, the remaining 6 codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes represents the lowercase alphabetic symbols, 5 additional special symbols, and another control character (DELETE). The lowercase character symbols use the ASCII codes 0x61 to 0x7A. If you convert the codes for the upper- and lowercase characters to binary, you will notice that the uppercase symbols differ from their lowercase equivalents in exactly one bit position. For example, consider the character codes for `E` and `e` in Figure 2-21.

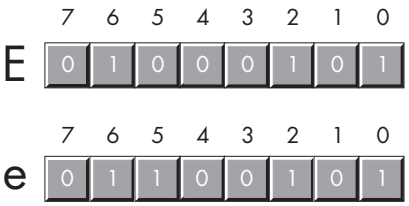


Figure 2-21: The ASCII codes for E and e

The only place upper- and lowercase differ is in bit 5. Uppercase characters always contain a 0 in bit 5; lowercase alphabetic characters always contain a 1 in bit 5. You can use this fact to quickly convert between upper- and lowercase. You can force an uppercase character to lowercase by setting bit 5 to 1, or force a lowercase character to uppercase by setting bit 5 to 0.

Indeed, bits 5 and 6 determine which of the four groups in the ASCII character set you're in, as Table 2-12 shows. You could, for instance, convert any upper- or lowercase (or corresponding special) character to its equivalent control character by setting bits 5 and 6 to 0.

Table 2-12: ASCII Groups

Bit 6	Bit 5	Group
0	0	Control characters
0	1	Digits and punctuation
1	0	Uppercase and special
1	1	Lowercase and special

Consider the ASCII codes of the numeric digit characters in Table 2-13.

Table 2-13: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
0	48	30h
1	49	31h
2	50	32h
3	51	33h
4	52	34h
5	53	35h
6	54	36h
7	55	37h
8	56	38h
9	57	39h

The LO nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (that is, setting to 0) the HO nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0 to 9 to its ASCII character representation by simply setting the HO nibble to 3. You can use the AND operation to force the HO bits to 0; likewise, you can use the OR operation to force the HO bits to 0b0011 (3).

Unfortunately, you *cannot* convert a string of numeric characters to its equivalent binary representation by simply stripping the HO nibble from each digit in the string. Converting 123 (0x31, 0x32, 0x33) in this fashion yields 3 bytes, or 0x010203, but the correct value for 123 is 0x7B. The conversions described in the preceding paragraph work only for single digits.

2.16.2 Gas Support for ASCII Characters

Gas provides support for character variables and literals in your assembly language programs. Character literal constants in Gas consist of a character surrounded by a pair of apostrophes (or single quotes):

```
'A'
```

Technically, a character constant in Gas consists of a single apostrophe followed by a single character. Gas allows a second version consisting of a character surrounded by apostrophes. However, the macOS assembler supports only the latter form, so this book uses only that form to ensure that all example code will assemble on both systems.

To represent an apostrophe as a character constant, use the backslash character followed by an apostrophe. For example:

```
'\''
```

You can also use the other escape character sequences in a character constant. See section 1.7, "Declaring Memory Variables in Gas," on page 16 for details.

To declare a character variable in a Gas program, use the `.byte` declaration. For example, the following declaration demonstrates how to declare a variable named `UserInput`:

```
          .data
UserInput: .byte  0
```

This declaration reserves 1 byte of storage that you could use to store any character value. You can also initialize character variables as follows:

```
             .data
TheCharA:    .byte 'A'
ExtendedChar: .byte 128  // Character code greater than 0x7F
```

Because character variables are 8-bit objects, you can manipulate them as you would any 8-bit value. You can move character variables into registers and store the LO byte of a register into a character variable.

2.17 Gas Support for the Unicode Character Set

Unfortunately, ASCII supports only 128 character codes. Even if you extend the definition to 8 bits (as IBM did on the original PC), you're limited to 256 characters. This is far too small for modern multinational, multilingual applications. Back in the 1990s, several companies developed an extension to ASCII, known as *Unicode*, using a 2-byte character size. Therefore, the original Unicode supported up to 65,536 character codes.

As well-thought-out as the original Unicode standard was, systems engineers discovered that even 65,536 symbols were insufficient. Today, Unicode defines 1,112,064 possible characters (code points), encoded using a variable-length character format.

Unfortunately, Gas provides almost no support for Unicode text in a source file. Certainly, if you have a text editor that supports editing UTF-8 source files, Gas will accept UTF-8 characters in character and string literals. However, it probably won't do much with Unicode beyond that (I haven't tried this, but I doubt Gas will accept UTF-16 or UTF-32 source files).

Chapter 14 covers Unicode format and implementation in much greater detail.

2.18 Machine Code

Gas translates human-readable source files into a special binary form known as *machine code*. With many (non-RISC) CPUs, it is possible to work in assembly language without knowing much about the underlying machine code that the assembler produces. With RISC processors, such as the ARM, you must have a basic understanding of the underlying machine code in order to understand how to write decent assembly language source code.

Like most RISC CPUs, the ARM64 translates individual machine instructions into a single 32-bit value. This is one of the fundamental principles behind RISC: instructions are always the same length on a given CPU, and that length is almost always 32 bits. Variable-length instructions are verboten. However, if the instruction set supports immediate constants (which the assembler encodes as part of the machine instruction), and you have 64-bit registers, how do you load a 64-bit immediate constant into a register when the instructions are limited to 32 bits? The short answer is, "You don't." You may recall from Chapter 1 that immediate constants were limited to a very small range, and now you know why: the constants must be encoded into a 32-bit instruction value, along with considerable other information. This severely limits the size of immediate constants.

> **UNDER MY THUMB**
>
> The 32-bit variants of the ARM support a special 16-bit instruction-length mode known as the Thumb instruction set. This was done to reduce the size of programs in cost-sensitive embedded applications. In fact, certain embedded versions of the ARM support only the Thumb instruction set. However, Thumb extensions are definitely non-RISC-like. The ARM64 CPUs do not support the Thumb instruction set (while operating in 64-bit mode), as most ARM64 CPUs have a fair amount of memory installed in the system.

Immediate constants aren't the only thing you must encode within an instruction's 32-bit value. Every instruction operand will require a certain number of bits to encode. For example, the ARM64 CPU has 32 general-purpose registers. It takes 5 bits to encode 32 values. Therefore, each register in an operand will consume 5 bits out of the 32 available for that instruction. The following `adds` instruction will require at least 15 bits to encode the three registers (as any general-purpose register is legal for the destination, first-source, and second-source registers):

```
adds x0, x1, x2
```

In addition to registers and constants, other pieces of information must be encoded in an ARM instruction, such as the size of the operation (32 bits versus 64 bits). Many instructions, like `adds` in the preceding example, allow immediate constants (as the second source operand) in addition to registers. There must be some way to differentiate those two operand forms, which take at least 1 bit. Many instructions provide an option to update the flags at the end of the instructions' execution, which takes another bit. Many additional options exist that this book hasn't even begun to cover. We're rapidly running out of bits.

RISC instructions must be not only fixed-length but also easy to decode using hardware. This means that for all instructions, a certain number of bits in fixed locations in the 32-bit instruction determine the type or classification of the instruction. Consider the basic instruction format for the ARM64 shown in Figure 2-22.

Figure 2-22: The basic ARM instruction format

The `op0` field (`op0` is short for *operation code 0*, itself usually shortened to *opcode*) specifies the instruction's operation. In this example, this 4-bit field divides the instruction set into seven components, as shown in Table 2-14.

Table 2-14: The `op0` 4-Bit Field in Instruction Encoding

op0	Encoding group or instruction page
0000 0001 0010 0011	Reserved/unallocated
1000 1001	Data processing instructions with immediate constants
1010 1011	Branches, exception-generating instructions, and system instructions
0100 0110 1100 1110	Loads and stores
0101 1101	Data processing instructions with registers
0111	Data processing: SIMD and floating-point instructions
1111	Data processing: SIMD and floating-point instructions

Consider the instructions in the second group in Table 2-14: data processing instructions with immediate constants. This group uses the decoding shown in Figure 2-23.

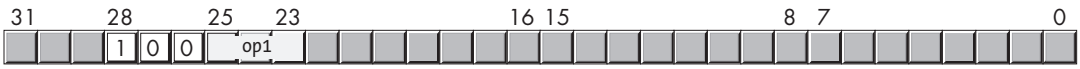


Figure 2-23: Encoding of data processing instructions with immediate constants

The 3 bits in `op1` (note that bit 25 is shared with `op0`) can be decoded as shown in Table 2-15.

Table 2-15: Instructions with `op0` Equal to 0b100

`op1`	Decoding group or instruction page
000 001	PC-relative addressing mode instructions
010 011	Add/subtract immediate instructions
100	Logical immediate instructions
101	Move Wide immediate instructions
110	Bitfield instructions
111	Extract instructions

Now consider the add/subtract immediate instructions group from Table 2-15. The full encoding for these instructions appears in Figure 2-24.

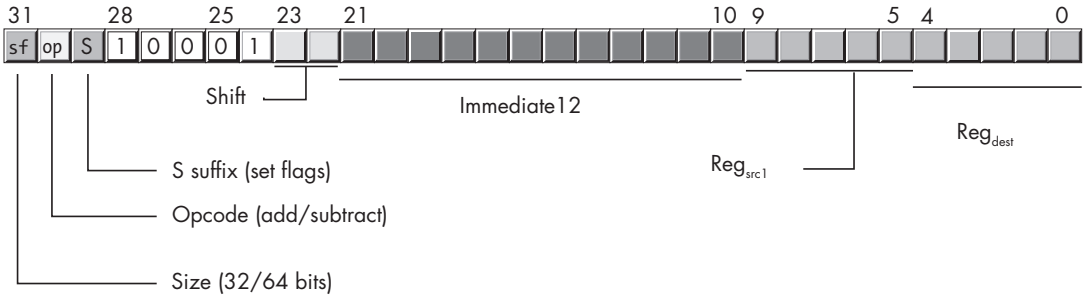


Figure 2-24: Add/subtract immediate instructions

The add and subtract instructions are a classic example of a packed-data field (as discussed in section 2.12, "Bit Fields and Packed Data," on page 85). The fields have the following meanings:

sf Indicates the instruction size (variant). If 0, this is a 32-bit instruction and the registers specified by the Reg_{src1} and Reg_{dest} fields are 32-bit registers. If 1, this is a 64-bit instruction, and the registers are 64-bit registers.

op (bit 30) Is an extension of the opcode (bits 24 through 28). If this bit is 0, the instruction is an `add/adds` instruction; if this bit is 1, it's a `sub/subs` instruction.

S Specifies whether there was an `s` suffix (for example, `adds`) on the instruction. If this bit is 1, the instruction will update the condition code flags after the execution of the instruction; if this bit is 0, no such update takes place.

Shift Specifies how the instruction treats the Immediate12 field. I'll discuss this field in greater detail shortly.

Immediate12 Is a 12-bit unsigned integer value (0 to +4,096). This instruction will zero-extend that value to the instruction's size (32 or 64 bits).

Reg_{src} Specifies the source register, the second operand for the instruction.

Reg_{dest} Specifies the destination register, the first operand for the instruction.

The Shift field depends on the Immediate12 field and is a bit complex. This field may contain 0b00 or 0b01 (0b10 and 0b11 are reserved values). If this field contains 0b00, the instruction uses the zero-extended value of the Immediate12 field as is. However, if this field contains 0b01, the instruction first shifts the Immediate12 to the left by 12 bits and uses that shifted value. This shifted form is useful when doing pointer arithmetic and adding in page offsets (see Chapter 3 for an explanation of memory-management pages).

If the add and subtract instructions are limited to a 12-bit immediate constant (possibly shifted to the left 12 bits), how do you add a 32- or 64-bit constant to a register? You can't do it directly; instead, you have to load that constant into another register and use that register as the second source operand rather than an immediate constant. As I pointed out earlier, the same problem arises with the `mov` instruction and immediate constants. As with add and subtract, the `mov` instruction is limited to 32 bits, meaning you cannot load a 32- or 64-bit constant into a register with a single `mov` instruction. The operative word here is *single*. You can load a 32- or 64-bit constant into a register by using multiple `mov` instructions. The next section discusses how to do this.

2.19 Operand2

Most ARM data processing instructions (such as `add` and `sub`) require three operands: a destination operand and two source operands. In the following instruction, X0 is the destination operand, X1 is the first source operand, and X2 is the second source operand:

```
add  x0, x1, x2  // Computes X0 = X1 + X2
```

Thus far in this book, I've used registers and immediate constants as the second source operand. However, the ARM CPUs support several formats for this second operand, known as *Operand2*. These forms, shown in Table 2-16, are extremely powerful, making Operand2 legendary on the ARM.

Table 2-16: Operand2 Allowable Fields

Operand2	Description
`#immediate`	A 12-bit immediate value of 0–4,095 (used by arithmetic instructions) or a 16-bit immediate value (used by move instructions).
`#pattern`	A constant that specifies a run of 0s and 1s. Used to generate a bitmask for the logical instructions. For logical instructions only.
`Wn` or `Xn`	One of the general-purpose registers (32- or 64-bit).
`Wn shiftOp #imm`	The contents of a 32-bit register shifted the number of positions specified by the `#imm` operand (0–31). *`shiftOp`* is `lsl`, `lsr`, `asr`, or `ror`.
`Xn shiftOp #imm`	The contents of a 64-bit register shifted the number of positions specified by the `#imm` operand (0–63).
`Wn extendOp #imm`	The contents of a 32-bit register are zero- or sign-extended and then shifted to the left by the immediate value (0–31). This form is not available for logical instructions, as sign extension doesn't logically apply to those. *`extendOp`* is `uxtb`, `uxth`, `uxtw`, `uxtx`, `sxtb`, `sxth`, `sxtw`, or `sxtx`.
`Xn extendOp #imm`	The contents of a 64-bit register are zero- or sign-extended and then shifted to the left by the immediate value (0–31). This form is not available for logical instructions, as sign extension doesn't logically apply to those.

The following sections describe each of these Operand2 forms.

2.19.1 #immediate

The immediate form of Operand2, or `#immediate`, is one of its more common uses (the other being one of the 32 general-purpose registers). Because the immediate operand is encoded as part of the 32-bit instruction value, it is always significantly less than 32 bits in length. As you've seen, the arithmetic instructions allow only a 12-bit unsigned integer as an immediate operand. Other instructions allow different immediate operand sizes. For example, the `mov` instruction allows 16-bit unsigned immediate operands.

Although many immediate constants you'll encounter in programs will fit into 12 or 16 bits, some values won't. As noted earlier in this chapter, in those situations you will have to load a register with the larger constant and use that value in that register, rather than using an immediate constant. See section 2.20, "Large Constants," on page 111 to learn how to handle this situation.

2.19.2 #pattern

The ARM logical instructions (such as `and`, `orr`, and `eor`) provide a 13-bit immediate (`#pattern`) field encoded into the 32-bit instruction. However, this is not a straightforward 13-bit immediate value. Instead, it's a combination of 3 separate bit fields that form a *bitmask pattern*. Chapter 12 describes the use of these bitmasks in greater detail. Until then, understand that there are some weird limitations on the type of immediate constants that the logical instructions support.

The Arm Compiler Armasm User Guide's entry is difficult to understand. Basically, it says that immediate constants for logical instructions consist of binary values that contain a run (consecutive sequence) of 1 bits followed by (and possibly preceded by) 0 bits. Each sequence can be 2, 4, 8, 16, 32, or 64 bits in length. The following are legal examples of such immediate constants:

```
and     x0, x0, #0b1
and     x0, x0, #0b11
and     x0, x0, #0b111
and     x0, x0, #0b1110
and     x0, x0, #0b11100
```

In each case, there is a single run of 1 bits, possibly surrounded by 0 bits.

The following examples are not legal immediate constants:

```
and     x0, x0, #0b101
and     x0, x0, #0b10101
and     x0, x0, #0b1110111
and     x0, x0, #0b101100
```

These examples are illegal because they contain multiple runs of 1 bits within the same immediate constant.

The "vector of identical elements" phrase (from the Armasm Guide) tells us that if the sequence is less than the register size (32 or 64 bits), the instruction replicates the sequence throughout the register in order to fill it to 32 or 64 bits. Therefore, it is possible to have multiple runs of 1 bits in an immediate constant if there are identical sequences of 1s and 0s, where each sequence is a multiple of 2, 4, 8, 16, or 32 bits in length. The following are legal examples:

```
// This AND instruction contains 4 copies of the sequence
// 0b11110000:

        and     w0, w0, #0b11110000111100001111000011110000

// This sequence is legal because it contains 16 copies of
// the 2-bit sequence 0b10:

        and     w0, w0, #0b01010101010101010101010101010101

// This sequence is legal because it contains 2 copies of
// the 32-bit sequence 0b11111111111111110000000000000000:

        and     x0, x0, #0xFFFF0000FFFF0000
```

However, if you want to use the "vector of identical elements" scheme, you must provide a constant that completely fills the destination register. The following example is illegal because it has two runs within 16 bits that are not replicated throughout the HO 16 bits of the 32-bit W0 register:

```
and     w0, w0, #0b1111000011110000
```

This scheme is confusing but generates the most common types of immediate constants with just a few bits, so the complexity is worth it.

If you accidentally supply an inappropriate constant, Gas will respond with an error message such as `error: expected compatible register or logical immediate` or `error: immediate out of range at operand 3 -- 'and w0,w0,#0b1111000011110000'`.

2.19.3 Register

The most common form for Operand2 is one of the ARM's general-purpose registers (32 or 64 bits). Given that registers have appeared in most examples thus far, there's no need to further discuss this form.

2.19.4 Shifted Register

Another Operand2 form combines an ARM register with a shift operation. This form adds an extra operand to the instruction, consisting of one of the shift operators in Table 2-17 along with a small immediate constant (in the range 0 to *n*, where *n* is the size of the destination register).

Table 2-17: Operand2 Shift Operators

Operator	Description
`lsl #`*`imm`*	Logically shifts a copy of the Operand2 register value to the left *`imm`* bits and uses the result.
`lsr #`*`imm`*	Logically shifts a copy of the Operand2 register value to the right *`imm`* bits and uses the result.
`asr #`*`imm`*	Arithmetically shifts a copy of the Operand2 register value to the right *`imm`* bits and uses the result.
`ror #`*`imm`*	Logically rotates a copy of the Operand2 register value to the right *`imm`* bits and uses the result. This form is available with only the logical instructions.

As you'll see in Chapter 4, using the shifted register Operand2 form will prove handy when indexing into arrays and other data structures.

To use the shifted register Operand2 form, simply tack on an extra operand to the end of the instruction's operand list with one of the operators appearing in Table 2-17. Here are some examples:

```
add w0, w1, w2, lsl #4  // W0 = W1 + (W2 << 4)
sub x0, x1, x2, lsr #1  // X0 = X1 - (X2 >> 1)
add x0, x1, x2, asr #1  // X0 = X1 + (X2 asr 1)
and x0, x1, x2, ror #2  // X0 = X1 & (X2 ror 2)
```

As the comments indicate, each of these instructions shifts the value in W2 or X2 before using that value as the second source operand.

2.19.5 Extending Register

The last set of Operand2 forms provide zero and sign extension, along with an optional logical shift left, of an Operand2 register. The basic instruction syntax is

```
instr  reg_dest, reg_src1, reg_src2, extendop #optional_imm
```

where *extendop* is one of the operators in Table 2-18. If the `#optional_imm` value is not present, it defaults to 0.

Table 2-18: Extend Operators

Extend operator	Description
`uxtb #optional_imm`	Zero-extends the LO byte of reg_{src2} to the size of reg_{dest} and reg_{src1}. The reg_{src2} operand should be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. (Gas seems to accept a dword register, substituting the corresponding word register.) If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits.
`uxth #optional_imm`	Zero-extends the LO half word of reg_{src2} to the size of reg_{dest}. The reg_{src2} operand should be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits.
`uxtw #optional_imm`	Zero-extends the LO word of reg_{src2} to the size of reg_{dest}. The reg_{src2} operand should be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits. Note that if all the registers are words (W*n*), then this operator is equivalent to `lsl #optional_imm`.
`uxtx #optional_imm`	This operator is applicable only when all the registers are 64 bits. This is the default condition if no extend (or shift) operator is present after an Operand2 register.
`sxtb #optional_imm`	Sign-extends the LO byte of reg_{src2} to the size of reg_{dest} and reg_{src1}. The reg_{src2} operand should be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits.
`sxth #optional_imm`	Sign-extends the LO half word of reg_{src2} to the size of reg_{dest}. The reg_{src2} operand should be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits.
`sxtw #optional_imm`	Sign-extends the LO word of reg_{src2} to the size of reg_{dest}. The reg_{src2} operand must be a word-sized register (W*n*), regardless of the size of reg_{dest} and reg_{src1}. If the optional immediate value is present, it must be a value in the range 0–4 and will shift the result of the extension by the specified number of bits. If all the registers are words (W*n*), this operator is equivalent to `lsl #optional_imm`. Note that `uxtw` is preferred over this form when all registers are word sized (both do the same thing with word-sized registers).
`sxtx #optional_imm`	This operator is applicable only when all the registers are 64 bits. This is effectively the same as `uxtx` (`uxtx` is the preferred form).
`lsl #optional_imm`	If the extend operator is redundant (`uxtx`/`sxtx` for double words, `uxtw`/`sxtw` for word registers), you should use the `lsl` operator for clarity (it is the same operation).

The extension operators are very useful for mixed-sized arithmetic. Chapter 8 discusses this when it covers operating on different-sized operands.

2.20 Large Constants

At several points, this chapter has punted on the solution to dealing with immediate constants that don't fit into 12 or 16 bits. It's time to rectify that omission.

As mentioned, if you need a constant for an arithmetic or logical operation that won't fit within the bits set aside for constants in the instruction's encoding, you'll have to load that constant into a register and operate on the register rather than directly using the constant. The drawback to this scheme is that you'll need at least one additional instruction, and often more, to first load the constant into a temporary register so you can use that value in an arithmetic operation. For example, suppose you want to add the value 40,000 to the X1 register. The following instruction won't work because 40,000 won't fit in 12 bits:

```
add x1, x1, #40000
```

However, since 40,000 will fit in 16 bits, you could do the following:

```
mov x0, #40000  // Works, because mov allows 16-bit consts
add x1, x1, x0  // Add 40000 to X1.
```

Sadly, your program will be a little larger (an extra 4 bytes for the `mov` instruction) and a little slower (executing two instructions rather than one), but it's about as efficient as it's going to get.

What if you want to add a constant that won't fit into 16 bits (perhaps 400,000)? This problem has a couple of solutions. First, as you saw in Chapter 1, a variant of the `ldr` instruction allows you to load any sized constant into a register (32 or 64 bits). That form has the following syntax

```
ldr reg, =largeConstant
```

where *`reg`* is a general-purpose register (32- or 64-bit) and *`largeConstant`* is an immediate value (literal or symbolic) that will fit in the specified register. This instruction form will set aside storage (within the `.text` section, which is read-only) and initialize that storage with the specified constant. When the `ldr` instruction executes, it will load the contents of that memory location into the specified register.

This single instruction is a convenient way to load a large constant into a register. However, this approach has a couple of problems. First, accessing memory on the ARM is a relatively slow process. Second, because Gas inserts the constant into your `.text` section, it could affect the performance of other code in your program; although this is rare and probably not worth worrying about, it's something to keep in mind.

Fortunately, you can load larger constants into a general-purpose register in other ways. These techniques involve additional variants of the `mov` instruction: `movz`, `movk`, and `mvn`.

2.20.1 movz

The movz instruction (move, with zeroing) has the following syntax

```
movz reg_dest, #imm_1
movz reg_dest, #imm_1, lsl #imm_2
```

where reg_{dest} is any general-purpose (32- or 64-bit) register, imm_1 is a 16-bit immediate constant, and imm_2 is one of the four values 0, 16, 32, or 48 (0 is the default value, if the lsl #imm_2 operand is not present).

The movz instruction will take the imm_1 constant and shift it to the left the number of bits specified by the imm_2 constant (with 0s in all the other bit positions, hence the *with zeroing* in the name). It will then move this shifted constant into the destination register. The following three instructions do exactly the same thing, loading the constant 122 into X0:

```
mov  x0, #122
movz x0, #122
movz x0, #122, lsl #0
```

The difference between mov and movz is that mov will sign-extend the immediate constant you supply, whereas movz will zero-extend the constant. For values less than 0x8000, both will load the same constant into the destination register (in fact, the assembler may convert the movz instruction to mov if both would produce the same result). Keep in mind that the shift value can be only 0, 16, 32, or 48; you cannot specify an arbitrary bit-shift value for this instruction.

The movz instruction is useful when you want to load a 16-bit value into the HO half word of a 32-bit register, or one of the three HO half words (1, 2, or 3) of a 64-bit register.

2.20.2 movk

Although the movz instruction allows you to move some values that are larger than 65,535 into a register, it's not a general solution for loading 32- and 64-bit constants into a register. The movk instruction (combined with movz and mov) fulfills that role. The movk instruction (move and keep unaffected bits) has a syntax very similar to movz:

```
movk reg_dest, #imm_1  // Default is "lsl #0"
movk reg_dest, #imm_1, lsl #imm_2
```

The movk instruction will shift the immediate operand by 0, 16, 32, or 48 bits and then merge that value into the destination register. (It does not zero the other bit positions but instead preserves their original values.)

To load a 32-bit immediate constant into the W0 register, use the following instruction sequence:

```
mov  w0, #LO_16_bits
movk w0, #HO_16_bits, lsl #16
```

To load a full 64 bits into X0, use the following:

```
mov  x0, #LO_16_bits
movk x0, #Bits_16_to_31, lsl #16
movk x0, #Bits_32_to_47, lsl #32
movk x0, #HO_16_bits, lsl #48
```

Most of the time, the immediate constant won't require a full 64 bits, so you might be able to get by with two or three instructions rather than the full four. However, you'll never need more than four instructions to load a 64-bit constant into a register (and never more than two to load a 32-bit constant).

2.20.3 movn

The `movn` (move not) instruction is another variant of `mov` that logically negates the immediate constant before loading it into the destination register. The syntax is the same as `movz` (swapping, of course, `movn` for `movz`):

```
movn reg_dest, #imm_1    // Default shift is lsl #0.
movn reg_dest, #imm_1, lsl #imm_2
```

The `movn` instruction shifts the immediate constant by 0, 16, 32, or 48 bits and then inverts the whole (32- or 64-bit) bit string before assigning it to the destination register.

Consider the following example:

```
movn x1, #0xff, lsl 16
```

This instruction loads 0xFFFFFFFFFF00FFFF into the X1 register. (0xFF shifts left 16 bit positions and then inverts all the bits.)

Particularly when loading negative constants into a register, the `movn` instruction can help reduce the number of instructions needed to load a 64-bit constant. However, 32-bit constants, which don't fit into 16 bits, will generally take two instructions no matter what. This differs from the `mvn` instruction in that it allows shifted immediate constants.

2.21 Moving On

This chapter covered basic data types, representation, and operations on those data types. This includes the decimal, binary, and hexadecimal numbering systems, and machine-level data including bits, nibbles, and so on. It discussed logical operations on bits and bit strings, signed and unsigned integer representation and sign and zero extension to expand the number of bits used by a number, as well as sign contraction and saturation to reduce the number of bits used by a number. It also introduced floating-point and BCD data formats and character data (including ASCII and Unicode characters).

This chapter also included information on machine instruction encoding and presented ARM assembly language instructions to load and store memory values, compare and branch instructions for controlling program flow, and shift and rotate instructions. It described packing data into bit fields, the Operand2 formats for constants and other operands, and how to load large constants that won't fit in the 32-bit instruction encoding into a register.

In short, this chapter provided the tools and techniques you need for manipulating various types of constants in assembly language programs. While constants are an important part of any assembly language program, being able to manipulate variable data is the basis of most computer systems. The next chapter discusses the ARM memory subsystem and how to create and efficiently use memory-based variables.

2.22 For More Information

- For general information about data representation and Boolean functions, consider reading my book *Write Great Code*, Volume 1, 2nd edition (No Starch Press, 2020), or a textbook on data structures and algorithms.
- ASCII and Unicode are both International Organization for Standardization (ISO) standards, and ISO provides reports for both character sets. Generally, those reports cost money, but you can also find lots of information about the ASCII and Unicode character sets by searching for them by name on the internet. You can also read about Unicode at *https://www.unicode.org*. Finally, *Write Great Code*, cited previously, contains additional information on the history, use, and encoding of the Unicode character set.
- For more on ARM CPUs, see *https://developer.arm.com*.
- To learn more on the IEEE floating-point single-precision format, see *https://en.wikipedia.org/wiki/Single-precision_floating-point_format*.
- Find out more about the IEEE floating-point double-precision format at *https://en.wikipedia.org/wiki/Double-precision_floating-point_format*.

TEST YOURSELF

1. What does the decimal value 9,384.576 represent (in terms of powers of 10)?
2. Convert the following binary values to decimal:
   a. 1010
   b. 1100
   c. 0111

d. 1001
e. 0011
f. 1111

3. Convert the following binary values to hexadecimal:
   a. 1010
   b. 1110
   c. 1011
   d. 1101
   e. 0010
   f. 1100
   g. 1100_1111
   h. 1001_1000_1101_0001
4. Convert the following hexadecimal values to binary:
   a. 12AF
   b. 9BE7
   c. 4A
   d. 137F
   e. F00D
   f. BEAD
   g. 4938
5. Convert the following hexadecimal values to decimal:
   a. A
   b. B
   c. F
   d. D
   e. E
   f. C
6. How many bits are there in a:
   a. Word
   b. Qword
   c. Half word
   d. Dword
   e. BCD digit
   f. Byte
   g. Nibble

(continued)

7. How many bytes are there in a:
    a. Word
    b. Dword
    c. Qword
    d. Half word
8. How many different values can you represent with a:
    a. Nibble
    b. Byte
    c. Half word
    d. Bit
9. How many bits does it take to represent a hexadecimal digit?
10. How are the bits in a byte numbered?
11. Which bit number is the LO bit of a word?
12. Which bit number is the HO bit of a dword?
13. Compute the AND of the following binary values:
    a. 0 and 0
    b. 0 and 1
    c. 1 and 0
    d. 1 and 1
14. Compute the OR of the following binary values:
    a. 0 or 0
    b. 0 and 1
    c. 1 and 0
    d. 1 and 1
15. Compute the XOR of the following binary values:
    a. 0 and 0
    b. 0 and 1
    c. 1 and 0
    d. 1 and 1
16. The NOT operation is the same as XORing with what value?
17. Which logical operation would you use to force bits to 0 in a bit string?
18. Which logical operation would you use to force bits to 1 in a bit string?
19. Which logical operation would you use to invert all the bits in a bit string?

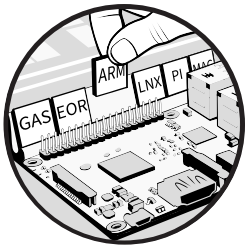
20. Which logical operation would you use to invert selected bits in a bit string?
21. Which machine instruction will invert all the bits in a register?
22. What is the two's complement of the 8-bit value 5 (00000101b)?
23. What is the two's complement of the signed 8-bit value –2 (11111110)?
24. Which of the following signed 8-bit values are negative?
    a. 1111_1111b
    b. 0111_0001b
    c. 1000_0000b
    d. 0000_0000b
    e. 1000_0001b
    f. 0000_0001b
25. Which machine instruction takes the two's complement of a value in a register or memory location?
26. Which of the following 16-bit values can be correctly sign-contracted to 8 bits?
    a. 1111_1111_1111_1111
    b. 1000_0000_0000_0000
    c. 000_0000_0000_0001
    d. 1111_1111_1111_0000
    e. 1111_1111_0000_0000
    f. 0000_1111_0000_1111
    g. 0000_0000_1111_1111
    h. 0000_0001_0000_0000
27. What machine instruction provides the equivalent of an HLL `goto` statement?
28. What is the syntax for a GNU statement label?
29. What flags are the condition codes?
30. Which condition code does `beq` test?
31. Which condition codes does `blo` test?
32. Which conditional branch instructions transfer control based on an unsigned comparison?
33. Which conditional branch instructions transfer control based on a signed comparison?
34. How does the `lsl` instruction affect the zero flag?
35. A shift left is equivalent to what arithmetic operation?

(continued)

36. A shift right is equivalent to what arithmetic operation?
37. When performing a chain of floating-point addition, subtraction, multiplication, and division operations, which operations should you try to do first?
38. What is a normalized floating-point value?
39. How many bits does a (standard) ASCII character require?
40. What is the hexadecimal representation of the ASCII characters 0 through 9?
41. What delimiter character(s) does Gas use to define character constants?

3

MEMORY ACCESS AND ORGANIZATION



Chapters 1 and 2 showed you how to declare and access simple variables in an assembly language program. This chapter fully explains ARM memory access. You'll learn how to efficiently organize your variable declarations to speed up access to their data. You'll also learn about the ARM stack and how to manipulate data on it.

This chapter discusses several important concepts, including the following:

- Memory organization
- Memory access and the memory management unit
- Position-independent executables and address space layout randomization
- Variable storage and data alignment
- Endianness (memory byte order)

- ARM memory addressing modes and address expressions
- Stack operations, return addresses, and preserving register data

This chapter will teach to you make efficient use of your computer's memory resources.

3.1 Runtime Memory Organization

A running program uses memory in many ways, depending on the data's type. Here are some common data classifications you'll find in an assembly language program:

Code Memory values that encode machine instructions (also known as the *text* section under Linux and macOS).

Uninitialized static data An area in memory set aside by the program for uninitialized variables that exist the whole time the program runs; the OS will initialize this storage area to 0s when it loads the program into memory.

Initialized static data A section of memory that also exists the whole time the program runs. However, the OS loads values for all the variables appearing in this section from the program's executable file, so they have an initial value when the program first begins execution.

Read-only data Similar to initialized static data, insofar as the OS loads initial data for this section of memory from the executable file. However, this section is marked *read-only* to prevent inadvertent modification of the data. Programs typically place constants and other unchanging data in this section (the code section is also marked read-only by the OS).

Heap This special section of memory is designated to hold dynamically allocated storage. Functions such as C's `malloc()` and `free()` are responsible for allocating and deallocating storage in the heap area. Section 4.4.4, "Pointer Variables and Dynamic Memory Allocation," on page 178 discusses dynamic storage allocation in greater detail.

Stack In this special section in memory, the program maintains local variables for procedures and functions, program state information, and other transient data. See section 3.9, "The Push and Pop Operations," on page 155 for more information about the stack section.

These are the typical sections you will find in common programs, assembly language or otherwise. Smaller programs won't use all these sections, though most programs have at least code, stack, and data sections. Complex programs may create additional sections in memory for their own purposes. Some programs may combine several of these sections. For example, many programs will combine the code and read-only sections into the same section in memory (as the data in both sections gets marked as read-only). Some programs combine the uninitialized and initialized data

sections, initializing the uninitialized variables to 0. Combining sections is generally handled by the linker program. See section 3.12, "For More Information," on page 167 concerning the GNU linker.

Linux and macOS tend to put different types of data into different sections (or *segments*) of memory. Although it is possible to reconfigure memory to your choice by running the linker and specifying various parameters, one typical organization might be similar to that in Figure 3-1.

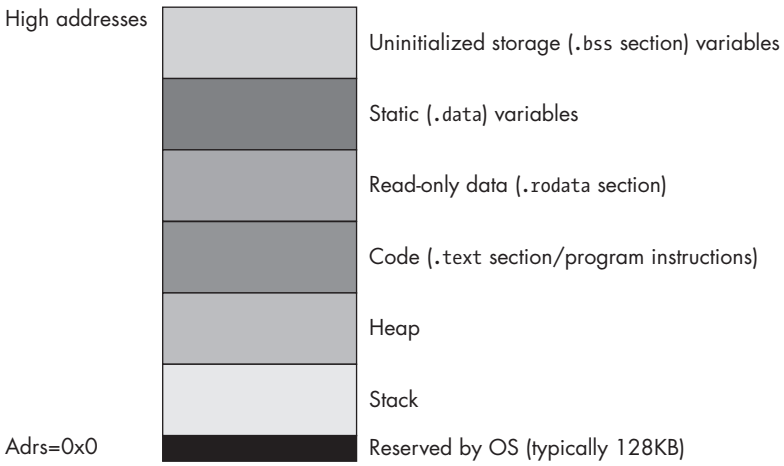


Figure 3-1: A Linux/macOS example runtime memory organization

This figure is just an example. Real programs will likely organize memory differently, especially when using address space layout randomization, discussed later in this chapter.

The OS reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at these low addresses. One reason the OS reserves this space is to help trap NULL pointer references: if you attempt to access memory location 0x0 (NULL), the OS will generate a *segmentation fault* (also known as a *general protection fault*), meaning you've accessed a memory location that doesn't contain valid data.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the `.text` (code) section, the `.data` section, the `.rodata` (read-only data) section, and the `.bss` (uninitialized data) section. Each of these memory sections corresponds to a type of data you can create in your Gas programs. I will describe the `.text`, `.data`, `.rodata`, and `.bss` sections in detail next. (The OS provides the stack and heap sections; you don't normally declare these two in an assembly language program, so there isn't anything more to discuss about them here.)

3.1.1 The .text Section

The `.text` section contains the machine instructions that appear in a Gas program. Gas translates each machine instruction you write into a

sequence of one or more word values. The CPU interprets these 32-bit word values as machine instructions during program execution.

By default, when GCC/Gas/*ld* links your program, it tells the system that your program can execute instructions and read data from the code segment, but cannot write data to the code segment. The OS will generate a segmentation fault if you attempt to store any data into the code segment.

3.1.2 The .data Section

You'll typically put your variables in the `.data` section. In addition to declaring static variables, you can embed lists of data into the `.data` declaration section. You use the same technique to embed data into your `.data` section that you use to embed data into the `.text` section: use the `.byte`, `.hword`, `.word`, `.dword`, and so on, directives. Consider the following example:

```
     .data
bb:  .byte    0
     .byte    1,2,3

u:   .word    1
     .dword   5,2,10

c:   .byte    0
     .byte    'a', 'b', 'c', 'd', 'e', 'f'

bn:  .byte    0
     .byte    true  // Assumes true is defined as 1
```

Values that Gas places in the `.data` memory segment by using these directives are written to the segment after the preceding variables. For example, the byte values `1,2,3` are emitted to the `.data` section after `bb`'s `0` byte. Because there aren't any labels associated with these values, you do not have symbolic access to these values in your program. You can use the indexed addressing modes (described later in this chapter) to access these extra values.

3.1.3 Read-Only Data Sections

Gas does not provide a stand-alone directive for creating sections that hold read-only constants. However, you can easily use the Gas `.section` directive to create a generic read-only constant section as follows:

```
.section .rodata, ""
```

Most programs use the `.rodata` identifier, by convention, for read-only data. For example, GCC uses this name for read-only constant sections. You could use any identifier you choose here. For example, I often use the name `.const` for constant sections. However, as GCC uses `.rodata`, I'll stick to that convention in this book. I'll say more about the `.section` directive a little later; for the time being, note that as long as the second argument is the empty string, Gas will create a read-only data section by using this directive.

The `.section .rodata` section holds constants, tables, and other data that your program cannot change during execution. This section is similar to the `.data` section, with two differences:

- The `.rodata` section is defined with `.section .rodata, ""` rather than `.data`.
- The system does not allow you to write data to variables in an `.rodata` object while the program is running.

Here's an example:

```
        .section  .rodata, ""
pi:     .single   3.141592653589793 // (rounded)
e:      .single   2.718281828459045 // (rounded)
MaxU16: .hword    65535
MaxI16: .hword    32767
```

For many purposes, you can treat `.rodata` objects as literal constants. However, because they are actually memory objects, they behave like read-only `.data` objects. You cannot use an `.rodata` object anywhere a literal constant is allowed. For example, you cannot use them as *displacements* (constant *offsets* from a base pointer) in addressing modes (see section 3.6, "The ARM Memory Addressing Modes," on page 140), in constant expressions, or as immediate values. In practice, you can use them anywhere that reading a `.data` variable is legal.

LINUX VS. MACOS: FORCED CODE ALIGNMENT

ARM machine instructions must be aligned on a word (32-bit) boundary. The ARM cannot physically address an instruction that is not so aligned. Therefore, if you insert data into the `.text` section that is not a multiple of 4 bytes long, any instructions following that data will be misaligned. You must always include an `.align 2` (or `.balign 4`) directive before any code appearing after data that is not a multiple of 4 bytes long in the `.text` section.

The macOS assembler is so paranoid about this that it requires all symbols appearing in the `.text` section to be aligned on a 4-byte boundary, and it will generate an error if it encounters a label declaration (*`label`*`:`, where *`label`* represents any identifier) that is not associated with a word-aligned address. The only way to correct this error is to insert an `.align 2` (or `.balign 4`) directive before the label declaration. This can create a problem for certain data declarations in the `.text` section. Consider the following code:

```
      .align 2
bb:   .byte  0
c:    .byte  0
```

(continued)

The macOS assembler will require both of these symbols to be word-aligned (requiring an `.align 2` directive between them), even if you don't want this. You might, for example, want `c` to immediately follow `bb` in memory. The macOS assembler does not allow this. If you define a label, that label must be aligned on a word boundary.

One solution is to avoid putting data in the `.text` section; just put your read-only constants, such as `.rodata`, in their own section. However, there are good reasons for wanting to put data in the `.text` section. In those situations, you'll have to work around this limitation when writing code for macOS.

As with the `.data` section, you may embed data values in the `.rodata` section by using the `.byte`, `.hword`, `.word`, `.dword`, and so on, data declarations. For example:

```
        .section  .rodata, ""
roArray: .byte    0
        .byte     1, 2, 3, 4, 5
dwVal:   .dword   1
        .dword    0
```

You can also declare constant values in the `.text` section. Data values you declare in this section are also read-only objects, as Linux and macOS write-protect the `.text` section. If you do place constant declarations in a `.text` section, take care to place them in a location that the program will not attempt to execute as code (such as after a `b.al` or `ret` instruction). Unless you're using data declarations to manually encode ARM machine instructions (which would be rare and done only by expert programmers), you don't want your program to attempt to execute data as machine instructions; the result is usually undefined.

NOTE *Technically, the result of executing data in the `.text` section is well defined: the machine will decode whatever bit pattern you place in memory as a machine instruction. However, few people will be able to look at a piece of data and interpret its meaning as a machine instruction.*

3.1.4 The .bss Section

The `.data` section requires that you initialize objects, even if you simply place a default value of `0` in the operand field. The `.bss` (block started by symbol) section lets you declare variables that are always uninitialized when the program begins running. This section begins with the `.bss` reserved word and contains variable declarations whose initializers must always be 0. Here is an example:

```
            .bss
UninitUns32: .word  0
i:          .word  0
character:  .byte  0
bb:         .byte  0
```

The OS will initialize all `.bss` objects to 0 when it loads your program into memory. However, it's probably not a good idea to depend on this implicit initialization. If you need an object initialized with 0, declare it in a `.data` section and explicitly set it to 0.

Annoyingly, Gas requires you to explicitly provide an initializer of 0 when declaring variables in the `.bss` section. Good assembly language programmers don't like doing this, because providing their source code with an explicit value tells the reader that they are expecting that variable to contain that value when the program runs. If the program explicitly isn't expecting the variable to be initialized, it would be nice to tell the reader that.

A very old convention to make this statement is to use the expression `.-.` in the operand field of such declarations. For example:

```
            .bss
UninitUns32: .word  .-.
i:          .word  .-.
character:  .byte  .-.
bb:         .byte  .-.
```

Gas substitutes the current value of the location counter (see section 3.2, "Gas Storage Allocation for Variables," on page 131) in place of the period (`.`). The expression *`location_counter`* minus *`location_counter`* is equal to 0, which satisfies the Gas requirements for initializers in the `.bss` section. This strange syntax lets the reader know that you're not explicitly expecting the variable to be initialized with 0 when the program runs.

If `.-.` is too bizarre for your tastes (or you don't want to have to type three characters), I've often used something like this to get the same results:

```
            .equ   _, 0  // "_" is a legitimate identifier
            .bss
UninitUns32: .word  _
i:          .word  _
character:  .byte  _
bb:         .byte  _
```

This book tends to use the `.-.` form (when not explicitly specifying `0`), as there is historical precedence for it. This form has one drawback, however: it does not work for `.qword` declarations (this is a Gas limitation).

Variables you declare in the `.bss` section may consume less disk space in the executable file for the program. This is because Gas writes out initial values for `.rodata` and `.data` objects to the executable file, but it may use a

compact representation for uninitialized variables you declare in the .bss section. Note, however, that this behavior is dependent on the OS version and object-module format.

3.1.5 The .section Directive

The .section directive allows you to create sections using any name you please (the .rodata section is an example). The syntax for this directive is

```
.section identifier, flags
```

where *identifier* is any legal Gas identifier (it does not have to begin with a period) and *flags* is a string surrounded by quotes. The contents of the string vary by OS, but both Linux and macOS seem to support the following characters:

- **b** Section is a .bss section and will hold uninitialized data. All data declarations must have a 0 initializer.
- **x** Section contains executable code.
- **w** Section contains writable data.
- **a** Section is allocatable (must be present for data sections).
- **d** Section is a data section.

The *flags* string may contain zero or more of these characters, though certain flags (such as "b" and "x" or "d") are mutually exclusive. If the "w" flag is not present in the string, the section will be read-only. Here are some typical .section declarations:

```
.section aDataSection, "adw" // Typical data section
.section .const, ""          // Like .rodata
.section .code, "x"          // Code section (like .text)
```

Each unique section you define will be given its own block of memory (such as the blocks that appear in Figure 3-1). The GNU linker/loader will merge all sections with the same name when assigning them to blocks of memory.

3.1.6 Declaration Sections

The .data, .rodata, .bss, .text, and other named sections may appear zero or more times in your program. The declaration sections may appear in any order, as the following example demonstrates:

```
            .data
i_static:   .word     0

            .bss
i_uninit:   .word     .-.
```

```
            .section  .rodata, ""
i_readonly: .word     5

            .data
j:          .word     0

            .section  .rodata, ""
i2:         .word     9

            .bss
c:          .byte     .-.

            .bss
d:          .word     .-.

            .text

Code goes here.
```

The sections may appear in an arbitrary order, and a given declaration section may appear more than once in your program. As noted previously, when multiple declaration sections of the same type (for example, the three `.bss` sections in the preceding example) appear in a declaration section of your program, Gas combines them into a single group, in any order it pleases.

3.1.7 Memory Access and MMU Pages

The ARM's *memory management unit (MMU)* divides memory into blocks known as *pages*. The OS is responsible for managing pages in memory, so application programs don't typically worry about page organization. However, when working with pages in memory, make sure you're aware of whether the CPU even allows access to a given memory location and whether it is read/write or read-only (write-protected).

Each program section appears in memory in contiguous MMU pages. That is, the `.rodata` section begins at offset 0 in an MMU page and sequentially consumes pages in memory for all the data appearing in that section. The next section in memory (perhaps `.data`) begins at offset 0 in the next MMU page following the last page of the previous section. If that previous section (for example, `.rodata`) does not consume an integral multiple of 4,096 bytes, padding space will be present between the end of that section's data and the end of its last page, to guarantee that the next section begins on an MMU page boundary.

Each new section starts in its own MMU page because the MMU controls access to memory by using page *granularity*. For example, the MMU controls whether a page in memory is readable/writable or read-only. For `.rodata` sections, you want the memory to be read-only. For the `.data` section, you want to allow reads and writes. Because the MMU can enforce these attributes only on a page-by-page basis, you cannot have `.data` section information in the same MMU page as an `.rodata` section.

Normally, all this is completely transparent to your code. Data you declare in a `.data` (or `.bss`) section is readable and writable, and data in an `.rodata` or `.text` section is read-only (`.text` sections are also executable). Beyond placing data in a particular section, you don't have to worry too much about the page attributes.

You do need to worry about MMU page organization in memory in one situation. Sometimes it is convenient to access (read) data beyond the end of a data structure in memory. However, if that data structure is aligned with the end of an MMU page, accessing the next page in memory could be problematic. Some pages in memory are *inaccessible*; the MMU does not allow reading, writing, or execution to occur on that page. Attempting to do so will generate an ARM *segmentation fault*. This will typically crash your program, unless you have an exception handler in place to handle segmentation faults. If you have a data access that crosses a page boundary, and the next page in memory is inaccessible, this will crash your program. For example, consider a half-word access to a byte object at the very end of an MMU page, as shown in Figure 3-2.

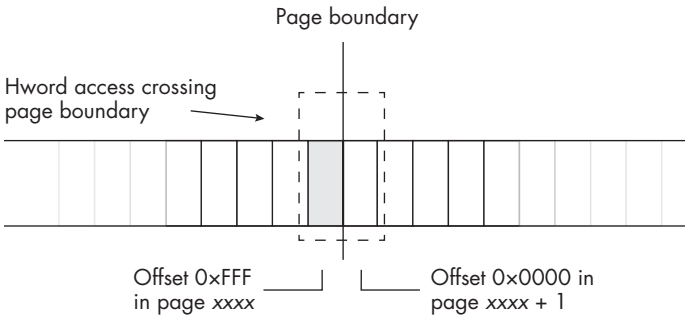


Figure 3-2: Half-word access at the end of a memory-management page

As a general rule, you should never read data beyond the end of a data structure. If for some reason you need to do so, ensure that it is legal to access the next page in memory. It goes without saying that you should never write data beyond the end of a given data structure; this is always incorrect and can create far more problems than just crashing your program (including severe security issues).

3.1.8 PIE and ASLR

As noted in Chapter 1, macOS forces all code to use a position-independent executables (PIE) form. Linux doesn't absolutely require this, but it allows you to write PIE code if you choose. There are two main reasons for PIE code: shared libraries and security, which were covered in "Linux vs. macOS: Position-Independent Executables" on page 23. However, as the behavior of PIE code profoundly affects the way you write ARM assembly language, it is worthwhile to spend a little more time discussing PIE, and especially *address space layout randomization (ASLR)*.

ASLR is an attempt by the OS to thwart various exploits (hacks) that try to figure out where the code and data reside in an application. Prior to PIE and ASLR, most OSes always loaded the executable code and data to the same address in memory, making it easy for a hacker to patch or otherwise mess with the executable program. By loading the code and data sections into random memory locations, PIE/ASLR make it much more difficult for exploits to tap into the executing code.

As a result of ASLR, the layout of an executing program in memory will not actually look like that in Figure 3-1. For one given instance of a program execution, it might look something like Figure 3-3.

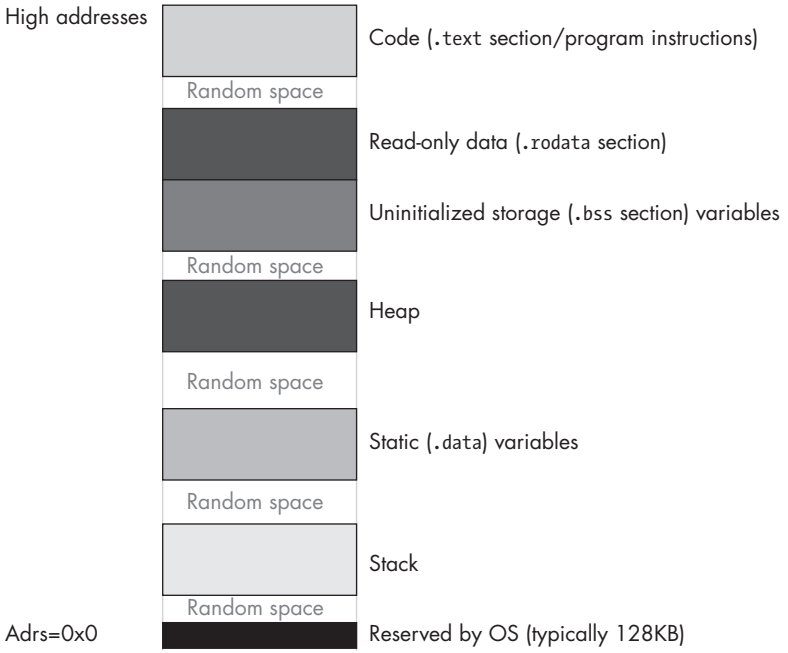


Figure 3-3: A possible memory layout for one execution of an application

However, on the next run of the program, the sections will likely be rearranged and placed at different locations in memory.

While PIE/ASLR makes it difficult for hackers to exploit your code, it also plays havoc with the ARM's instruction set. Consider the following (legitimate) ARM `ldr` instruction:

```
ldr w0, someWordVar  // Assume someWordVar is in .data
```

This would normally load the W0 register from the 32-bit variable `someWordVar` found in the `.data` section. This particular instruction uses the *PC-relative addressing mode*, which means that the instruction encodes an offset from the address of the `ldr` instruction to the `someWordVar` variable in

memory. However, if you assemble this program under macOS, you get the following error:

```
error: unknown AArch64 fixup kind!
```

Under Linux (Ubuntu and Raspberry Pi OS seem to be different; your mileage may vary), you get something like

```
relocation truncated to fit: R_AARCH64_LD_PREL_LO19 against `.data'
```

This is a real ARM64 instruction and *should* work. In fact

```
ldr reg, =constant
```

is just a special form of this instruction, and it does work.

The problem is due to the ARM 32-bit instruction length. If you look up the encoding for the `ldr` instruction in the ARM reference manual, you'll discover that it sets aside 19 bits for the address of the memory location. This turns out to be an *offset* (a distance in bytes) from the address of the `ldr` instruction (that is, the value of the 19-bit field is added to the PC to get the actual memory address). Because it's referencing data in the `.text` section, and everything is word-aligned in the text section, the 19-bit offset is actually a word offset, not a byte offset. This effectively gives the `ldr` instruction another 2 bits (the LO 2 bits will always be 0). This effective 21-bit offset allows the `ldr` instruction to access data at a location ±1MB around the `ldr` instruction.

Unfortunately, when accessing data in the `.data` section, which the OS has been nice enough to place at a random address (probably farther than 1MB away), the 21-bit range of the `ldr` instruction won't be sufficient. This is why Gas complains about attempting to access a variable in the `.data` section with the `ldr` instruction. As a bottom line, you can't use that instruction to directly access data unless that data is also in the `.text` section and isn't more than ±1MB away.

3.1.9 The .pool Section

The `.pool` section is a Gas pseudo-section in your program. As noted previously, the following instruction loads a large constant into a register by placing that constant somewhere in memory, then loading the contents of that memory location into the destination register:

```
ldr reg, =largeConstant
```

In other words, this instruction is completely equivalent to either of the following:

```
    ldr x0, a64_bit_constant
    ldr w0, a32_bit_constant
     .
     .
     .
```

```
// Somewhere in the .text section that will never
// be executed as code:

a64_bit_constant: .dword  The_Actual_64bit_Constant_Value
a32_bit_constant: .word   The_Actual_32bit_Constant_Value
```

Gas automatically figures out an appropriate place to put such constants: near the instructions that reference them but out of the code path.

If you'd like to control the placement of these constants in your `.text` section, you can use the `.pool` directive. Wherever you place this directive in your `.text` section (and it must be in the `.text` section), Gas will emit the constants it produces. Just make sure that if you put a `.pool` directive in your code, you place it after an unconditional branch or return instruction so that the program flow won't attempt to execute that data as machine instructions.

Normally, you don't need to place a `.pool` directive in your source code, since Gas will do a reasonable job of finding a location to place its data. However, if you intend to also insert data of your own in the `.text` section, you may want to insert the `.pool` directive and place your data declarations immediately afterward. Note that the data after `.pool` is part of the `.text` section, so you can continue to place machine instructions after the `.pool`.

3.2 Gas Storage Allocation for Variables

Gas associates a current *location counter* with each of the declaration sections (`.text`, `.data`, `.rodata`, `.bss`, and any other named sections). These location counters initially contain `0`. Whenever you declare a variable in one of these sections (or write code in a code section), Gas associates the current value of that section's location counter with the label and bumps up the value of that location counter by the size of the object you're declaring.

For example, assume that the following is the only `.data` declaration section in a program:

```
    .data
bb: .byte  0        // Location counter = 0, size = 1
s:  .hword 0        // Location counter = 1, size = 2
w:  .word  0        // Location counter = 3, size = 4
d:  .dword 0        // Location counter = 7, size = 8
q:  .qword 0        // Location counter = 15, size = 16
                    // Location counter is now 31.
```

Variable declarations listed in a single `.data` section have contiguous offsets (location counter values) into the `.data` section. Given the preceding declaration, `s` will immediately follow `bb` in memory, `w` will immediately follow `s` in memory, `d` will immediately follow `w`, and so on. These offsets aren't the actual runtime addresses of the variables. At runtime, the system loads each section to a base address in memory. The linker and the OS add the base address of the memory section to each of these location counter values

(normally called *displacements*, or *offsets*) to produce the actual memory address of the variables.

OBTAINING THE CURRENT LOCATION COUNTER VALUE

If you ever want to use the current location counter value in your program, Gas will substitute it for a single period (`.`) wherever a constant is allowed, as in the following example:

```
.dword . // Stores the address of this dword in memory
```

You'd normally use the `.` operator to compute lengths of sections of code, using something like the following:

```
lbl:  .byte 0, 1, 2, 3, 4
lbl2: .hword 55
size: .word . - lbl
```

The `. - lbl` expression computes the number of bytes between the `lbl` symbol and the `size` label. The `.` operator returns the location counter value at the beginning of the `.word` directive and does not include the 4 bytes that `.word` will emit to the output file.

Keep in mind that you may link other modules with your program (for example, from the C stdlib) or even additional `.data` sections in the same source file, and the linker has to merge the `.data` sections. Each individual section (even when it has the same name as another section) has its own location counter that starts from 0 when allocating storage for the variables in the section. Hence, the offset of an individual variable may have little bearing on its final memory address.

Gas allocates memory objects you declare in `.rodata`, `.data`, and `.bss` sections in completely different regions of memory. Therefore, you cannot assume that the following three memory objects appear in adjacent memory locations (indeed, they probably will not):

```
    .data
bb: .byte    0

    .section .rodata, ""
w:  .word    0x1234

    .bss
d:  .dword   .-.
```

In fact, Gas will not even guarantee that variables you declare in separate `.data` (or other) sections are adjacent in memory, even if there is nothing between the declarations in your code. For example, you cannot assume

whether `bb`, `w`, and `d` are—or aren't—in adjacent memory locations in the following declarations:

```
    .data
bb: .byte   0

    .data
w:  .word   0x1234

    .data
d:  .dword  0
```

If your code requires these variables to consume adjacent memory locations, you must declare them in the same `.data` section.

3.3 Little-Endian and Big-Endian Data Organization

As you learned in section 1.6.2, "The Memory Subsystem," on page 14, the ARM stores multibyte data types in memory, with the LO byte at the lowest address in memory and the HO byte at the highest address (see Figure 1-6). This type of data organization in memory is known as *little endian*. Little-endian data organization, in which the LO byte comes first and the HO byte comes last, is common in many modern CPUs. It is not, however, the only possible approach.

Big-endian data organization reverses the order of the bytes in memory. The HO byte of the data structure appears first, in the lowest memory address, and the LO byte appears in the highest memory address. Table 3-1 describes the memory organization for half words.

Table 3-1: Half-Word Object Memory Organization

Data byte	Little endian	Big endian
0 (LO byte)	base + 0	base + 1
1 (HO byte)	base + 1	base + 0

Table 3-2 describes the memory organization for words.

Table 3-2: Word Object Memory Organization

Data byte	Little endian	Big endian
0 (LO byte)	base + 0	base + 3
1	base + 1	base + 2
2	base + 2	base + 1
3 (HO byte)	base + 3	base + 0

Table 3-3 describe the memory organization for double words.

Table 3-3: Dword Object Memory Organization

Data byte	Little endian	Big endian
0 (LO byte)	base + 0	base + 7
1	base + 1	base + 6
2	base + 2	base + 5
3	base + 3	base + 4
4	base + 4	base + 3
5	base + 5	base + 2
6	base + 6	base + 1
7 (HO byte)	base + 7	base + 0

Normally, you wouldn't be too concerned with big-endian memory organization on an ARM CPU. However, on occasion, you may need to deal with data produced by a different CPU (or by a protocol, such as Transmission Control Protocol/Internet Protocol, or TCP/IP) that uses big-endian organization as its canonical integer format. If you were to load a big-endian value in memory into a CPU register, the value would be incorrect.

If you have a 16-bit big-endian value in memory and you load it into a register, its bytes will be swapped. For 16-bit values, you can correct this issue by using the `rev16` instruction, which has the following syntax:

```
rev16 reg_dest, reg_src
```

Here, reg_{dest} and reg_{src} are any 32- or 64-bit general-purpose registers (both must be the same size). This instruction will swap the 2 bytes in each of the 16-bit half-words in the source register; that is, this operates on `hword0` and `hword1` in a 32-bit register and on `hword0`, `hword1`, `hword2`, and `hword3` in a 64-bit register. For example

```
ldr     w1, =0x12345678
rev16   w1, w1
```

will produce 0x34127856 in the W1 register, having swapped bytes 0 and 1 as well as bytes 2 and 3.

If you have a 32-bit value in a register (32- or 64-bit), you can swap the 4 bytes in that register by using the `rev32` instruction:

```
rev32 reg_dest, reg_src
```

Again, the registers can be 32- or 64-bit, but both must be the same size. In a 32-bit register, this will swap bytes 0 and 3 as well as 1 and 2. In a

64-bit register, it will swap bytes 0 and 3, 1 and 2, 7 and 4, and 6 and 5 (see Figure 3-4).

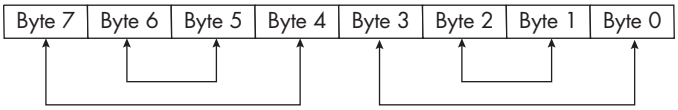


Figure 3-4: Operation of the `rev32` instruction

The `rev` instruction will swap bytes 7 and 0, 6 and 1, 5 and 2, and 4 and 3 in a 64-bit register (see Figure 3-5).

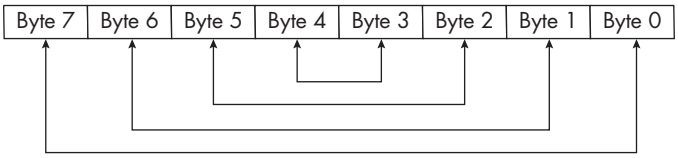


Figure 3-5: Operation of the `rev` instruction

The `rev` instruction accepts only 64-bit registers.

3.4 Memory Access

Section 1.6.2, "The Memory Subsystem," on page 14 describes how the ARM CPU fetches data from memory on the data bus. In an idealized CPU, the data bus is the size of the standard integer registers on the CPU; therefore, you would expect the ARM CPUs to have a 64-bit data bus. In practice, modern CPUs often make the physical data bus connection to main memory much larger in order to improve system performance. The bus brings in large chunks of data from memory in a single operation and places that data in the CPU's *cache*, which acts as a buffer between the CPU and physical memory.

From the CPU's point of view, the cache *is* memory. Therefore, when the remainder of this section discusses memory, it's generally talking about data sitting in the cache. As the system transparently maps memory accesses into the cache, we can discuss memory as though the cache were not present and discuss the advantages of the cache as necessary.

On early processors predating the ARM, memory was arranged as an array of bytes (8-bit machines, such as the Intel 8088), half words (16-bit machines, such as the Intel 8086 and 80286), or words (32-bit machines, such as the 32-bit ARM CPUs). On a 16-bit machine, the LO bit of the address did not physically appear on the address bus. This means the addresses 126 and 127 put the same bit pattern on the address bus (126, with an implicit 0 in bit position 0), as shown in Figure 3-6.

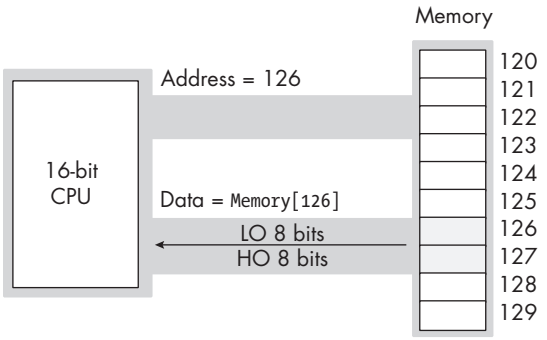


Figure 3-6: The address and data bus for 16-bit processors

When reading a byte, the CPU uses the LO bit of the address to select the LO byte or HO byte on the data bus. Figure 3-7 shows the process when accessing a byte at an even address (126 in this figure).

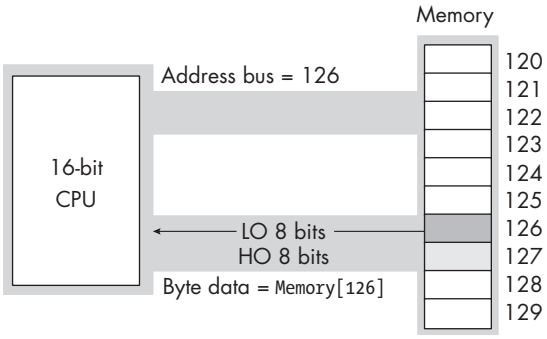


Figure 3-7: Reading a byte from an even address on a 16-bit CPU

Figure 3-8 shows memory access for the byte at an odd address (127 in this figure). Note that in both Figures 3-7 and 3-8, the address appearing on the address bus is 126.

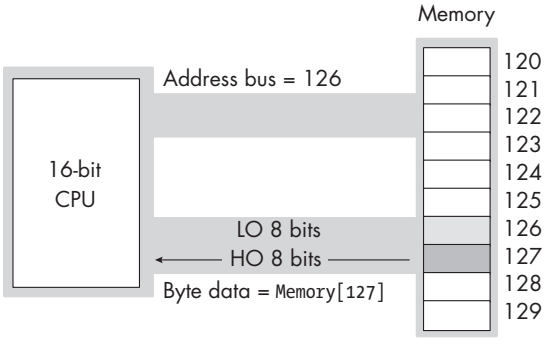


Figure 3-8: Reading a byte from an odd address on a 16-bit CPU

What happens when this 16-bit CPU wants to access 16 bits of data at an odd address? For example, suppose that in these figures, the CPU reads the word at address 125. When the CPU puts address 125 on the address bus, the LO bit doesn't physically appear. Therefore, the actual address on the bus is 124. If the CPU were to read the LO 8 bits off the data bus at this point, it would get the data at address 124, not address 125.

Fortunately, the CPU is smart enough to figure out what's going on here: it extracts the data from the HO 8 bits on the data bus and uses this as the LO 8 bits of the data operand. However, the HO 8 bits that the CPU needs are not found on the data bus. The CPU has to initiate a second read operation, placing address 126 on the address bus, to get the HO 8 bits (these will be sitting in the LO 8 bits of the data bus, but the CPU can figure that out). It takes two memory cycles for this read operation to complete. Therefore, the instruction reading the data from memory will take longer to execute than it would have if the data had been read from an address that was an integral multiple of 2 (16-bit alignment).

The same problem exists on 32-bit processors, except that the 32-bit data bus allows the CPU to read 4 bytes at a time. Reading a 32-bit value at an address that is not an integral multiple of 4 incurs the same performance penalty. However, accessing a 16-bit operand at an odd address doesn't always guarantee an extra memory cycle—only addresses that, when divided by 4, have a remainder of 3 incur the penalty. In particular, if you access a 16-bit value (on a 32-bit bus) at an address where the LO 2 bits contain 0b01, the CPU can read the word in a single memory cycle, as shown in Figure 3-9.

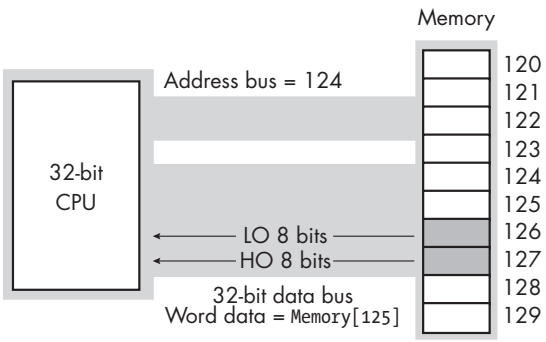


Figure 3-9: Accessing a word on a 32-bit data bus

Modern ARM CPUs with cache systems have largely eliminated this problem. As long as the data (1, 2, 4, or 8 bytes in size) is fully within a *cache line*—a processor-defined number of bytes—no memory cycle penalty occurs for an unaligned access. If the access does cross a cache-line boundary, the CPU will run a little slower while it executes two memory operations to get (or store) the data.

3.5 Gas Support for Data Alignment

To write fast programs, you must ensure that you properly align data objects in memory. Proper *alignment* means that the starting address for an object is a multiple of a certain size—usually the size of an object, if the object's size is a power of 2 for values up to 32 bytes in length. For objects greater than 32 bytes, aligning the object on an 8-, 16-, or 32-byte address boundary is probably sufficient. For objects fewer than 16 bytes, aligning the object at an address that is the next power of 2 greater than or equal to the object's size is usually fine.

As noted in the previous section, accessing data that is not aligned at an appropriate address may require extra time. Therefore, if you want to ensure that your program runs as rapidly as possible, you should try to align data objects according to their size.

Data becomes misaligned whenever you allocate storage for different-sized objects in adjacent memory locations. For example, if you declare a byte variable, it will consume 1 byte of storage, and the next variable you declare in that declaration section will have the address of that byte object plus 1. If the byte variable's address happens to be an even address, the variable following that byte will start at an odd address. If that following variable is a half-word, word, or dword object, its starting address will not be optimal.

In this section, we'll explore ways to ensure that a variable is aligned at an appropriate starting address based on its size. Consider the following Gas variable declarations:

```
    .data
w:  .word  0
bb: .byte  0
s:  .hword 0
w2: .word  0
s2: .hword 0
b2: .byte  0
dw: .dword 0
```

The first `.data` declaration in a program places its variables at an address that is an even multiple of 4,096 bytes. Whatever variable first appears in that `.data` declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables, plus the starting address of that `.data` section.

Therefore, assuming Gas allocates the variables in the previous example at a starting address of `4096`, it will allocate them at the following addresses:

```
                    // Start Adrs       Length
w:    .word   0     //    4096            4
bb:   .byte   0     //    4100            1
s:    .hword  0     //    4101            2
w2:   .word   0     //    4103            4
s2:   .hword  0     //    4107            2
b2:   .byte   0     //    4109            1
dw:   .dword  0     //    4110            8
```

With the exception of the first variable (which is aligned on a 4KB boundary) and the byte variables (whose alignment doesn't matter), all these variables are misaligned. The `s`, `s2`, and `w2` variables start at odd addresses, and the `dw` variable is aligned on an even address that is not a multiple of 8 (word-aligned but not dword-aligned).

An easy way to guarantee that your variables are aligned properly is to put all the dword variables first, the word variables second, the half-word variables third, and the byte variables last in the declaration, as shown here:

```
      .data
dw:   .dword  0
w:    .word   0
w2:   .word   0
s:    .hword  0
s2:   .hword  0
bb:   .byte   0
b2:   .byte   0
```

This organization produces the following addresses in memory:

```
                    //  Start Adrs     Length
dw:    .dword  0    //     4096          8
w2:    .word   0    //     4104          4
w3:    .word   0    //     4108          4
s:     .hword  0    //     4112          2
s2:    .hword  0    //     4114          2
bb:    .byte   0    //     4116          1
b2:    .byte   0    //     4117          1
```

These variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While many technical reasons make this alignment impossible, a good practical reason for not doing this is that it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another, regardless of their size).

To resolve this problem, Gas provides the `.align` and `.balign` directives. As noted in section 1.2, "The Anatomy of an Assembly Language Program," on page 5, the `.align` argument is a value that will be raised to that power of 2, and the `.balign`'s operand is an integer that must be a power of 2 (1, 2, 4, 8, 16, and so on). These directives ensure that the next memory object will be aligned to the specified size.

By default, these directives will pad the data bytes they skip with 0s; in a `.text` section, Gas aligns the code by using `nop` (no-operation) instructions. If you would like to use a different padding value, these two directives allow a second operand:

```
.align  pwr2Alignment, padValue
.balign alignment, padValue
```

Here, `padValue` must be an 8-bit constant, which these directives will use as the padding value. Gas also allows a third argument, which is the maximum allowable padding; see the Gas documentation for more details.

The previous example could be rewritten, using the `.align` directive, as follows:

```
     .data
     .align  2   // Align on 4-byte boundary.
w:   .word   0
bb:  .byte   0
     .align  1   // Align on 2-byte boundary.
s:   .hword  0
     .align  2   // Align on 4-byte boundary.
w2:  .word   0
s2:  .hword  0
b2:  .byte   0
     .align  3   // Align on 8-byte boundary.
dw:  .dword  0
```

If Gas determines that an `.align` directive's current address (location counter value) is not an integral multiple of the specified value, Gas will quietly emit extra bytes of padding after the previous variable declaration until the current address in the `.data` section is a multiple of the specified value. This makes your data larger by a few bytes, in exchange for faster access to it. Since your data will grow only slightly larger when you use this feature, this is probably a good trade-off.

As a general rule, if you want the fastest possible access, choose an alignment value equal to the size of the object you want to align. That is, align half words to even boundaries with an `.align 1` statement, words to 4-byte boundaries with `.align 2`, double words to 8-byte boundaries with `.align 3`, and so on. If the object's size is not a power of 2, align it to the next higher power of 2.

Data alignment isn't always necessary, since the cache architecture of modern ARM CPUs handles most misaligned data. Use the alignment directives only with variables for which speedy access is absolutely critical.

3.6 The ARM Memory Addressing Modes

For the most part, the ARM uses a very standard RISC *load/store architecture*. This means that it accomplishes almost all memory access by using instructions that load registers from memory or store the value held in registers to memory. The load and store instructions access memory by using memory *addressing modes*, mechanisms the CPU uses to determine the address of a memory location. The ARM memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, structs, pointers, and other complex data types. Mastering ARM addressing modes is an important step toward mastering ARM assembly language.

In addition to loads and stores, ARM uses *atomic instructions*. For the most part, these are variations of the load and store instructions, with a few

extra bells and whistles needed for multiprocessing applications. Atomic instructions are beyond the scope of this text; for more information, see the ARM V8 reference manual.

Until now, this book has presented only two mechanisms for accessing memory: the register-indirect addressing mode (for example, `[X0]`) introduced in Chapter 1, and the PC-relative addressing mode discussed in section 3.1.8, "PIE and ASLR," on page 128. However, the ARM provides more than half a dozen modes (depending on how you count them) for accessing data in memory. The following sections describe each of these modes.

3.6.1 PC-Relative

The PC-relative addressing mode is useful only for fetching values from the `.text` section, as the other sections will likely fall out of the ±1MB range of this addressing mode. Therefore, it is much easier to directly access constant data in the `.text` section than it would be in the `.rodata` section (or another read-only section).

A couple of issues arise when using the PC-relative addressing mode in the `.text` section. First, because the 19-bit offset buried in the 32-bit instruction encoding is shifted left 2 bits to produce a word offset (as discussed earlier), you can load only word and double-word values when using this addressing mode—no bytes or half words. For example, you can access byte and half-word values in the `.text` section with the register-indirect addressing mode, but not with the PC-relative addressing mode.

When accessing data in the `.text` section by using the PC-relative addressing mode, keep the following points in mind:

- Under macOS, all labels in the `.text` section must be aligned on a 4-byte boundary, even if the data associated with that label doesn't require such alignment (such as bytes and half words).
- Data values in the `.text` section cannot refer to other sections (for example, pointer constants, discussed in Chapter 4). However, such objects can refer to data within the `.text` section itself (this is important for *jump tables*, covered in Chapter 7).
- The data must reside within ±1MB of the instruction(s) that reference it. For example, you cannot create an array of data that exceeds 1MB.
- Only word and dword accesses are allowed when using the PC-relative addressing mode.
- As the data resides in the `.text` section, it is read-only; you cannot put variables in the `.text` section.

To use the PC-relative addressing mode, just reference the label you used to declare the object in the `.text` section:

```
        ldr w0, wordVar
          .
          .
          .
wordVar: .word 12345
```

Don't forget that all data declarations you put in the `.text` section need to be out of the execution path, preferably in the `.pool` section. (You'll see an exception to this rule in Chapter 5 when I discuss passing parameters in the code stream.)

3.6.2 Register-Indirect

Up to this point, most examples in this book have used the register-indirect addressing mode. *Indirect* means that the operand is not the actual address, but that the operand's value specifies the memory address to use. In a register-indirect addressing mode, the value held in the register is the address of the memory location to access. For example, the instruction

```
ldr x0, [x1]
```

tells the CPU to load X0's value from the location whose address is currently in X1. The square brackets around X1 tell Gas to use the register-indirect addressing mode.

The ARM has 32 forms of this addressing mode, one for each of the 32 general-purpose 64-bit registers (though X31 is not legal; use SP instead). You cannot specify a 32-bit register in the square brackets when using an indirect addressing mode.

Technically, you could load a 64-bit register with an arbitrary numeric value and access that location indirectly by using the register-indirect addressing mode:

```
ldr x1, =12345678
ldr x0, [x1]    // Attempts to access location 12345678
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause the OS to generate a segmentation fault because it's not always legal to access arbitrary memory locations. There are better ways to load the address of an object into a register, as you'll see shortly.

You can use the register-indirect addressing modes to access data referenced by a pointer, to step through array data, and, in general, whenever you need to modify an object's address while your program is running.

When using a register-indirect addressing mode, you refer to the value of a variable by its numeric memory address (the value you load into a register) rather than by the name of the variable. This is an example of using an *anonymous variable*.

The *aoaa.inc* include file provides the `lea` macro, which you can use to take the address of a variable and put it into a 64-bit register:

```
lea x1, j
```

After executing this `lea` instruction, you can use the `[x1]` register-indirect addressing mode to indirectly access the value of `j` (which is how almost every example up to this point has accessed memory). In section 3.8, "Getting the Address of a Memory Object," on page 153, you'll see how the `lea` macro works.

3.6.3 Indirect-Plus-Offset

Consider the following data declaration, similar to other examples given in this book:

```
bVar:  .byte 0, 1, 2, 3
```

If you load X1 with the address of `bVar`, you can access that byte (`0`) by using an instruction such as this:

```
ldrb w1, [x1]   // Load byte at bVar (0) into W1.
```

To access the other 3 bytes following that `0` in memory, you can use the *indirect-plus-offset* addressing mode. Here is the mode's syntax:

```
[Xn|SP, #signed_expression]
```

`Xn|SP` means X0 to X30 or SP, and *`signed_expression`* is a small integer expression in the range –256 to +255. This particular addressing mode will compute the sum of the address in X*n* (*n* = 0 to 30, or SP) with the signed constant and use that as the *effective memory address* (the memory address to access).

For example, if X1 contains the address of `bVar` from the previous example, the following instruction will fetch the byte just beyond `bVar` (that is, the byte containing `1` in that example):

```
ldrb  w0, [x1, #1] // Fetch byte at address X1 + 1.
```

Once again, the 32-bit instruction size severely limits the range of this addressing mode (only 9 bits are available for the signed offset). If you need a greater offset, you must explicitly add a value to the address in X1 (perhaps using a different register if you need to maintain the base address in X1). For example, the following code does this using X2 to hold the effective address:

```
add  x2, x1, #2000  // Access location X1 + 2000.
ldrb w2, [x2]
```

This computes `X2 = X1 + 2000` and loads W2 with the word at that address.

3.6.4 Scaled Indirect-Plus-Offset

The *scaled indirect-plus-offset* addressing mode is a somewhat more complex variant of the indirect-plus-offset mode. It incorporates a 12-bit unsigned constant into the instruction encoding that is scaled (multiplied) by 1, 2, 4, or 8, depending on the size of the data transfer. This provides a range extension to the 9-bit signed offset of the indirect-plus-offset mode.

This addressing mode uses the same syntax as the indirect-plus-offset addressing mode, except that it doesn't allow signed offsets:

```
[Xn|SP, #unsigned_expression]
```

For byte transfers (`ldrb`), the unsigned expression can be a value in the range 0 to 0xFFF (4,095). For half-word transfers (`ldrh`), the unsigned expression can be a value in the range 0 to 0x1FFE, but the offset must be even. For word transfers (`ldr`), the unsigned expression must be in the range 0 to 0x3FFC and must also be divisible by 4. For dword transfers, the unsigned expression must be in the range 0 to 0x7FF8 and must be divisible by 8. As you'll see in Chapter 4, these numbers work great for accessing elements of a byte, half-word, word, or double-word array.

Generally, the assembler will automatically select between the indirect-plus-offset and scaled indirect-plus-offset addressing modes, based on the value of the offset appearing in the addressing mode. Sometimes the choice might be ambiguous. For example:

```
ldr  w0, [X2, #16]
```

Here, the assembler could choose the scaled or unscaled versions of the addressing mode. Typically, it would choose the scaled form. Its decision shouldn't matter to your code; either form will load the appropriate word in memory into the W0 register.

If, for some reason, you wish to explicitly specify the unscaled addressing mode, you can do so using the `ldur` and `stur` instructions (load or store register unscaled).

3.6.5 Pre-indexed

The *pre-indexed* addressing mode is very similar to the indirect-plus-offset addressing mode, insofar as it combines a 64-bit register and a signed 9-bit offset. However, this addressing mode copies the sum of the register and offset into the register before accessing memory. In the end, it accesses the same address as the indirect-plus-offset mode, but once the instruction finishes, the index register points into memory at the indexed location. This mode is useful for stepping through arrays and other data structures by incrementing the register after each access in a loop.

Here's the syntax for the pre-indexed addressing mode:

```
[Xn|SP, #signed_expression]!  // Xn|SP has the usual meaning.
```

The `!` at the end of this sequence differentiates the pre-indexed addressing mode. As with the indirect-plus-offset mode, the *signed_expression* value is limited—in this case, to 9 bits (–256 to +255).

The following code fragment uses this addressing mode:

```
bVar:  .byte 0, 1, 2, 3
         .
         .
         .
       lea  x0, bVar-1  // Initialize with adrs of bVar - 1.
       mov  x1, 4
loop:  ldrb w2, [x0, #1]!

       Do something with the byte in W2.

       subs x1, x1, #1
       bne  loop
```

On the first iteration of this loop, the addressing mode adds `1` to X0 so that it points at the first byte in the `bVar` array of 4 bytes. This also leaves X0 pointing at that first byte. On each successful iteration of the loop, X0 is incremented by 1, accessing the next byte in the `bVar` array.

The `subs` instruction will set the Z flag when it decrements X1 down to 0. When that happens, the `bne` (branch if Z = 0) instruction will fall through, terminating the loop.

3.6.6 Post-Indexed

The post-indexed addressing mode is very similar to the pre-indexed addressing mode, except it uses the value of the register as the memory address *before* updating the register with the signed immediate value. Here's the syntax for the post-indexed addressing mode:

```
[Xn|SP], #signed_expression  // Xn|SP has the usual meaning.
```

Again, the *signed_expression* is limited to 9 bits (–256 to +255).

The example of the previous section can be rewritten and slightly improved by using the post-indexed addressing mode:

```
bVar:  .byte 0, 1, 2, 3
         .
         .
         .
       lea  x0, bVar
       mov  x1, 4
loop:  ldrb w2, [x0], #1

       Do something with the byte in W2.

       subs x1, x1, #1
       bne  loop
```

This example starts with X0 pointing at `bVar` and ends with X0 pointing at the first byte beyond the (four-element) `bVar` array. On the first iteration of this loop, the `ldr` instruction first uses the value in X0, pointing at `bVar`, then increments X0 after fetching the byte where X0 points.

3.6.7 Scaled-Indexed

The *scaled-indexed* addressing mode contains two register components (rather than a register and an immediate constant) that form the effective address. The syntax for this mode is the following:

```
[Xn|SP, Xi]
[Xn|SP, Wi, extend]
[Xn|SP, Xi, extend]
```

The first form is the easiest to understand: it computes the effective address (EA) by adding the values in X*n* (or SP) and X*i*. Generally, X*n* (or SP) is known as the *base address*, and the value in X*i* is the index (which must be X0 to X30 or XZR). The base address is the lowest memory address of an object, and the index is an offset from that base address (much like the immediate constants in the indirect-plus-offset addressing mode). This is just a simple *base* + *index* addressing mode: no scaling takes place.

WHY XN|SP, NOT X31?

As noted in section 1.6, "The ARM64 CPU Architecture," on page 11, the stack pointer register, SP, is the same as X31. However, if you try to use X31 as the base register in an addressing mode, Gas will report an error. This is because the ARM64 CPU actually maps two separate registers to X31: SP and XZR (the zero register). You use one of those register names rather than X31.

In addressing modes, the ARM does not allow you to use XZR as a base register. You can, however, use SP as the base register. Conversely, XZR *is* allowed as an index register (though it's somewhat redundant to do so), and SP is not allowed there.

The base + index form is useful in these situations:

- You have a pointer to an array object in a register (X*n*, the base address), and you want to access an element of that array by using an integer index (typically in a memory variable). In this case, you would load the index into the index register (X*i*) and use the base + index mode to access the actual element.
- You want to use the indirect-plus-offset addressing mode, but the offset is outside the range –256 to +255. In this case, you can load the larger offset into X*i* and use the base + index addressing mode to access the memory location regardless of the offset.

The second and third forms of the scaled-indexed addressing mode provide an extension/scaling operation, which is quite useful for indexing into arrays whose element size is larger than a byte. Of these two

scaled-indexed modes, one uses a 32-bit register as the index register, and the other uses a 64-bit register.

The 32-bit form is convenient because most of the time indices into an array are held in a 32-bit integer variable. If you load that 32-bit integer into a 32-bit register (W*i*), you can easily use it as an index into an array with the

```
[Xn, Wi, extend]
```

form of the scaled-indexed addressing mode.

Ultimately, all effective addresses turn out to be 64 bits. In particular, when the CPU adds X*n* and W*i* together, it must somehow extend the W*i* index value to 64 bits prior to adding them. The *extend* operator tells Gas how to extend W*i* to 64 bits.

The simplest forms of *extend* are the following:

```
[Xn|SP, Wi, uxtw]
[Xn|SP, Wi, sxtw]
```

The [X*n*|SP, W*i*, uxtw] form zero-extends W*i* to 64 bits before adding it to X*n*, while the [X*n*|SP, W*i*, sxtw] form sign-extends W*i* to 64 bits before the addition.

Another form of the scaled-indexed addressing mode introduces the *scaled* component. This form allows you to load elements from an array of bytes, half words, words, or dwords scaled by the size of the array element (1, 2, 4, or 8 bytes). These particular forms are not stand-alone addressing modes that can be used with an arbitrary `ldr` or `str` instruction. Instead, each addressing mode form is tied to a specific instruction size. The following is the allowable syntax for the `ldrb`/`ldrsb` and `strb` instructions (W*d* is a 32-bit destination register, and W*s* is a 32-bit source register):

```
ldrb  Wd, [Xn|SP, Wi, sxtw #0]  // #0 is optional;
ldrb  Wd, [Xn|SP, Wi, uxtw #0]  // 0 is default shift.
ldrb  Wd, [Xn|SP, Xi, lsl #0]

ldrsb Wd, [Xn|SP, Wi, sxtw #0]
ldrsb Wd, [Xn|SP, Wi, uxtw #0]
ldrsb Wd, [Xn|SP, Xi, lsl #0]

strb  Ws, [Xn|SP, Wi, sxtw #0]
strb  Ws, [Xn|SP, Wi, uxtw #0]
strb  Ws, [Xn|SP, Xi, lsl #0]
```

These forms zero- or sign-extend W*i* (or X*i*) and add the result with X*n* to produce the EA. The previous instructions are equivalent to the following (because the `#0` is optional):

```
ldrb  Wd, [Xn|SP, Wi, sxtw]
ldrb  Wd, [Xn|SP, Wi, uxtw]
ldrb  Wd, [Xn|SP, Xi]
```

```
ldrsb Wd, [Xn|SP, Wi, sxtw]
ldrsb Wd, [Xn|SP, Wi, uxtw]
ldrsb Wd, [Xn|SP, Xi]

strb  Ws, [Xn|SP, Wi, sxtw]
strb  Ws, [Xn|SP, Wi, uxtw]
strb  Ws, [Xn|SP, Xi]
```

For the `ldrh`/`ldrsh` and `strh` instructions, you can specify either the 0 (×1) or 1 (×2) scale factor:

```
ldrh  Wd, [Xn|SP, Wi, sxtw #1]  // #0 is also legal, or
ldrh  Wd, [Xn|SP, Wi, uxtw #1]  // no immediate value (which
ldrh  Wd, [Xn|SP, Xi, lsl #1]   // defaults to 0).

ldrsh Wd, [Xn|SP, Wi, sxtw #1]
ldrsh Wd, [Xn|SP, Wi, uxtw #1]
ldrsh Wd, [Xn|SP, Xi, lsl #1]

strh  Ws, [Xn|SP, Wi, sxtw #1]
strh  Ws, [Xn|SP, Wi, uxtw #1]
strh  Ws, [Xn|SP, Xi, lsl #1]
```

With a scaling factor of `#1`, these addressing modes compute W*i* × 2 or X*i* × 2 (after any zero or sign extension) and then add the result with the value in X*n* to produce the EA. This scales the EA to access half-word values (2 bytes per array element). If the scaling factor is `#0`, no scaling occurs, as the scaling factor is 2^0. The preceding code must multiply W*i* or X*i* by an appropriate scaling factor, if needed. Loading or storing half words allows a scaling factor of only 0 or 1.

For the 32-bit `ldr` instruction (W*d* is the destination register) and `str` instruction (W*s* is the 32-bit source register), the allowable scaling factors are 0 (×1) or 2 (×4):

```
ldr  Wd, [Xn|SP, Wi, sxtw #2]  // #0 is also legal, or
ldr  Wd, [Xn|SP, Wi, uxtw #2]  // no immediate value (which
ldr  Wd, [Xn|SP, Xi, lsl #2]   // defaults to 0).

str  Ws, [Xn|SP, Wi, sxtw #2]
str  Ws, [Xn|SP, Wi, uxtw #2]
str  Ws, [Xn|SP, Xi, lsl #2]
```

Finally, for the 64-bit `ldr` and `str` instructions, the allowable scaling factors are 0 (×1) and 3 (×8):

```
ldr  Xd, [Xn|SP, Wi, sxtw #3]  // #0 is also legal, or
ldr  Xd, [Xn|SP, Wi, uxtw #3]  // no immediate value (which
ldr  Xd, [Xn|SP, Xi, lsl #3]   // defaults to 0).

str  Xs, [Xn|SP, Wi, sxtw #3]
str  Xs, [Xn|SP, Wi, uxtw #3]
str  Xs, [Xn|SP, Xi, lsl #3]
```

You'll see the main uses for the scaled-indexed addressing modes in the next chapter, when it discusses accessing elements of arrays.

3.7 Address Expressions

Often, when accessing variables and other objects in memory, you will need to access locations immediately before or after a variable rather than at the address of the variable. For example, when accessing an element of an array, or a field of a struct, the exact element or field is probably not at the address of the variable itself. *Address expressions* provide a mechanism to access memory at an offset from the variable's address.

Consider the following legal Gas syntax for a memory address. This isn't a new addressing mode but simply an extension of the PC-relative addressing mode:

```
varName + offset
```

This form computes its effective address by adding the constant offset to the variable's address. For example, the instruction

```
ldr w0, i + 4
```

loads the W0 register with the word in memory that is 4 bytes beyond the `i` object (which, presumably, is in the `.text` section; see Figure 3-10).

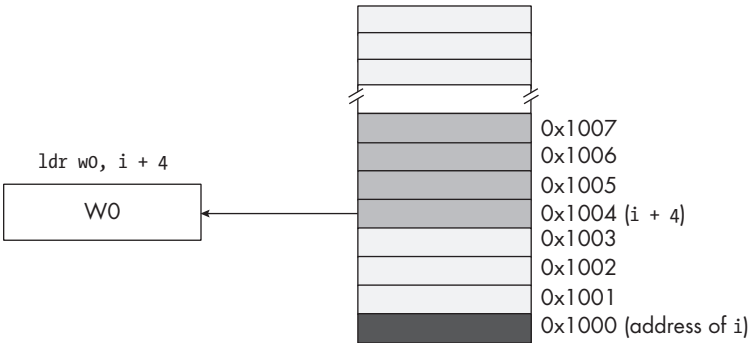


Figure 3-10: Using an address expression to access data beyond a variable

The *offset* value in this example must be a constant (for example, 3). If *Index* is a word variable, then *varName* + *Index* is not a legal address expression. If you wish to specify an index that varies at runtime, you must use one of the indirect or scaled-indexed addressing modes. Also remember that the offset in *varName* + *offset* is a byte address. This does not properly index into an array of objects unless *varName* is an array of bytes.

NOTE *The ARM CPU does not allow the use of the* `ldrb` *and* `ldrh` *instructions when using the PC-relative addressing mode. You can only load words or double words when using this addressing mode. Furthermore, because the instructions don't encode the LO 2 bits of the offset, any offset you specify using an address expression must be a multiple of 4.*

Until this point, the offset in the addressing mode examples has always been a single numeric constant. However, Gas also allows a constant expression anywhere an offset is legal. A *constant expression* consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, and a wide variety of others, as shown in Table 3-4. Note that operators at the same precedence level are left-associative.

Table 3-4: Gas Constant Expression Operators

Operator	Precedence	Description
+	3	Unary plus (no effect on expression)
-	3	Unary minus (negates expression)
*	2	Multiplication
/	2	Division
<<	2	Shift left
>>	2	Shift right
|	1	Bitwise OR
&	1	Bitwise AND
^	1	Bitwise XOR
!	1	Bitwise AND-NOT
+	0	Addition
-	0	Subtraction

Most address expressions, however, involve only addition, subtraction, multiplication, and sometimes division. Consider the following example:

```
ldr w0, X + 2*4
```

This instruction will move the byte at address `X + 8` into the W0 register.

The value `X + 2*4` is an address expression that is always computed at compile time, never while the program is running. When Gas encounters the preceding instruction, it calculates

```
2 × 4
```

on the spot and adds this result to the base address of `X` in the `.text` section. Gas encodes this single sum (base address of `X` plus 8) as part of the instruction; it does not emit extra instructions (that would waste time) to compute this sum for you at runtime. Because Gas computes the value of address

expressions at compile time, and therefore Gas cannot know the runtime value of a variable while it is compiling the program, all components of the expression must be constants.

Address expressions are useful for accessing the data in memory beyond a variable, particularly when you've used directives like `.byte`, `.hword`, `.word`, and so on in a `.data` or `.text` section to tack on additional values after a data declaration. For example, consider the program in Listing 3-1 that uses address expressions to access the four consecutive words associated with memory object `i` (each word is 4 bytes apart in memory).

```
// Listing3-1.S
//
// Demonstrates address expressions

#include "aoaa.inc"

            .data
saveLR:     .dword      0
outputVal:  .word       0

ttlStr:     .asciz      "Listing 3-1"
fmtStr1:    .asciz      "i[0]=%d "
fmtStr2:    .asciz      "i[1]=%d "
fmtStr3:    .asciz      "i[2]=%d "
fmtStr4:    .asciz      "i[3]=%d\n"

            .text
            .extern     printf

            .align      2
i:          .word       0, 1, 2, 3

// Return program title to C++ program:

            .global     getTitle
getTitle:
            lea         x0, ttlStr
            ret

// Here is the asmMain function:

            .global     asmMain
asmMain:

// "Magic" instruction offered without
// explanation at this point:

            sub     sp, sp, #256
```

```
// Save LR so we can return to the C++
// program later:

            lea     x0, saveLR
            str     lr, [x0]

// Demonstrate the use of address expressions:

            lea     x0, fmtStr1
          ❶ ldr     w1, i + 0
            lea     x2, outputVal
            str     w1, [x2]
            vparm2  outputVal
            bl      printf

            lea     x0, fmtStr2
          ❷ ldr     w1, i + 4
            lea     x2, outputVal
            str     w1, [x2]
            vparm2  outputVal
            bl      printf

            lea     x0, fmtStr3
          ❸ ldr     w1, i + 8
            lea     x2, outputVal
            str     w1, [x2]
            vparm2  outputVal
            bl      printf

            lea     x0, fmtStr4
          ❹ ldr     w1, i + 12
            lea     x2, outputVal
            str     w1, [x2]
            vparm2  outputVal
            bl      printf

            lea     x0, saveLR
            ldr     lr, [x0]
            add     sp, sp, #256
            ret
```

Loading W1 from location `i + 0` fetches `0` from the word array ❶. Loading W1 from location `i + 4` fetches `1` from the second word in the array, located 4 bytes beyond the first element ❷. Loading W1 from location `i + 8` fetches `2` from the third word in the array ❸, located 8 bytes beyond the first element. Loading W1 from location `i + 12` fetches `3` from the fourth word in the array ❹, located 12 bytes beyond the first element.

Here's the program's output:

```
$ ./build Listing3-1
$ ./Listing3-1
Calling Listing3-1:
i[0]=0 i[1]=1 i[2]=2 i[3]=3
Listing3-1 terminated
```

Because the value at the address of `i` is `0`, the output displays the four values `0`, `1`, `2`, and `3` as though they were array elements. The address expression `i + 4` tells Gas to fetch the word appearing at `i`'s address plus 4. This is the value `1`, because the `.word` statement in this program emits the value `1` to the `.text` segment immediately after the (word/4-byte) value `0`. Likewise, for `i + 4` and `i + 8`, this program displays the values `2` and `3`.

3.8 Getting the Address of a Memory Object

Up to this point, this book has used the `lea` macro to obtain the address of a memory object. Now that this chapter has provided the necessary prerequisite information, instead of treating `lea` like a black box, it's time to look behind the curtains to see what this macro is doing for you.

The ARM CPU provides two instructions for computing the effective address of a symbol in an assembly language program. The first is `adr`:

```
adr  Xd, label
```

This instruction loads the 64-bit destination register (X*d*) with the address of the specified label. Because instruction encodings (operation codes, or *opcodes*) are limited to 32 bits, a huge caveat is attached to `adr`: it has room for only a 21-bit offset within the opcode, so *`label`* must be a PC-relative address within ±1MB of the `adr` instruction. This effectively limits `adr` to taking the address of symbols within the `.text` section.

To rectify this situation, the ARM CPU also provides the `adrp` (address of a page) instruction. This instruction has roughly the same generic syntax as `adr`:

```
adrp Xd, label
```

The instruction loads the address of the MMU page containing the *`label`* into the destination register. By adding the offset of the label into that page to the value in X*d*, you can obtain the actual address of the memory object, using code that looks something like this:

```
adrp Xd, label
add  Xd, Xd, page_offset_of_label
```

At this point, X*d* will contain the address of *`label`*.

This scheme has a couple of issues: first, computing the page offset of the *`label`* symbol is done differently in macOS versus Linux. Second, when you use the syntax just given to try the `adrp` instruction, you'll find that Gas rejects this on macOS.

Let's first consider the Linux solutions to these problems, as they're a little simpler than those for macOS. If you're not creating a PIE application and the symbol is less than ±1MB away, you don't have to use the `adrp` instruction. Instead, you can get by with the single `adr` instruction. If the data is more than ±1MB from the `adr`, you must use the `adrp` version. If you

need to reference a memory object outside the .text section, you must use the adrp/add sequence. Here's the code to do this:

```
adrp x0, label
add  x0, x0, :lo12:label
```

The :lo12: item is a special operator that tells Gas to extract the LO 12 bits of *label*'s relocatable address; this value is the index into a 4,096-byte memory management page. For more information on this operator, see section 3.12, "For More Information," on page 167. Unfortunately, the macOS assembler uses a completely different syntax to obtain the LO 12 bits of an address; you must use the following instead:

```
adrp x0, label@PAGE
add  x0, x0, label@PAGEOFF
```

The lea macro resolves this issue, automatically expanding into the appropriate sequence for whichever OS you're using.

LINUX VS. MACOS: ABSOLUTE ADDRESSES

Apple's macOS (and presumably, iOS, iPadOS, and so on) is far more restrictive about what you can and cannot do in a PIE program. Specifically, macOS does not allow any absolute pointers in your .text section that reference other sections. Linux, on the other hand, doesn't have a problem with this at all, in either PIE or non-PIE mode.

For example, say you're working in Linux and have the following symbol in your .data section:

```
var: .word 55
```

You can use the instruction

```
ldr x0, =var
```

to load the address of that symbol into X0. If you try to use this instruction in macOS, however, the program will give the following complaint:

```
ld: Absolute addressing not allowed in arm64 code but used in
          'noPrint' referencing 'var'
```

Likewise, if you put the statement

```
ptrToVar: .dword var
```

in your .text section somewhere, Linux is perfectly happy with it, but macOS will reject it, using roughly the same message.

Pointers into the `.text` section from other sections are perfectly acceptable to Gas under macOS. Apparently, Apple thinks that the only way hackers are going to determine your data memory location is by looking for addresses buried in the executable code, while pointers in your `.data`, `.rodata`, and other sections are immune to such attacks.

Ultimately, this means that you'll need to use the `adrp` instruction (or the `lea` macro) to obtain at least your first pointer out of the `.text` section. This makes assembly language programming a touch more difficult under macOS than under Linux. Fortunately, the `lea` macro helps smooth out these issues.

3.9 The Push and Pop Operations

The ARM maintains a hardware stack in the stack segment of memory (for which the OS reserves the storage). The *stack* is a dynamic data structure that grows and shrinks according to certain needs of the program. It also stores important information about the program, including local variables, subroutine information, and temporary data.

The ARM CPU controls its stack via the SP register. When your program begins execution, the OS initializes SP with the address of the last memory location in the stack memory segment. Data is written to the stack segment by *pushing* data onto the stack and *popping* it off the stack.

The ARM stack must always be 16-byte aligned—that is, the SP register must always contain a value that is a multiple of 16. If you load the SP register with a value that is not 16-byte aligned, the application will immediately terminate with a bus error fault. One of the stack's primary purposes is to provide a temporary storage area where you can save things such as register values. You will typically push a register's value onto the stack, do some work (such as calling a function) that uses the register, and then pop that value off the stack and back into the register when you want to restore its value. However, the general-purpose registers are only 64 bits (8 bytes); pushing a dword value on the stack will not leave it 16-byte aligned, which will crash the system.

In this section, I'll describe how to push and pop register values. Then I'll present three solutions to the problem of pushing dword values that don't leave the stack 16-byte aligned: wasting storage; pushing two registers simultaneously; and reserving storage on the stack, then moving the register's data into this reserved area.

3.9.1 Using Double Loads and Stores

The `ldp` instruction will load two registers from memory simultaneously. The generic syntax for this instruction is shown here:

```
ldp  Xd1, Xd2, mem  // mem is any addressing mode
ldp  Wd1, Wd2, mem  // except PC-relative.
```

The first form will load Xd_1 from the memory location specified by *mem* and Xd_2 from the memory location 8 bytes later. The second form will load Wd_1 from the specified memory location and Wd_2 from the location 4 bytes later.

The `stp` instruction has a similar syntax; it stores a pair of registers into adjacent memory locations:

```
stp  Xd1, Xd2, mem  // Store Xd1 to mem, Xd2 to mem + 8.
stp  Wd1, Wd2, mem  // Store Wd1 to mem, Wd2 to mem + 4.
                    // mem is any addressing mode except
                    // PC-relative.
```

These instructions have many uses. With respect to using the stack, however, the forms that load and store a pair of 64-bit registers will manipulate 16 bytes at a time—exactly what you need when pushing and popping data on the stack.

3.9.2 Executing the Basic Push Operation

Many CPUs, such as the Intel x86-64, provide an explicit instruction that will push a register onto the stack. Because of the 16-byte stack alignment requirement, you can't push a single 8-byte register onto the stack (without creating a stack fault). However, if you're willing to use 16 bytes of space on the stack to hold a single register's value, you can push that register's value on the stack with the following instruction:

```
str Xs, [sp, #-16]!
```

Remember, the pre-indexed addressing mode will first add –16 to SP and then store X*s* (the source register) at the new location pointed at by SP. This store operation writes only to the LO 8 bytes of the 16-byte block created by dropping SP down by 16 (wasting the HO 8 bytes). However, this scheme keeps the CPU happy, so you won't get a bus error.

This push operation does the following:

```
SP := SP - 16
[SP] := Xs
```

For example, assuming that SP contains 0x00FF_FFE0, the instruction

```
str x0, [sp, #-16]!
```

will set SP to 0x00FF_FFD0 and store the current value of X0 into memory location 0x00FF_FFD0, as Figures 3-11 and 3-12 show.

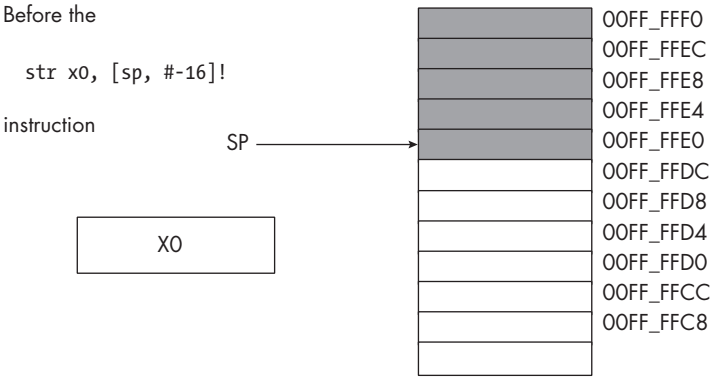


Figure 3-11: The stack segment before the `str x0, [sp, #-16]!` operation

After the `str` instruction, the stack looks like Figure 3-12.

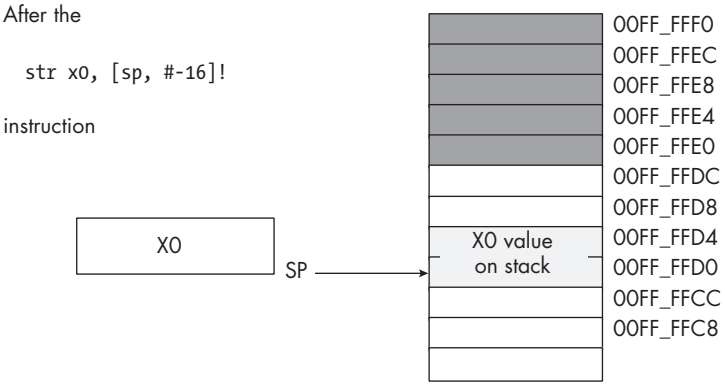


Figure 3-12: The stack segment after the `str x0, [sp, #-16]!` operation

Although this wastes 8 bytes of space on the stack (shown at addresses 0x00FF_FFD8 through 0x00FF_FFDF), the usage is probably temporary, and the stack space will be reclaimed when the program pops the data off the stack later.

3.9.3 Executing the Basic Pop Operation

The pop operation can be handled using the post-indexed addressing mode and a `ldr` instruction:

```
ldr Xd, [sp], #16
```

This instruction fetches the data from the stack, where SP is pointing, and copies that data into the destination register (X*d*). When the operation is complete, this instruction adjusts SP by 16, restoring it to its original value (its value before the push operation). Figure 3-13 shows the stack before the pop operation.

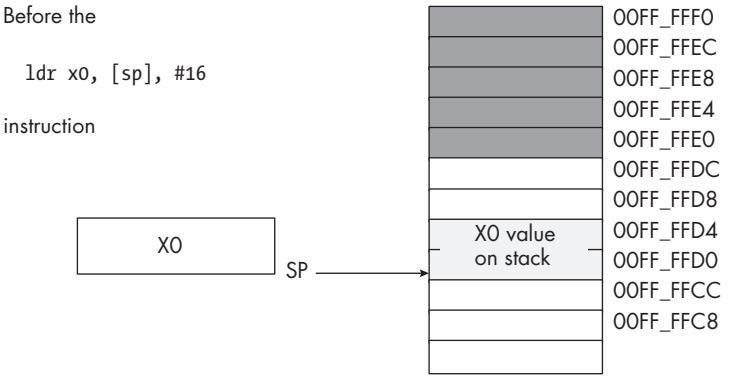


Figure 3-13: Before the `str` operation

Figure 3-14 shows the stack organization after executing `ldr`.

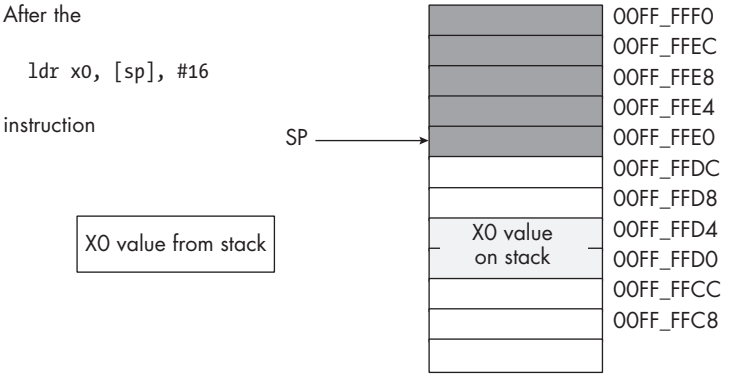


Figure 3-14: After the pop operation

Popping a value does not erase the value in memory; it just adjusts the stack pointer so that it points at the next value above the popped value. However, never attempt to access a value you've popped off the stack. The next time something is pushed onto the stack, the popped value will be obliterated. Because your code isn't the only thing that uses the stack (for example, the OS uses the stack to do subroutines), you cannot rely on data remaining in stack memory once you've popped it off the stack.

3.9.4 Preserving at Least Two Registers

If you need to preserve at least two registers, you can reclaim the wasted space shown in Figures 3-11 and 3-12 by using the `stp` instruction rather than `str`. The following code fragment demonstrates how to push and pop both X0 and X7 simultaneously:

```
stp  x0, x7, [sp, #-16]!
 .
 .   // Use X0 and X7 for other purposes.
 .
ldp  x0, x7, [sp], #16  // Restore X0 and X7.
```

The third way to push data on the stack is to drop SP down by a multiple of 16 bytes and then store the value into the stack area by indexing off the SP register. The following code does basically the same thing as the `stp`/`ldp` pair:

```
sub  sp, sp, #16   // Make room for X0 and X7.
stp  x0, x7, [sp]
 .
 .   // Use X0 and X7 for other purposes.
 .
ldp  x0, x7, [sp]
add  sp, sp, #16
```

While this clearly takes more instructions (and, therefore, takes longer to execute), it's possible to reserve the stack storage only once within a function and reuse that space throughout the execution of the function. You'll see examples of this in Chapter 5.

3.9.5 Preserving Register Values on the Stack

As you've seen in previous examples, the stack is a great place to temporarily preserve registers so they can be used for other purposes. Consider the following program outline:

```
Some instructions that use the X20 register.

Some instructions that need to use X20, for a
different purpose than the above instructions.

Some instructions that need the original value in X20.
```

The push and pop operations are perfect for this situation. By inserting a push sequence before the middle sequence, and a pop sequence after the middle sequence, you can preserve the value in X20 across those calculations:

```
Some instructions that use the X20 register.

     str x20, [sp, #-16]!

Some instructions that need to use X20, for a
different purpose than the above instructions.
```

```
    ldr x20, [sp], #16

Some instructions that need the original value in X20.
```

This push sequence copies the data computed in the first sequence of instructions onto the stack. Now the middle sequence of instructions can use X20 for any purpose it chooses. After the middle sequence of instructions finishes, the pop sequence restores the value in X20 so the last sequence of instructions can use the original value in X20.

3.9.6 Saving Function Return Addresses on the Stack

Throughout the example programs up to this point, I've preserved the return address appearing in the link register (LR) by using instructions like the following:

```
lea  x0, saveLR
str  lr, [x0]
 .
 .
 .
lea  x0, saveLR
ldr  lr, [x0]
ret
```

I've also mentioned that this is a *truly horrible* way of preserving the value in LR. It takes six instructions to accomplish (remember, `lea` expands into two instructions), making it slower and bulkier than it needs to be. This scheme also creates problems when you have one user-written function calling another: all of a sudden, you need two separate `saveLR` variables, one for each function. In the presence of recursion (see Chapter 5) or, worse, multithreaded code, this mechanism fails completely.

Fortunately, saving return addresses in the stack is the perfect solution. The stack's LIFO structure (see the next section) completely emulates the way (nested) function calls and returns work, and it takes only a single instruction to push LR onto the stack or pop LR off the stack. The earlier code sequence can be easily replaced by:

```
str  lr, [sp, #-16]!
 .
 .
 .
ldr  lr, [sp], #16
ret
```

Using the stack to save and restore the LR register is probably the most common use of the stack. Chapter 5 discusses managing return addresses and other function-related values in much greater depth.

3.10 Pushing and Popping Stack Data

You can push more than one value onto the stack without first popping previous values off the stack. However, the stack is a *last-in, first-out (LIFO)* data structure, so you must be careful in the way you push and pop multiple values.

For example, suppose you want to preserve X0 and X1 across a block of instructions. The following code demonstrates the obvious (but incorrect) way to handle this:

```
str  x0, [sp, #-16]!
str  x1, [sp, #-16]!
   Code that uses X0 and X1 goes here.
ldr  x0, [sp], #16
ldr  x1, [sp], #16
```

Unfortunately, this code will not work properly! Figures 3-15 through 3-18 show the problem, with each box in these figures representing 8 bytes (note the addresses). Because this code pushes X0 first and X1 second, the stack pointer is left pointing at X1's value on the stack.

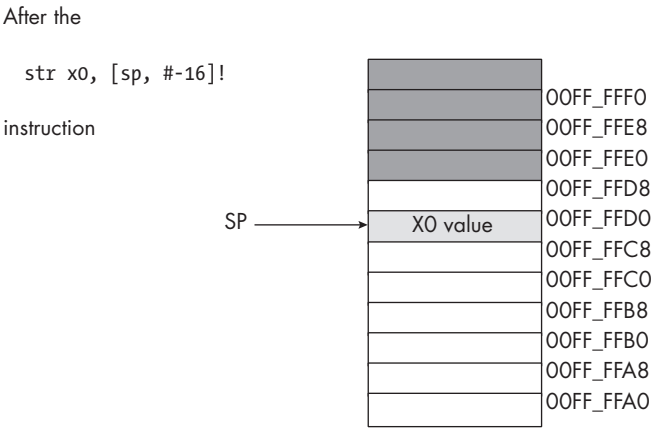


Figure 3-15: The stack after pushing X0

Figure 3-16 shows the stack after pushing the second register (X1).

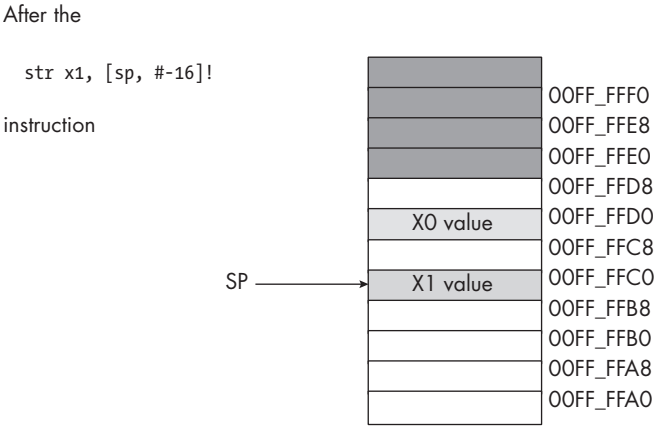


Figure 3-16: The stack after pushing X1

When the `ldr x0, [sp], #16` instruction comes along, it removes the value that was originally in X1 from the stack and places it in X0 (see Figure 3-17).

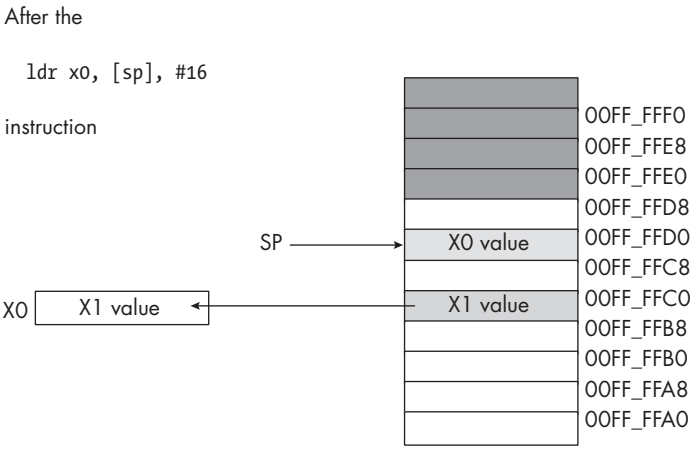


Figure 3-17: The stack after popping X0

Likewise, the `ldr x1, [sp], #16` instruction pops the value that was originally in X0 into the X1 register. In the end, this code manages to swap the values in the registers by popping them in the same order that it pushes them (see Figure 3-18).

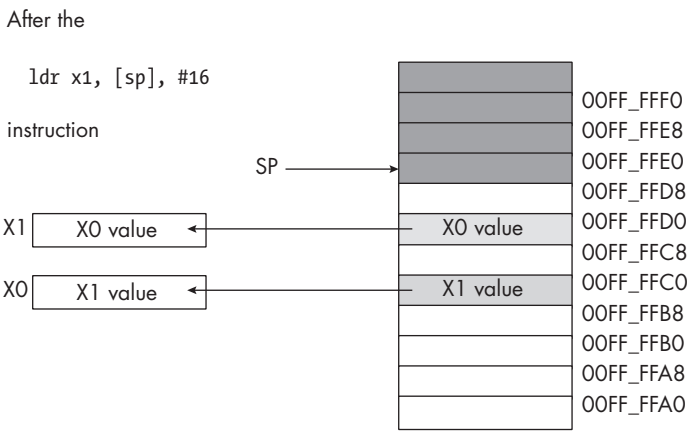


Figure 3-18: The stack after popping X1

To rectify this problem, because the stack is a LIFO data structure, the first thing you must pop is the last thing you push onto the stack. Therefore, *always pop values in the reverse order that you push them.*

The correction to the previous code is shown here:

```
str  x0, [sp, #-16]!
str  x1, [sp, #-16]!
   Code that uses X0 and X1 goes here.
ldr  x1, [sp], #16
ldr  x0, [sp], #16
```

Also remember to *always pop exactly the same number of bytes that you push.* In general, this means you'll need exactly the same the number of pushes and pops. If you have too few pops, you will leave data on the stack, which may confuse the running program. If you have too many pops, you will accidentally remove previously pushed data, often with disastrous results.

As a corollary, *be careful when pushing and popping data within a loop.* It's easy to put the pushes in a loop and leave the pops outside the loop (or vice versa), creating an inconsistent stack. Remember, it's the execution of the push and pop operations that matters, not the number of push and pop operations that appear in your program. At runtime, the number (and order) of the push operations the program executes must match the number (and reverse order) of the pop operations.

Finally, remember that *the ARM requires the stack to be aligned on a 16-byte boundary.* If you push and pop items on the stack (or use any other instructions that manipulate the stack), make sure that the stack is aligned on a 16-byte boundary before calling any functions or procedures that adhere to the ARM requirements.

3.10.1 Removing Data from the Stack Without Popping It

You may often discover that you've pushed data you no longer need onto the stack. Although you could pop the data into an unused register, there

is an easier way to remove unwanted data from the stack: simply adjust the value in the SP register to skip over the unwanted data on the stack.

Consider the following dilemma (in pseudocode, not actual assembly language):

```
str  x0, [sp, #-16]!  // Push X0.
str  x1, [sp, #-16]!  // Push X1.

Some code that winds up computing some values we want
to keep in X0 and X1.

if( Calculation_was_performed ) then

      // Whoops, we don't want to pop X0 and X1!
      // What to do here?

else

      // No calculation, so restore X1, X0.

      ldr  x1, [sp], #16
      ldr  x0, [sp], #16

endif;
```

Within the `then` section of the `if` statement, this code wants to remove the old values of X0 and X1 without otherwise affecting any registers or memory locations. How can you do this?

Because the SP register contains the memory address of the item on the top of the stack, we can remove the item from the top by adding the size of that item to the SP register. In the preceding example, we wanted to remove two dword items from the top. We can easily accomplish this by adding 16 to the stack pointer:

```
str  x0, [sp, #-16]!  // Push X0
str  x1, [sp, #-16]!  // Push X1

Some code that winds up computing some values we want to keep
into rax and rbx.

if( Calculation_was_performed ) then

     // Remove unneeded X0/X1 values
     // from the stack.

     add  sp, sp, #32

else

     // No calculation, so restore X1, X0.
```

```
    ldr  x1, [sp], #16
    ldr  x0, [sp], #16

endif;
```

Effectively, this code pops the data off the stack without moving it anywhere. This code is faster than two dummy pop operations, because it can remove any number of bytes from the stack with a single `add` instruction.

Remember to keep the stack aligned on a quad-word (16-byte) boundary. This means you should always add a constant that is a multiple of 16 to SP when removing data from the stack.

3.10.2 Accessing Data Pushed onto the Stack Without Popping It

Once in a while, you'll push data onto the stack and will want to get a copy of that data's value, or perhaps you'll want to change that data's value without actually popping the data off the stack (that is, you wish to pop the data off the stack at a later time). The ARM `[SP, #±offset]` addressing mode provides the mechanism for this.

Consider the stack after the execution of the following instruction:

```
stp  x0, x1, [sp, #-16]!  // Push X0 and X1.
```

This produces the stack result shown in Figure 3-19.

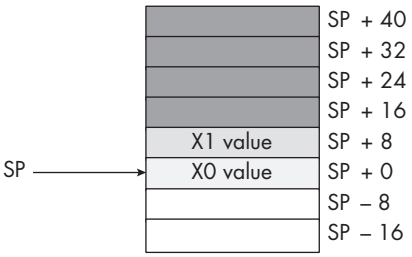


Figure 3-19: The stack after pushing X0 and X1

If you wanted to access the original X0 value without removing it from the stack, you could cheat by popping the value, then immediately pushing it again. Suppose, however, that you wish to access X1's old value or another value even farther up the stack. Popping all the intermediate values and then pushing them back onto the stack is problematic at best, impossible at worst.

However, as Figure 3-19 shows, each value pushed on the stack is at a certain offset from the SP register in memory. Therefore, we can use the `[SP, #±offset]` addressing mode to gain direct access to the value we are interested in. In the preceding example, you can reload X1 with its original value by using this single instruction:

```
ldr  x1, [sp, #8]
```

This code copies the 8 bytes starting at memory address SP + 8 into the X1 register. This value just happens to be the previous value of X1 that was pushed onto the stack. You can use this same technique to access other data values you've pushed onto the stack.

Don't forget that the offsets of values from SP into the stack change every time you push or pop data. Abusing this feature can create code that is hard to modify; using this feature throughout your code will make it difficult to push and pop other data items between the point where you first push data onto the stack and the point where you decide to access that data again using the `[SP, #±offset]` memory addressing mode.

The previous section pointed out how to remove data from the stack by adding a constant to the SP register. That pseudocode example could probably be written more safely as this:

```
stp  x0, x1, [sp, #-16]!

Some code that winds up computing some values we want
to keep into X0 and X1.

if( Calculation_was_performed ) then

     // Overwrite saved values on the stack with
     // new X0/X1 values (so the pops that
     // follow won't change the values in X0/X1).

     stp  x0, x1, [sp]

endif;
ldp  x0, x1, [sp], #16
```

In this code sequence, the calculated result was stored over the top of the values saved on the stack. Later, when the program pops the values, it loads these calculated values into X0 and X1.

THE "MAGIC" INSTRUCTIONS

In most of the example programs in this book so far, the following lines of code have appeared in `asmMain` (and in other functions):

```
// "Magic" instruction offered without
// explanation at this point:

sub     sp, sp, #256
 .
 .
 .
add     sp, sp, #256
```

At this point, it should be clearer what this code is doing: reserving storage on the stack (and removing that storage before returning from the function).

Chapter 5 covers this scheme in greater detail when it discusses local variables and parameter functions. For the time being, just know that the purpose of these statements is to reserve storage on the stack for parameters being passed to the `printf()` function via the `vparmn` macros.

3.11 Moving On

This chapter discussed memory organization and access, and how to create and access memory variables on the ARM CPU. It went over problems that can occur when accessing data beyond the end of a data structure that crosses over into a new MMU page, then discussed little- and big-endian memory organizations and how to use the ARM memory addressing modes and address expressions to access those memory objects in multiple ways. You learned how to align data in memory to improve performance, how to obtain the address of a memory object, and the purpose of the ARM stack structure.

Thus far, this book has generally employed only basic data types such as different-sized integers, characters, Boolean objects, and floating-point numbers. Fancier data types, such as pointers, arrays, strings, and structs are the subject of the next chapter.

3.12 For More Information

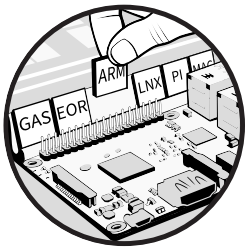
- See *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html* for details on the GNU assembler.
- Learn more about the GNU linker at *https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html*.
- For more about the macOS (LLVM) linker, see *https://lld.llvm.org*.
- Visit the ARM developer website at *https://developer.arm.com* for more on ARM CPUs.
- Wikipedia offers an explanation of address space layout randomization at *https://en.wikipedia.org/wiki/Address_space_layout_randomization*.
- To better understand position-independent executables, see *https://en.wikipedia.org/wiki/Position-independent_code*.
- For information on the `:lo12:` operator, see the "Assembly Expressions" section in the document downloadable from *https://developer.arm.com/documentation/100067/0612/armclang-Integrated-Assembler*.

TEST YOURSELF

1. The PC-relative addressing mode indexes off which 64-bit register?
2. What does *opcode* stand for?
3. What type of data is the PC-relative addressing mode typically used for?
4. What is the address range of the PC-relative addressing mode?
5. In a register-indirect addressing mode, what does the register contain?
6. Which of the following registers is valid for use with the register-indirect addressing mode?
   a. W0
   b. X0
   c. XZR
   d. SP
7. What instruction would you normally use to load the address of a memory object into a register?
8. What is an effective address?
9. How would you align a variable in the `.data` section to an 8-byte boundary?
10. What does *MMU* stand for?
11. What is an address expression?
12. What is the difference between a big-endian value and a little-endian value?
13. If W0 contains a 32-bit big-endian value, what instruction could you use to convert it to a little-endian value?
14. If W0 contains a 16-bit little-endian value, what instruction could you use to convert it to a big-endian value?
15. If X0 contains a 64-bit big-endian value, what instruction could you use to convert it to a little-endian value?
16. Explain, step by step, what the `str x0, [sp, #-16]!` instruction does.
17. Explain, step by step, what the `ldr x0, [sp], #16` instruction does.
18. When using the push and pop operations to preserve registers, you must always pop the registers in the _____ order that you pushed them.
19. What does *LIFO* stand for?

4

CONSTANTS, VARIABLES, AND DATA TYPES



Chapter 2 discussed the basic format for data in memory, and Chapter 3 covered how a computer system physically organizes that data in memory. This chapter completes that discussion by connecting the concept of *data representation* to its actual physical representation. I'll focus on three main topics: constants, variables, and data structures.

This chapter doesn't assume you've taken a formal course in data structures, though such experience would be useful. You'll learn to declare and use constants, scalar variables, integers, data types, pointers, arrays, structs, and unions. Work to master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seem to present a multitude of problems to beginning assembly language programmers, but the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you'll pick it up as needed later; you'll need to comprehensively understand it right away.

4.1 Gas Constant Declarations

Probably the first place to start is with constant declarations that allow you to attach a name to a literal constant value. Gas provides four directives, collectively known as *equates*, that let you define constants in your assembly language programs. You've already seen the most common form, `.equ`:

```
.equ symbol, constantExpression
```

For example:

```
.equ MaxIndex, 15
```

Once you declare a symbolic constant in this manner, you may use the symbolic identifier anywhere the corresponding literal constant is legal. These constants are known as *manifest constants*—symbolic representations that allow you to substitute the literal value for the symbol anywhere in the program.

NOTE *Technically, you could also use CPP macros to define constants in Gas. See Chapter 13 for more details.*

Contrast this with `.rodata` objects: an `.rodata` value is a constant value, because you cannot change it at runtime. However, a memory location is associated with an `.rodata` declaration, and the OS, not the Gas assembler, enforces the read-only attribute. Although the following instruction sequence will crash your program when it runs, writing it is perfectly legal:

```
lea  x0, ReadOnlyVar
str  x1, [x0]
```

On the other hand, it is no more legal to write the following, using the preceding declaration

```
str  x1, MaxIndex
```

than it is to write this:

```
str  x1, #15
```

In fact, both statements are equivalent: the compiler substitutes `15` for `MaxIndex` whenever it encounters this manifest constant.

Constant declarations are great for defining magic numbers that could change during program modification. Examples include constants like `nl` (newline), `maxLen`, and NULL.

The GNU `.set` directive uses the following syntax:

```
.set label, expression
```

This is semantically equivalent to the following:

```
label = expression
```

Both the .set and = directives allow you to redefine a symbol previously defined with these directives.

For example:

```
maxLen = 10

At this point in the code, Gas will replace maxLen with 10.

maxLen = 256

In this section of the code, maxLen gets replaced by 256.
```

You'll see how to take advantage of this feature in Chapter 13, which discusses macros and the Gas compile-time language.

Note that .equ also allows you to redefine symbols in your source file. These many synonyms for the same directive are Gas's attempt to maintain compatibility with multiple assemblers and assembler versions.

The final equate directive Gas offers is .equiv:

```
.equiv symbol, expression
```

Unlike the other three directives, .equiv will generate an error if the symbol is already defined. This is therefore likely the safest equate to use, unless you really need to redefine symbols in your program.

Expressions appearing in these equates are limited to 64 bits. If you specify a value greater than 64 bits, the assembler will report an error.

4.2 The Location Counter Operator

One very special constant you'll frequently use is the current location counter value. As noted in the previous chapter, Gas will substitute the value of the current section's location counter in place of an individual period (.) appearing in a constant expression. You could in theory use this operator to embed a pointer to a variable within that variable itself:

```
ptrVar:  .dword  .   // Stores the address of ptrVar in ptrVar
```

However, this isn't especially useful. It's a better idea to use the location counter operator to compute offsets and lengths within a particular section. If you subtract a label in a section from the location counter, the difference is the (signed) distance from that point in the code to the specified label. This allows you to compute string lengths, function lengths, and other values that involve measuring the byte distance within a section.

Here's an example that uses this technique to compute a string length:

```
someStr:  .ascii  "Who wants to manually count the characters"
          .asciz  "in this string to determine its length?"
ssLen     =       .-someStr
```

This counts all the bytes Gas emits (including the zero-terminating byte) by the two string directives. You can use this technique to compute the length of any data object, not just the characters in a string.

Intuitively, there is a subtle difference between the location counter constant (`.`) and a literal constant such as `0`. The constant `0` will always have the same value wherever it appears in the source file, whereas the location counter constant will have a different value through the source file. An HLL would associate a different type with these two types of constants. The next sections discuss types in assembly language, including relocatable types (the location counter is a relocatable type in assembly language).

4.3 Data Types and Gas

Like most traditional (that is, 1960s-era) assemblers, Gas is completely *typeless*. It relies on you, the programmer, to make sense of all the data types you use in your program, via your choice of instructions. In particular, Gas will be more than happy to accept any of the following statements:

```
    .text
    .align 2
wv: .word  0
      .
      .
      .
    ldr   w0, wv  // Yes, this one's "type correct."
    ldr   x0, wv  // Loads more data than is present
```

The second instruction loads 64 bits from a 32-bit variable. However, Gas accepts this erroneous code and loads the 64 bits at the address you specify, which might include the 32 bits just beyond the `wv` declaration you've placed in your `.text` section.

Accessing data by using the wrong data type can lead to subtle defects within your code. One advantage of (strongly typed) HLLs is that they can catch most program errors resulting from the misuse of data types. Assembly language, however, provides very little in the way of type checking. Type checking is *your* responsibility in assembly language. Section 4.4, "Pointer Data Types," covers this issue next in great detail. Also see "Relocatable and Absolute Expressions" on page 176, which describes one of the few cases where Gas provides a small amount of type checking on your code.

4.4 Pointer Data Types

If you had a bad experience when you first encountered pointers in an HLL, fear not: pointers are easier to deal with in assembly language. Any problems you had with pointers probably had more to do with the linked-list and tree data structures you were trying to implement with them. Pointers, on the other hand, have many uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and structs often involve the use of pointers.

A *pointer* is a memory location whose value is the address of another memory location. Unfortunately, HLLs like C/C++ tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity tends to frighten programmers because they don't understand what's going on behind the scenes.

To illuminate how pointers work, consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is simple. `M` is an array with 1,024 integers in it, indexed from `M[0]` to `M[1023]`. Each one of these array elements can hold an integer value independent of the others. In other words, this array gives you 1,024 integer variables, each of which you refer to by number (the array index).

It's easy to see that the statement `M[0]:=100;` is storing the value `100` into the first element of the array `M`. The following two statements perform an identical operation:

```
i := 0; (* Assume "i" is an integer variable. *)
M [i] := 100;
```

Indeed, you can use any integer expression in the range 0 to 1,023 as an index into this array. The following statements still perform the same operation as our single assignment to index 0:

```
i := 5;      (* Assume all variables are integers. *)
j := 10;
k := 50;
M [i*j-k] := 100;
```

"Okay, so what's the point?" you're probably thinking. "Anything that produces an integer in the range 0 to 1,023 is legal. So what?" Consider the following code that adds an interesting layer of indirection:

```
M [1] := 0;
M [ M [1] ] := 100;
```

With a little thought, you should see that these two instructions perform the exact same operation as the previous examples. The first statement stores 0 into array element M[1]. The second statement fetches the value of M[1], a legal array index, and uses that value (0) to control where it stores the value 100.

If you're willing to accept this as reasonable, you'll have no problems with pointers. If you were to change M to *memory* and imagine that this array represents system memory, then M[1] is a pointer: that is, a memory location whose value is the address (or index) of another memory location. Pointers are easy to declare and use in an assembly language program; you don't even have to worry about array indices.

Okay, this section has used a Pascal array as an example of a pointer, which is fine, but how do you use pointers in an ARM assembly language program?

4.4.1 Pointer Usage in Assembly Language

An ARM64 pointer is a 64-bit value that may contain the address of another variable. For a dword variable p that contains 0x1000_0000, p "points" at memory location 0x1000_0000. To access the dword that p points at, you could use code like the following:

```
lea x0, p      // Load X0 with the
ldr x0, [x0]   // value of pointer p.
ldr x1, [x0]   // Fetch the data at which p points.
```

By loading the value of p into X0, this code loads the value 0x1000_0000 into X0 (assuming p contains 0x1000_0000). The second instruction loads the X1 register with the dword starting at the location whose offset appears in X0. Because X0 now contains 0x1000_0000, this will load X1 from locations 0x1000_0000 through 0x1000_0007.

Why not just load X1 directly from location 0x1000_0000, like this?

```
lea x1, varAtAddress1000_0000
ldr x1, [x1]
```

The primary reason not to do so is that this ldr instruction always loads X1 from location varAtAddress1000_0000. You cannot change the address from where it loads X1.

The former instructions, however, always load X1 from the location where p is pointing. This is easy to change under program control. Consider the following pseudocode instruction sequence:

```
    lea x0, i
    lea x1, p      // Set p = address of i.
    str x0, [x1]

Some code that sets or clears the carry flag ...

    bcc skipSetp
```

```
        lea x0, j
        lea x1, p   // Set p = address of j.
        str x0, [x1]
         .
         .
         .

skipSetp:           // Assume both code paths wind up
    lea x0, p       // down here.
    ldr x0, [x0]    // Load p into X0.
    ldr x1, [x0]    // X1 = i or j, depending on path here.
```

This short example demonstrates two execution paths through the program. The first path loads the variable `p` with the address of the variable `i`. The second path through the code loads `p` with the address of the variable `j`. Both execution paths converge on the last two `ldr` instructions that load X1 with `i` or `j`, depending on which execution path was taken. In many respects, this is like a parameter to a procedure in an HLL like Swift. Executing the same instructions accesses different variables depending on whose address (`i` or `j`) winds up in `p`.

4.4.2 Pointer Declarations in Gas

Because pointers are 64 bits long, you could use the `.dword` directive to allocate storage for your pointers:

```
            .data
bb:         .byte   .-.  // Uninitialized
            .align  3
d:          .dword  .-.  // Uninitialized
pByteVar:   .dword  bb   // Initialized with the address of bb
pDWordVar: .dword   d    // Initialized with the address of d
```

This example demonstrates that it is possible to initialize as well as declare pointer variables in Gas. You may specify addresses of static variables (`.data`, `.rodata`, and `.bss` objects) in the operand field of a `.dword` directive, so you can initialize only pointer variables with the addresses of static objects by using this technique.

Remember that macOS does not allow you to take the address of a symbol in the `.text` section because of the limitation of PIE code.

4.4.3 Pointer Constants and Expressions

Gas allows very simple constant expressions wherever a pointer constant is legal. Pointer constant expressions take one of the following forms:

```
StaticVarName + PureConstantExpression
StaticVarName - PureConstantExpression
```

The *PureConstantExpression* term is a numeric constant expression that does not involve any pointer constants (an *absolute constant*, using Gas terminology). This type of expression produces a memory address that is the specified number of bytes before or after (- or +, respectively) the *StaticVarName* variable in memory. The first two forms shown here are semantically equivalent: both return a pointer constant whose address is the sum of the static variable and the constant expression.

RELOCATABLE AND ABSOLUTE EXPRESSIONS

Gas divides constant expressions into two categories: relocatable and absolute. *Absolute expressions* are those that Gas can evaluate to a numeric value during assembly. Examples include the following:

```
5    8 + 2 * 3    (8 * 2) - 45    'A'    0xFFFF + 1    0xFFFE & 0xABCD
```

Relocatable expressions, on the other hand, involve symbolic names that reference memory locations in various sections of the program.

Expressions can have a mixture of absolute and relocatable components. The class of the resulting expression (that is, relocatable or absolute) depends on a few simple rules. If *R* is a relocatable expression (for example, a single symbol) and *A* is an absolute expression, then:

- $R + A$ is also a relocatable expression.
- $R - A$ is also a relocatable expression.
- $R_1 - R_2$ is an absolute expression (both R_1 and R_2 must be in the same section of memory).
- $R_1 + R_2$ is illegal.

Since you can create pointer constant expressions, it should come as no surprise that Gas lets you define manifest pointer constants by using equates. Listing 4-1 demonstrates how to do this.

```
// Listing4-1.S
//
// Pointer constant demonstration

#include "aoaa.inc"

          .section  .rodata, ""
ttlStr:   .asciz    "Listing 4-1"
fmtStr:   .ascii    "pb's value is %p\n"
          .asciz    "*pb's value is %d\n"

          .data
bb:       .byte     0
          .byte     1, 2, 3, 4, 5, 6, 7
```

```
❶ pb        =         bb + 2     // Address of "2" in bb

❷ pbVar:    .dword    pb

  pbValue:  .word     0

            .text
            .align    2
            .extern   printf

  // Return program title to C++ program:

            .global   getTitle
  getTitle:
            lea       x0, ttlStr
            ret

  // Here is the asmMain function:

           .global    asmMain
  asmMain:
           sub     sp, sp, #64     // Reserve space on stack.
           str     lr, [sp, #56]   // Save return address.

           lea     x0, pbVar       // Get pbVar.
           ldr     x0, [x0]
           ldrb    w0, [x0]        // Fetch data at *pbVar.
         ❸ lea     x1, pbValue     // Save in pbValue for now.
           str     w0, [x1]

  // Print the results:

           lea     x0, fmtStr
         ❹ vparm2  pbVar
         ❺ vparm3  pbValue
           bl      printf

           ldr     lr, [sp, #56]   // Restore return address.
           add     sp, sp, #64
           ret     // Returns to caller
```

The equate `pb = bb + 2` initializes the constant `pb` with the address of the third element ❶ (index 2) of the `bb` array. The `pbVar: .dword pb` declaration ❷ creates a pointer variable (named `pbVar`) and initializes with the value of the `pb` constant. Because `pb` is the address of `bb[2]`, this statement initializes `pbVar` with the address of `bb[2]`. The program stores the value held in `pbVar` into the `pbValue` variable ❸, then passes `pbVar` ❹ and `pbValue` ❺ to `printf()` to print their values.

Here's the `build` command and sample output:

```
$ ./build Listing4-1
$ ./Listing4-1
Calling Listing4-1:
pb's value is 0x411042
```

```
*pb's value is 2
Listing4-1 terminated
```

The address that's printed may vary on different machines and OSes.

4.4.4 Pointer Variables and Dynamic Memory Allocation

Pointer variables are the perfect place to store the return result from the C stdlib `malloc()` function. This function returns the address of the storage it allocates in the X0 register; therefore, you can store the address directly into a pointer variable immediately after a call to `malloc()`. Listing 4-2 demonstrates calls to the C stdlib `malloc()` and `free()` functions.

```
// Listing4-2.S
//
// Demonstration of calls
// to C stdlib malloc
// and free functions

#include "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 4-2"
fmtStr:     .asciz      "Addresses returned by malloc: %p, %p\n"

            .data
ptrVar:     .dword      .-.
ptrVar2:    .dword      .-.

            .text
            .align      2
            .extern     printf
            .extern     malloc
            .extern     free

// Return program title to C++ program:

            .global     getTitle
getTitle:
            lea         x0, ttlStr
            ret

// Here is the "asmMain" function:

            .global     asmMain
asmMain:
            sub         sp, sp, #64     // Space on stack
            str         lr, [sp, #56]   // Save return address.

// C stdlib malloc function
//
// ptr = malloc( byteCnt );
```

```
//
// Note: malloc has only a single parameter; it
// is passed in X0 as per ARM/macOS ABI.

        ❶ mov         x0, #256        // Allocate 256 bytes.
          bl          malloc
          lea         x1, ptrVar      // Store pointer into
          str         x0, [x1]        // ptrVar variable.

          mov         x0, #1024       // Allocate 1,024 bytes.
          bl          malloc
          lea         x1, ptrVar2     // Store pointer into
          str         x0, [x1]        // ptrVar2 variable.

// Print the addresses of the two malloc'd blocks:

          lea         x0, fmtStr
          vparm2      ptrVar
          vparm3      ptrVar2
          bl          printf

// Free the storage by calling
// C stdlib free function.
//
// free( ptrToFree );
//
// Once again, the single parameter gets passed in X0.

        ❷ lea         x0, ptrVar
          ldr         x0, [x0]
          bl          free

          lea         x0, ptrVar2
          ldr         x0, [x0]
          bl          free

          ldr         lr, [sp, #56]   // Get return address.
          add         sp, sp, #64     // Clean up stack.
          ret
```

Because `malloc()` ❶ and `free()` ❷ have only a single argument, you pass those arguments to them in the X0 register. For the call to `malloc()`, you pass an integer value specifying the amount of storage you want to allocate on the heap. For `free()`, you pass the pointer to the storage (previously allocated by `malloc()`) that you want to return back to the system.

Here's the `build` command and sample output:

```
$ ./build Listing4-2
$ ./Listing4-2
Calling Listing4-2:
Addresses returned by malloc: 0x240b46b0, 0x240b47c0
Listing4-2 terminated
```

As usual, the addresses you get will vary by OS and perhaps even by different runs of the program.

4.4.5 Common Pointer Problems

In most programming languages, programmers encounter five common problems. Some of these errors will cause your programs to immediately stop with a diagnostic message; other problems are subtler, yielding incorrect results or simply affecting the performance of your program without otherwise reporting an error. These five problems are as follows:

- Using an uninitialized pointer (illegal memory access)
- Using a pointer that contains an illegal value (for example, NULL)
- Continuing to use `malloc()`'d storage after that storage has been freed
- Failing to `free()` storage once the program is finished using it
- Accessing indirect data using the wrong data type

The following subsections describe each of these problems, their effects, and how to avoid them.

4.4.5.1 Illegal Memory Access Due to an Uninitialized Pointer

Beginning programmers often don't realize that declaring a pointer variable reserves storage only for the pointer itself; it does not reserve storage for the data that the pointer references. Therefore, you'll run into problems if you attempt to dereference a pointer that does not contain the address of a valid memory location. Listing 4-3 demonstrates this problem (don't try to compile and run this program; it will crash).

```
// Listing4-3.S
//
// Uninitialized pointer demonstration
// This program will not run properly.

#include "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 4-3"
fmtStr:     .asciz      "Pointer value= %p\n"

            .data
❶ ptrVar:   .dword      .-.   // ".-." means uninitialized.

            .text
            .align      2
            .extern     printf

// Return program title to C++ program:

            .global     getTitle
getTitle:
```

```
            lea         x0, ttlStr
            ret

// Here is the "asmMain" function:

            .global     asmMain
asmMain:
            sub         sp, sp, #64     // Stack storage
            str         lr, [sp, #56]   // Save return address.

          ❷ lea         x0, ptrVar
            ldr         x1, [x0]        // Get ptrVar into X1.
            ldr         x2, [x1]        // Will crash the system

            ldr         lr, [sp, #56]   // Retrieve return adrs.
            add         sp, sp, #64     // Restore stack.
            ret
```

Although variables you declare in the `.data` section are, technically, initialized, static initialization still doesn't initialize the pointer in this program ❶ with a valid address (but instead with a 0, which is NULL).

Of course, there is no such thing as a truly uninitialized variable on the ARM. There are variables that you've explicitly given an initial value, and there are variables that happen to inherit whatever bit pattern was in memory when storage for the variable was allocated. Much of the time, these garbage bit patterns don't correspond to a valid memory address. Attempting to *dereference* such a pointer (that is, access the data in memory at which it points ❷) typically raises a *memory access violation* exception (segmentation fault).

Sometimes, however, those random bits in memory just happen to correspond to a valid memory location you can access. In this situation, the CPU will access the specified memory location without aborting the program. Although to a naive programmer this situation may seem preferable to stopping the program, in reality this is far worse, since your defective program continues to run without alerting you to the problem. If you store data through an uninitialized pointer, you may overwrite the values of other important variables in memory. This defect can produce some difficult-to-locate problems in your program.

4.4.5.2 Invalid Addresses

The second common problem is storing invalid address values into a pointer. The previous problem is actually a special case of this second problem (with garbage bits in memory supplying the invalid address, rather than you producing it via a miscalculation). The effects are the same: if you attempt to dereference a pointer containing an invalid address, you either will get a memory access violation exception or will access an unexpected memory location.

4.4.5.3 The Dangling Pointer Problem

The third problem, continuing to use `malloc()`'d storage after that storage has been freed, is also known as the *dangling pointer problem*. To understand this problem, consider the following code fragment:

```
mov  x0, #256
bl   malloc       // Allocate some storage.
lea  x1, ptrVar
str  x0, [x1]     // Save address away in ptrVar.
 .
 .     Code that uses the pointer variable ptrVar
 .
lea  x0, ptrVar   // Pass ptrVar's value to free.
ldr  x0, [x0]
bl   free         // Free storage associated with ptrVar.
 .
 .    Code that does not change the value in ptrVar
 .
lea  x0, ptrVar
ldr  x1, [x0]
strb w2, [x1]
```

This code allocates 256 bytes of storage and saves the address of that storage in the `ptrVar` variable. It then uses this block of 256 bytes for a while and frees the storage, returning it to the system for other uses.

Calling `free()` does not change the value of `ptrVar` in any way; `ptrVar` still points at the block of memory allocated by `malloc()` earlier. The value in `ptrVar` is a *dangling pointer*, or *wild pointer*—a pointer that is pointing at deallocated storage. In this example, `free()` does not change any data in the block allocated by `malloc()`, so upon return from `free()`, `ptrVar` still points at the data stored into the block by this code. However, the call to `free()` tells the system that the program no longer needs this 256-byte block of memory, so the system can use this region of memory for other purposes.

The `free()` function cannot enforce the fact that you will never access this data again; you are simply promising that you won't. Of course, the preceding code fragment breaks this promise; as you can see in the last three instructions, the program fetches the value in `ptrVar` and accesses the data it points at in memory.

The biggest problem with dangling pointers is that you can often get away with using them. As long as the system doesn't reuse the storage you've freed, a dangling pointer produces no ill effects. However, with each new call to `malloc()`, the system may decide to reuse the memory released by that previous call to `free()`. When this happens, any attempt to dereference the dangling pointer may produce unintended consequences. The problems range from reading data that has been overwritten (by the new, legal use of the data storage), to overwriting the new data, to, in the worst case, overwriting system heap management pointers and likely crashing your program. The solution is clear: *never use a pointer value after you free the storage associated with that pointer.*

4.4.5.4 Memory Leaks

Of all the pointer problems listed at the beginning of this section, failing to free allocated storage will probably have the least negative impact. The following code fragment demonstrates this problem:

```
mov  x0, #256
bl   malloc
lea  x1, ptrVar
str  x0, [x1]

Code that uses ptrVar
This code does not free up the storage
associated with ptrVar.

mov  x0, #512
bl   malloc
lea  x1, ptrVar
str  x0, [x1]

// At this point, there is no way to reference the original
// block of 256 bytes pointed at by ptrVar.
```

In this example, the program allocates 256 bytes of storage and references it by using the `ptrVar` variable. Later, the program allocates another block of bytes and overwrites the value in `ptrVar` with the address of this new block. The former value in `ptrVar` is lost. Because the program no longer has this address value, there is no way to call `free()` to return the storage for later use.

As a result, these 256 bytes of memory are no longer available to your program. While this may seem like only a minor cost, imagine that this code is in a repeating loop. With each execution of the loop, the program loses another 256 bytes of memory, eventually exhausting the memory available on the heap. This problem is often called a *memory leak*, because it's as though the memory bits are leaking out of your computer during program execution.

Memory leaks are far less damaging than dangling pointers. They create only two problems: the danger of running out of heap space (which ultimately may cause the program to abort, though this is rare) and performance problems due to virtual memory page swapping. Nevertheless, you should get in the habit of always freeing all storage after you have finished using it. When your program quits, the OS reclaims all storage, including the data lost via memory leaks. Therefore, memory lost via a leak is lost only to your program, not to the whole system.

4.4.5.5 Lack of Type-Safe Access

Because Gas cannot and does not enforce pointer type checking, you can load the address of a data structure into a register and access that data as though it were a completely different type (often resulting in logic errors in your program). For example, consider Listing 4-4.

```
// Listing4-4.S
//
// Demonstration of lack of type
// checking in assembly language
// pointer access

#include "aoaa.inc"

maxLen       =          256

             .section    .rodata, ""
ttlStr:      .asciz      "Listing 4-4"
prompt:      .asciz      "Input a string: "
fmtStr:      .asciz      "%d: Hex value of char read: %x\n"

             .data
valToPrint: .word        .-.
bufIndex:    .dword      .-.
bufPtr:      .dword      .-.
bytesRead:   .dword      .-.

             .text
             .align      2
             .extern     readLine
             .extern     printf
             .extern     malloc
             .extern     free

// Return program title to C++ program:

             .global     getTitle
getTitle:
             lea         x0, ttlStr
             ret

// Here is the asmMain function:

             .global     asmMain
asmMain:

             sub     sp, sp, #64     // Reserve stack space.
             str     lr, [sp, #56]   // Save return address.

// C stdlib malloc function
// Allocate sufficient characters
// to hold a line of text input
// by the user:

             mov     x0, #maxLen     // Allocate 256 bytes.
             bl      malloc
             lea     x1, bufPtr      // Save pointer to buffer.
             str     x0, [x1]
```

```
// Read a line of text from the user and place in
// the newly allocated buffer:

        lea     x0, prompt      // Prompt user to input
        bl      printf          // a line of text.

        lea     x0, bufPtr
        ldr     x0, [x0]        // Pointer to input buffer
        mov     x1, #maxLen     // Maximum input buffer length
        bl      readLine        // Read text from user.
        cmp     x0, #-1         // Skip output if error.
        beq     allDone
        lea     x1, bytesRead
        str     x0, [x1]        // Save number of chars read.

// Display the data input by the user:

        mov     x1, #0          // Set index to 0.
        lea     x0, bufIndex
        str     x1, [x0]
displp: lea     x0, bufIndex    // Get buffer index
        ldr     x1, [x0]        // into X1.
        lea     x2, bufPtr      // Get pointer to buffer.
        ldr     x2, [x2]
        ldr     w0, [x2, x1]    // Read word rather than byte!
        lea     x1, valToPrint
        str     w0, [x1]
        lea     x0, fmtStr
        vparm2  bufIndex
        vparm3  valToPrint
        bl      printf

        lea     x0, bufIndex    // Increment index by 1.
        ldr     x1, [x0]
        add     x1, x1, #1
        str     x1, [x0]

        lea     x0, bytesRead   // Repeat until
        ldr     x0, [x0]        // you've processed "bytesRead"
        cmp     x1, x0          // bytes.
        blo     displp

// Free the storage by calling
// C stdlib free function.
//
// free( bufPtr )

allDone:
        lea     x0, bufPtr
        ldr     x0, [x0]
        bl      free

        ldr     lr, [sp, #56]   // Restore return address.
        add     sp, sp, #64
        ret     // Returns to caller
```

Here are the commands to build and run the program in Listing 4-4:

```
$ ./build Listing4-4
$ ./Listing4-4
Calling Listing4-4:
Input a string: Hello world
0: Hex value of char read: 6c6c6548
1: Hex value of char read: 6f6c6c65
2: Hex value of char read: 206f6c6c
3: Hex value of char read: 77206f6c
4: Hex value of char read: 6f77206f
5: Hex value of char read: 726f7720
6: Hex value of char read: 6c726f77
7: Hex value of char read: 646c726f
8: Hex value of char read: 646c72
9: Hex value of char read: 646c
10: Hex value of char read: 64
11: Hex value of char read: 0
Listing4-4 terminated
```

Listing 4-4 reads data from the user as character values and then displays the data as double-word hexadecimal values. While assembly language lets you ignore data types at will and automatically coerce the data without any effort, this power is a double-edged sword. If you make a mistake and access indirect data by using the wrong data type, Gas and the ARM may not catch the mistake and your program may produce inaccurate results. Therefore, you need to ensure that you use data consistently with respect to data type when working with pointers and indirection in your programs.

This demonstration program has one fundamental flaw that could create a problem for you: when reading the last two characters of the input buffer, the program accesses data beyond the characters input by the user. If the user inputs 255 characters (plus the zero-terminating byte that `readLine()` appends), this program will access data beyond the end of the buffer allocated by `malloc()`. In theory, this could cause the program to crash. This is yet another problem that can occur when accessing data by using the wrong type via pointers.

Despite all the problems that pointers suffer from, they are essential for accessing common data structures such as arrays, structs, and strings. That's why this chapter discussed pointers prior to these other *composite data types*. However, with the discussion of pointers out of the way, it's time to look at those other data types.

4.5 Composite Data Types

Composite data types, also known as *aggregate data types*, are those that are built up from other, generally scalar, data types. A string, for example, is a composite data type, since it's built from a sequence of individual characters and other data. The following sections cover several of the more

important composite data types: character strings, arrays, multidimensional arrays, structs, and unions.

4.6 Character Strings

After integer values, character strings are probably the most common data type that modern programs use. This section provides a couple definitions of character strings (the ubiquitous zero-terminated string, the more efficient length-prefixed string, and other combinations of the two) and discusses how to process those strings.

In general, a *character string* is a sequence of ASCII characters that possesses two main attributes: a length and character data. Different languages use different data structures to represent strings. For assembly language (at least, sans any library routines), you can choose to implement strings in whichever format you want—perhaps based on the format's compatibility with an HLL or on a desire to produce faster string functions. All you need do is create a sequence of machine instructions to process the string data in whatever format the strings take.

It's also possible for strings to hold Unicode characters. This section uses ASCII in all the examples (because Gas does a better job of supporting ASCII characters). The principles apply to Unicode as well, with an appropriate extension in the amount of storage you use.

4.6.1 Zero-Terminated Strings

Zero-terminated strings are the most common string representation in use today, since this is the native string format for C, C++, and other languages. A *zero-terminated string* consists of a sequence of zero or more ASCII characters ending with a 0 byte. For example, in C/C++, the string `"abc"` requires 4 bytes: the three characters `a`, `b`, and `c`, followed by a byte containing 0.

To create zero-terminated strings in Gas, simply use the `.asciz` directive. The easiest place to do this is in the `.data` section, using code like the following:

```
            .data
zeroString: .asciz  "This is the zero-terminated string"
```

Whenever a character string appears in the `.asciz` directive, as it does here, Gas emits each character in the string to successive memory locations and terminates the whole string with a 0 byte.

There are a couple of ways to accommodate a zero-terminated string that's longer than a single source line. First, you can use `.ascii` directives for all but the last source code line in a long string. For example:

```
            .data
longZString: .ascii  "This is the first line"
             .ascii  "This is the second line"
             .asciz  "This is the last line"
```

The .asciz directive zero-terminates the entire string. However, if you prefer, you can always use a .byte directive to explicitly add the zero-terminating byte yourself:

```
            .data
longZString: .ascii  "This is the first line"
            .ascii  "This is the second line"
            .ascii  "This is the last line"
            .byte   0
```

Use whichever scheme you like. Some people prefer the explicit .byte directive because it's easy to add and remove strings from the list without having to worry about changing .ascii to .asciz (or vice versa).

Zero-terminated strings have two principal attributes: they are simple to implement, and the strings can be any length. However, they also have a few drawbacks. First, zero-terminated strings cannot contain the NUL character (whose ASCII code is 0). Generally, this isn't a problem, but it does create havoc once in a while. Second, many operations on zero-terminated strings are somewhat inefficient. For example, to compute the length of a zero-terminated string, you must scan the entire string looking for that 0 byte (counting characters up to the 0). The following program fragment demonstrates how to compute the length of the preceding string:

```
          lea   x1, longZString
          mov   x2, x1          // Save pointer to string.
whileLp:  ldrb  w0, [x1], #1    // Fetch next char and inc X1.
          cmp   w0, #0          // See if 0 byte.
          bne   whileLp         // Repeat while not 0.
          sub   x0, x1, x2      // X0 = X1 - X2
          sub   x0, x0, #1      // Adjust for extra increment.

// String length is now in X0.
```

This code saves the initial string address (in X2), then subtracts the final pointer (just beyond the 0 byte) from the initial address to compute the length. The extra sub (by 1) is present because we don't normally include the zero-terminating byte in the string's length.

As you can see, the time it takes to compute the length of the string is proportional to the length of the string; as the string gets longer, computing its length takes longer.

4.6.2 Length-Prefixed Strings

The length-prefixed string format overcomes some of the problems with zero-terminated strings. *Length-prefixed strings* are common in languages like Pascal; they generally consist of a length byte followed by zero or more character values. The first byte specifies the string length, and the following bytes (up to the specified length) are the character data. In a length-prefixed scheme, the string "abc" would consist of 4 bytes: 3 (the string length)

followed by a, b, and c. You can create length-prefixed strings in Gas by using code like the following:

```
                        .data
lengthPrefixedString: .byte   3
                        .ascii "abc"
```

Counting the characters ahead of time and inserting them into the byte statement, as was done here, may seem like a major pain. Fortunately, there are ways to have Gas automatically compute the string length for you.

Length-prefixed strings solve the two major problems associated with zero-terminated strings. It is possible to include the NUL character in length-prefixed strings, and those operations on zero-terminated strings that are relatively inefficient (for example, string length) are more efficient when using length-prefixed strings. However, length-prefixed strings have their own drawbacks; most important, they are limited to a maximum of 255 characters in length (assuming a 1-byte length prefix).

Of course, if you have a problem with a string-length limitation of 255 characters, you can always create a length-prefixed string by using any number of bytes for the length as you need. For example, the High-Level Assembler (HLA) uses a 4-byte length variant of length-prefixed strings, allowing strings up to 4GB long. (See section 4.11, "For More Information," on page 221 for more on the HLA.) In assembly language, you can define string formats however you like.

To create length-prefixed strings in your assembly language programs, you don't want to manually count the characters in the string and emit that length in your code. It's far better to have the assembler do this kind of grunt work for you by using the location counter operator (.), as follows:

```
                        .data
lengthPrefixedString: .byte  lpsLen
                        .ascii "abc"
lpsLen                  =      . - lengthPrefixedString - 1
```

The lpsLen operand subtracts 1 in the address expression because

```
. - lengthPrefixedString
```

also includes the length prefix byte, which isn't considered part of the string length.

Gas does not require you to define lpsLen before using it as the operand field in the .byte directive. Gas is smart enough to go back and fill in the value after it is defined in the equate statement.

4.6.3 String Descriptors

Another common string format is a string descriptor. A *string descriptor* is typically a small data structure (see section 4.8, "Structs," on page 212) that contains several pieces of data describing a string.

At a bare minimum, a string descriptor will probably have a pointer to the actual string data and a field specifying the number of characters in the string (that is, the string length). Other possible fields might include the number of bytes currently occupied by the string, the maximum number of bytes the string could occupy, the string encoding (for example, ASCII, Latin-1, UTF-8, or UTF-16), and any other information the string data structure's designer could dream up.

By far, the most common descriptor format incorporates a pointer to the string's data and a size field specifying the number of bytes currently occupied by that string data. Note that this particular string descriptor is not the same thing as a length-prefixed string. In a length-prefixed string, the length immediately precedes the character data itself. In a descriptor, the length and a pointer are kept together, and this pair is (usually) separate from the character data itself.

4.6.4 Pointers to Strings

Often, an assembly language program won't directly work with strings appearing in the `.data` (or `.text`, `.rodata`, or `.bss`) section. Instead, the program will work with pointers to strings (including strings whose storage the program has dynamically allocated with a call to a function like `malloc()`). Listing 4-4 provided a simple (though broken) example. In such applications, your assembly code will typically load a pointer to a string into a base register and then use a second (index) register to access individual characters in the string.

4.6.5 String Functions

Unfortunately, few assemblers provide a set of string functions you can call from your assembly language programs. As an assembly language programmer, you're expected to write these functions on your own. Fortunately, a couple of solutions are available if you don't quite feel up to the task.

The first set of string functions you can call, without having to write them yourself, are the C stdlib string functions from the *string.h* header file in C. Of course, you'll have to use C strings (zero-terminated strings) in your code when calling C stdlib functions, but this generally isn't a big problem. Listing 4-5 provides examples of calls to various C string functions, further described in Appendix E.

```
// Listing4-5.S
//
// Calling C stdlib string functions

#include "aoaa.inc"

maxLen      =           256
saveLR      =           56

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 4-5"
```

```
prompt:     .asciz      "Input a string: "
fmtStr1:    .asciz      "After strncpy, resultStr='%s'\n"
fmtStr2:    .asciz      "After strncat, resultStr='%s'\n"
fmtStr3:    .asciz      "After strcmp (3), WO=%d\n"
fmtStr4:    .asciz      "After strcmp (4), WO=%d\n"
fmtStr5:    .asciz      "After strcmp (5), WO=%d\n"
fmtStr6:    .asciz      "After strchr, XO='%s'\n"
fmtStr7:    .asciz      "After strstr, XO='%s'\n"
fmtStr8:    .asciz      "resultStr length is %d\n"

str1:       .asciz      "Hello, "
str2:       .asciz      "World!"
str3:       .asciz      "Hello, World!"
str4:       .asciz      "hello, world!"
str5:       .asciz      "HELLO, WORLD!"

            .data
strLength:  .dword      .-.
resultStr:  .space      maxLen, .-.
resultPtr:  .dword      resultStr
cmpResult:  .dword      .-.

            .text
            .align      2
            .extern     readLine
            .extern     printf
            .extern     malloc
            .extern     free

// Some C stdlib string functions:
//
// size_t strlen(char *str)

            .extern     strlen

// char *strncat(char *dest, const char *src, size_t n)

            .extern     strncat

// char *strchr(const char *str, int c)

            .extern     strchr

// int strcmp(const char *str1, const char *str2)

            .extern     strcmp

// char *strncpy(char *dest, const char *src, size_t n)

            .extern     strncpy

// char *strstr(const char *inStr, const char *search4)

            .extern     strstr
```

```
// Return program title to C++ program:

            .global     getTitle
getTitle:
            lea         x0, ttlStr
            ret

// Here is the "asmMain" function.

            .global     asmMain
asmMain:
        sub     sp, sp, #64         // Allocate stack space.
        str     lr, [sp, #saveLR]   // Save return address.

// Demonstrate the strncpy function to copy a
// string from one location to another:

        lea     x0, resultStr   // Destination string
        lea     x1, str1        // Source string
        mov     x2, #maxLen     // Max number of chars to copy
        bl      strncpy

        lea     x0, fmtStr1
        vparm2  resultPtr
        bl      printf

// Demonstrate the strncat function to concatenate str2 to
// the end of resultStr:

        lea     x0, resultStr
        lea     x1, str2
        mov     x2, #maxLen
        bl      strncat

        lea     x0, fmtStr2
        vparm2  resultPtr
        bl      printf

// Demonstrate the strcmp function to compare resultStr
// with str3, str4, and str5:

        lea     x0, resultStr
        lea     x1, str3
        bl      strcmp
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr3
        vparm2  cmpResult
        bl      printf

        lea     x0, resultStr
        lea     x1, str4
        bl      strcmp
```

```
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr4
        vparm2  cmpResult
        bl      printf

        lea     x0, resultStr
        lea     x1, str5
        bl      strcmp
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr5
        vparm2  cmpResult
        bl      printf

// Demonstrate the strchr function to search for
// ',' in resultStr:

        lea     x0, resultStr
        mov     x1, #','
        bl      strchr
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr6
        vparm2  cmpResult
        bl      printf

// Demonstrate the strstr function to search for
// str2 in resultStr:

        lea     x0, resultStr
        lea     x1, str2
        bl      strstr
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr7
        vparm2  cmpResult
        bl      printf

// Demonstrate a call to the strlen function:

        lea     x0, resultStr
        bl      strlen
        lea     x1, cmpResult
        str     x0, [x1]

        lea     x0, fmtStr8
        vparm2  cmpResult
        bl      printf
```

```
        ldr     lr, [sp, #saveLR]   // Restore return address.
        add     sp, sp, #64         // Deallocate storage.
        ret     // Returns to caller
```

Here's the `build` command and sample output from Listing 4-5:

```
$ ./build Listing4-5
$ ./Listing4-5
Calling Listing4-5:
After strncpy, resultStr='Hello, '
After strncat, resultStr='Hello, World!'
After strcmp (3), W0=0
After strcmp (4), W0=-128
After strcmp (5), W0=128
After strchr, X0=', World!'
After strstr, X0='World!'
resultStr length is 13
Listing4-5 terminated
```

Of course, you could make a good argument that if all your assembly code does is call a bunch of C stdlib functions, you should have written your application in C in the first place. Most of the benefits of writing code in assembly language happen only when you "think" in assembly language, not C.

In particular, you can dramatically improve the performance of your string function calls if you stop using zero-terminated strings and switch to another string format (such as length-prefixed or descriptor-based strings that include a length component). Chapter 14 presents some pure assembly string functions for those who want to avoid the inefficiencies of using zero-terminated strings with the C stdlib.

4.7 Arrays

Along with strings, arrays are probably the most commonly used composite data type. Yet most beginning programmers don't understand their internal operation or their associated efficiency trade-offs. It's surprising how many novice (and even advanced!) programmers view arrays from a completely different perspective once they learn how to deal with arrays at the machine level.

Abstractly, an *array* is an aggregate data type whose members (elements) are all the same type. Selection of a member from the array is by an integer index (or other ordinal type such as Boolean or character). Different indices select unique elements of the array. This book assumes that the integer indices are contiguous, though this is by no means required. That is, if the number x is a valid index into the array and y is also a valid index, with $x < y$, then all i such that $x < i < y$ are valid indices. Most HLLs use contiguous array indices, and they are the most efficient to use, hence their use here.

Whenever you apply the indexing operator to an array, the result is the specific array element chosen by that index. For example, `A[i]` chooses the *i*th element from array `A`. There is no formal requirement that element `i` be anywhere near element `i + 1` in memory; the definition of an array is satisfied as long as `A[i]` always refers to the same memory location and `A[i + 1]` always refers to its corresponding location (and the two are different).

As noted, this book assumes that array elements occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 4-1.

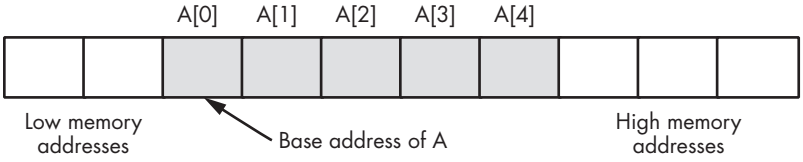


Figure 4-1: An array layout in memory

The *base address* of an array is the address of that array's first element and always appears in the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, and so on. Indices are not required to start at 0. They may start with any number as long as they are contiguous. However, for the purposes of discussion, this book starts all indices at 0.

To access an element of an array, you need a function that translates an array index to the address of the indexed element. For a single-dimensional array, this function is very simple:

```
Element_Address =
   Base_Address + ((Index - Initial_Index) × Element_Size)
```

Here, *`Initial_Index`* is the value of the first index in the array (which you can ignore if it's 0), and the value *`Element_Size`* is the size, in bytes, of an individual array element (this may include padding bytes used to keep elements properly aligned).

4.7.1 Declaring Arrays in Gas Programs

Before you can access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations you've already seen. To allocate *n* elements in an array, you would use a declaration like the following in one of the variable declaration sections:

```
ArrayName: .fill  n, element_size, initial_value
```

`ArrayName` is the name of the array variable, *`n`* is the number of array elements, *`element_size`* is the size (in bytes) of a single element, and *`initial_value`* is the initial value to assign to each array element. The *`element_size`* and *`initial_value`* arguments are optional, defaulting to `1` and `0`, respectively.

For example, to declare an array of sixteen 32-bit words, you could use the following:

```
wordArray:   .fill    16, 4
```

This would set aside sixteen 4-byte words, each initialized with 0 (the default initial value).

The value for *element_size* must not exceed 8; if it does, Gas will clip the value to 8. For historical (Gas) reasons, you should limit the initial value to 32 bits; larger values are transformed in nonintuitive ways (and differently on macOS and Linux). As a general rule, I strongly recommend defaulting to 0 for each array element when using the `.fill` directive.

NOTE *If you use the `.fill` directive in a `.bss` section, the initial value must be absent or set to 0.*

An alternative to the `.fill` directive is `.space`

```
ArrayName:  .space size, fill
```

where *size* is the number of bytes to allocate for the array and *fill* is an optional 8-bit value that Gas will use to initialize each byte of the array. If the *fill* argument is absent, Gas uses a default value of 0.

To declare an array of a type other than bytes, you must compute the size argument as *numberOfElements* × *elementSize*. For example, to create a 16-element array of words, you could use the following declaration:

```
wordArray:   .space    16 * (4)   // word wordArray[16]
```

Because the *fill* argument is not present, Gas will initialize this array with bytes containing 0s. I recommend putting parentheses around the element size in the expression to better document your intent; this differentiates the element size from the element count. As you'll see in section 4.7.4, "Implementing Multidimensional Arrays," on page 203, the element count could be an expression based on the size of each dimension.

To obtain the base address of these arrays, just use `ArrayName` or `wordArray` in an address expression. If you prefer to initialize an array with different values in each of the elements, you must manually supply those values in the directives `.byte`, `.hword`, `.word`, `.dword`, and so on. Here's a 16-word array initialized with the values 0 to 15:

```
wordArray:    .word   0, 1, 2, 3, 4, 5, 6, 7
              .word   8, 9, 10, 11, 12, 13, 14, 15
```

If you need to initialize a large array with different values, you're best off either writing an external program (perhaps in an HLL like C/C++) or using Gas's macro facilities to generate the array. I discuss this further in Chapters 10 and 13.

4.7.2 Accessing Elements of a Single-Dimensional Array

To access an element of a zero-based array, use this formula:

```
Element_Address = Base_Address + index × Element_Size
```

If the array is located within your `.text` section (an array of constants), or if you're writing a Linux application and the array isn't farther than ±1MB from your code that accesses the array, you can use the array's name for the *Base_Address* entry. This is because Gas associates the address of the first element of an array with the name of that array.

Otherwise, you'll need to load the base address of the array into a 64-bit register. For example:

```
lea x1, Base_Address
```

The *Element_Size* entry is the number of bytes for each array element. If the object is an array of bytes, the *Element_Size* field is 1 (resulting in a very simple computation). If each element of the array is a half word (or other 2-byte type), then *Element_Size* is 2, and so on. To access an element of the `wordArray` array in the previous section, you'd use the following formula (the size is 4 because each element is a word object):

```
Element_Address = wordArray + (index × 4)
```

The ARM code equivalent to the statement `w0 = wordArray[index]` is as follows:

```
lea x1, index   // Assume index is a 32-bit integer.
ldr w1, [x1]    // Get index into W1.
lea x2, wordArray
ldr w0, [x2, w1, uxtw #2] // index * 4 and zero-extended
```

This instruction sequence does not explicitly compute the sum of the base address plus the index times 4 (the size of a 32-bit integer element in `wordArray`). Instead, it relies on the scaled-indexed addressing mode (the `uxtx #2` operand) to implicitly compute this sum. The instruction

```
ldr w0, [x2, w1, uxtw #2]
```

loads W0 from location `X2 + W1 * 4`, which is the base address plus `index * 4` (because W1 contains `index`).

To multiply by a constant other than 1, 2, 4, or 8 (the immediate shift constants possible with the scaled-indexed addressing mode), you'll need to use the `lsl` instruction to multiply by the element size (if multiplying by a power of 2) or the `mul` instruction. You'll see some examples in a bit.

The scaled-indexed addressing mode on the ARM is the natural addressing mode for accessing elements of a single-dimensional array. Make sure

you remember to multiply the index by the size of an element; failure to do so will produce incorrect results.

The examples in this section assume that the `index` variable is a 32-bit value, which is common for array indices. To use a smaller integer, you'd need to sign- or zero-extend it to 32 bits. To use a 64-bit integer, simply adjust the scaled-indexed addressing mode to use a 64-bit index register and use the shift-left scaling without zero or sign extension.

4.7.3 Sorting an Array of Values

When introducing arrays, books commonly introduce sorting the elements of an array. To acknowledge this historical precedent, this section takes a quick look at a simple sort in Gas. The program presented in this section uses a variant of the bubble sort, which is great for short lists of data and lists that are nearly sorted, but horrible for just about everything else. However, a bubble sort is easy to implement and understand, which is why this and other introductory texts continue to use it in examples.

Because of the relative complexity of Listing 4-6, I'll insert comments throughout the source code rather than explaining it at the end. We begin by including *aoaa.inc*, as usual.

```
// Listing4-6.S
//
// A simple bubble sort example

#include "aoaa.inc"
```

Right away, let's make some coding improvements as compared to many of the previous examples in this book. Those examples, such as Listing 4-1, used "magic" numbers, like 64 for the amount of stack space to allocate and 56 for the offset into the stack allocation where I preserve the LR register. I used these literal constants directly in the code to be as transparent as possible; however, good programming style demands the use of symbolic names in place of those magic numbers. The two equates below accomplish this.

```
// Listing4-6.S (cont.)

stackAlloc  =   64   // Space to allocate on stack
saveLR      =   56   // Save LR here (index into stack frame).
```

The next couple of statements in the source file define offsets into the stack frame (allocated storage on the stack) where the program can preserve register values. In all the example programs so far, I've placed (global) variables in memory locations. That's not the appropriate paradigm for RISC assembly language programming.

The ARM ABI reserves registers X19 through X28 for use as nonvolatile (permanent) variable storage. *Nonvolatile* means you can call functions (like `printf()`) without worrying about those registers' values being changed. The

drawback to using nonvolatile registers is that you have to preserve their values upon entry into your code. The following two equates specify the offset into the stack allocation area for register preservation. This code will use registers X19 and X20 as loop control variables:

```
// Listing4-6.S (cont.)

x19Save     =   saveLR - 8   // Save X19 here.
x20Save     =   x19Save - 8  // Save X20 here.
```

The remaining equates define other constants used in this code:

```
// Listing4-6.S (cont.)

maxLen      =   256
true        =   1
false       =   0
```

Next come the usual read-only and writable data sections. In particular, the `.data` section contains the `sortMe` array, which will be the subject of the sorting operation. Also, this block of statements contains the `getTitle` function required by the *c.cpp* program:

```
// Listing4-6.S (cont.)

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 4-6"
fmtStr:     .asciz      "Sortme[%d] = %d\n"

            .data

// sortMe - A 16-element array to sort:

sortMe:
            .word   1, 2, 16, 14
            .word   3, 9, 4,  10
            .word   5, 7, 15, 12
            .word   8, 6, 11, 13
sortSize    =       (. - sortMe) / 4   // Number of elements

// Holds the array element during printing:

valToPrint: .word   .-.
i:          .word   .-.

            .text
            .align  2
            .extern printf

// Return program title to C++ program:

            .global getTitle
```

```
getTitle:
            lea     x0, ttlStr
            ret
```

Now we get to the bubble-sort function itself:

```
// Listing4-6.S (cont.)
//
// Here's the bubble-sort function.
//
//       sort( dword *array, qword count )
//
// Note: this is not an external (C)
// function, nor does it call any
// external functions, so it will
// dispense with some of the OS-calling-
// sequence stuff.
//
// array- Address passed in X0
// count- Element count passed in X1
//
// Locals:
//
// W2 is "didSwap" Boolean flag.
// X3 is index for outer loop.
// W4 is index for inner loop.
```

The bubble-sort function could just use register names like X0, X1, W2, and X3 for all the local variables. However, the following `#define` statements let you use more meaningful names. X5, X6, and X7 are pure temporaries (no meaningful name is attached to them), so this code continues to use the ARM register names for these temporary or local objects. Technically, X0 through X7 are reserved for parameters. As the `sort` function has only two parameters (`array` and `count`), it uses X2 through X7 as local variables (which is fine, as these registers are volatile, according to the ARM ABI):

```
// Listing4-6.S (cont.)

#define array   x0
#define count   x1
#define didSwap w2
#define index   x3
```

The `count` parameter just defined contains the number of array elements (which will be 16 in the main program). Since it's more convenient for this to be a byte count rather than a (word) element count, the following code multiplies `count` (X1) by 4, using a shift left by `2`. Also, the loop executes `count` –1 times, so this code also preps `count` by subtracting 1 from it:

```
// Listing4-6.S (cont.)

sort:
        sub     count, count, #1   // numElements - 1

        lsl     count, count, #2   // Make byte count.
```

The bubble sort works by making `count` – 1 passes through the array, where `count` is the number of elements. On each pass, it compares each adjacent pair of array elements; if the first element is greater than the second one, the program swaps them. At the end of each pass, one element winds up being moved to its final position. As an optimization, if no swaps occur, then all the elements are already in place, so the sort terminates:

```
// Listing4-6.S (cont.)
//
// Outer loop

outer:  mov     didSwap, #false

        mov     index, #0          // Outer loop index
inner:  cmp     index, count       // while outer < count - 1
        bhs     xInner

        add     x5, array, index   // W5 = &array[index]
        ldr     w6, [x5]           // W6 = array[index]
        ldr     w7, [x5, #4]       // W7 = array[index + 1]
        cmp     w6, w7             // If W5 > W
        bls     dontSwap           // then swap.

        // sortMe[index] > sortMe[index + 1], so swap elements.

        str     w6, [x5, #4]
        str     w7, [x5]
        mov     didSwap, #true

dontSwap:
        add     index, index, #4    // Next word
        b.al    inner

// Exited from inner loop, test for repeat
// of outer loop:

xInner: cmp     didSwap, #true
        beq     outer

        ret
```

The main program begins by preserving the nonvolatile registers (LR, X19, and X20) that it uses:

```
// Listing4-6.S (cont.)
//
// Here is the asmMain function:

            .global asmMain
asmMain:

            sub     sp, sp, #stackAlloc   // Allocate stack space.
            str     lr, [sp, #saveLR]     // Save return address.
            str     x19, [sp, #x19Save]   // Save nonvolatile
            str     x20, [sp, #x20Save]   // X19 and X20.
```

Next, the main program calls the `sort` function to sort the array. As per the ARM ABI, this program passes the first argument (the address of the array) in X0 and the second argument (element count) in X1:

```
// Listing4-6.S (cont.)
//
// Sort the "sortMe" array:

            lea     x0, sortMe
            mov     x1, #sortSize   // 16 elements in array
            bl      sort
```

Once `sort` has finished, the program executes a loop to display the 16 values in the array. This loop uses the nonvolatile registers X19 and X20 to hold the base address of the array and the loop index, so these values don't have to be reloaded on each iteration of the loop. Because they are nonvolatile, we know that `printf()` won't disturb their values:

```
// Listing4-6.S (cont.)
//
// Display the sorted array.

            lea     x19, sortMe
            mov     x20, xzr                // X20 = 0 (index)
dispLp:     ldr     w0, [x19, x20, lsl #2]  // W0 = sortMe[X20]
            lea     x1, valToPrint
            str     w0, [x1]
            lea     x1, i
            str     x20, [x1]

            lea     x0, fmtStr      // Print the index
            vparm2  i               // and array element
            vparm3  valToPrint      // on this loop iteration.
            bl      printf

            add     x20, x20, #1    // Bump index by 1.
            cmp     x20, #sortSize  // Are we done yet?
            blo     dispLp
```

Once the output is complete, the main program must restore the nonvolatile registers before returning to the C++ program:

```
// Listing4-6.S (cont.)

            ldr     x19, [sp, #x19Save]  // Restore nonvolatile
            ldr     x20, [sp, #x20Save]  // registers.
            ldr     lr, [sp, #saveLR]    // Restore rtn adrs.
            add     sp, sp, #stackAlloc  // Restore stack.
            ret     // Returns to caller
```

You could slightly optimize this program by using the `stp` and `ldp` instructions to preserve both X19 and X20. To emphasize saving and restoring both registers as independent operations, I didn't make that optimization here. However, you should get in the habit of optimizing your code in this manner in order to reap the benefits of using assembly language.

Here's the `build` command and output for Listing 4-6:

```
$ ./build Listing4-6
$ ./Listing4-6
Calling Listing4-6:
Sortme[0] = 1
Sortme[1] = 2
Sortme[2] = 3
Sortme[3] = 4
Sortme[4] = 5
Sortme[5] = 6
Sortme[6] = 7
Sortme[7] = 8
Sortme[8] = 9
Sortme[9] = 10
Sortme[10] = 11
Sortme[11] = 12
Sortme[12] = 13
Sortme[13] = 14
Sortme[14] = 15
Sortme[15] = 16
Listing4-6 terminated
```

As is typical for a bubble sort, this algorithm terminates if the innermost loop completes without swapping any data. If the data is already presorted, the bubble sort is very efficient, making only one pass over the data. Unfortunately, if the data is not sorted (or, worst case, if the data is sorted in reverse order), then this algorithm is extremely inefficient. Chapter 5 provides an example of a more efficient sorting algorithm, quicksort, in ARM assembly language.

4.7.4 Implementing Multidimensional Arrays

The ARM hardware can easily handle single-dimensional arrays. Unfortunately, however, accessing elements of *multidimensional arrays* takes some work and several instructions.

Before discussing how to declare or access multidimensional arrays, I'll show you how to implement them in memory. First, how do you store a multidimensional object into a one-dimensional memory space? Consider for a moment a Pascal array of this form:

```
A:array[0..3,0..3] of char;
```

This array contains 16 bytes organized as four rows of four characters. Somehow, you have to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. Figure 4-2 shows one way to do this.

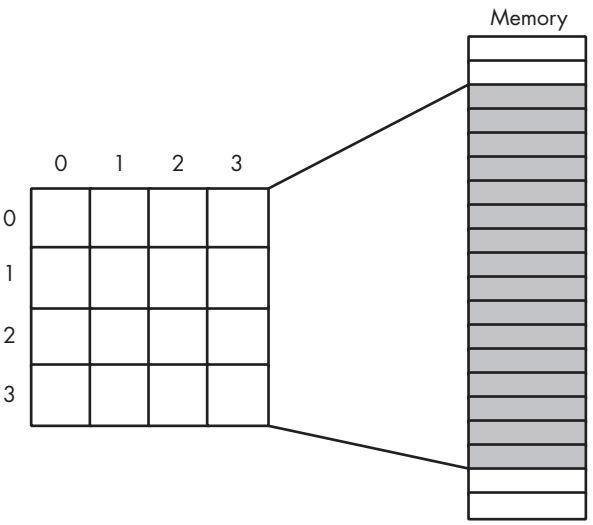


Figure 4-2: Mapping a 4×4 array to sequential memory locations

The actual mapping is not important as long as two things occur: (1) each element maps to a unique memory location (no two entries in the array occupy the same memory locations), and (2) the mapping is consistent (a given element in the array always maps to the same memory location). Therefore, you need a function with two input parameters (row and column) that produces an offset into a linear array of 16 memory locations.

Any function that satisfies these constraints will work fine. Indeed, you could randomly choose a mapping, as long as it's consistent. However, you really want a mapping that is efficient to compute at runtime and that works for any size array (not just 4×4 or even limited to two dimensions). While many possible functions fit this bill, two in particular are used by most programmers and HLLs: row-major ordering and column-major ordering.

4.7.4.1 Row-Major Ordering

Row-major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. Figure 4-3 demonstrates this mapping.

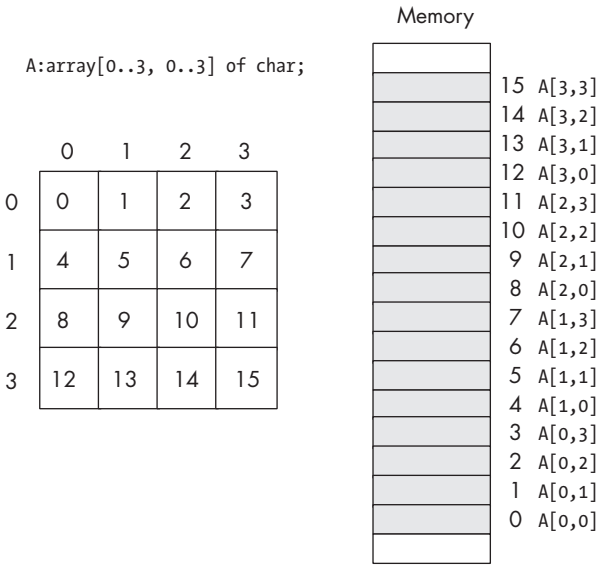


Figure 4-3: Row-major ordering of array elements

ROW AND COLUMN INDICES VS. NUMBERS

When discussing multidimensional arrays, it is easy to confuse row and column numbers and indices. A *row number* is, quite literally, the number associated with a row. In Figure 4-3, the row numbers are the values 0, 1, 2, and 3 to the left of the 4×4 matrix, numbering each of the rows. Similarly, the *column numbers* are the values 0, 1, 2, and 3 at the top of the matrix, numbering each of the columns.

An *index* into a row is the offset from the beginning of each row into the elements of that row. For example, in row 1 in Figure 4-3, the element at index 2 contains the value 6. Similarly, a column index is an index into a column (moving from top to bottom in Figure 4-3). The array element in column 2 at column index 3 is the value 14.

Here's where confusion could occur: a column number is the same as a row index; likewise, a row number is the same as a column index. When this chapter presents the formulas for indexing into multidimensional arrays, be cognizant of the difference between row and column numbers and indices.

Row-major ordering is the method most HLLs employ. It is easy to implement and use in machine language: you start with the first row (row 0) and then concatenate the second row to its end. You then concatenate the third row to the end of the list, then the fourth row, and so on (see Figure 4-4).

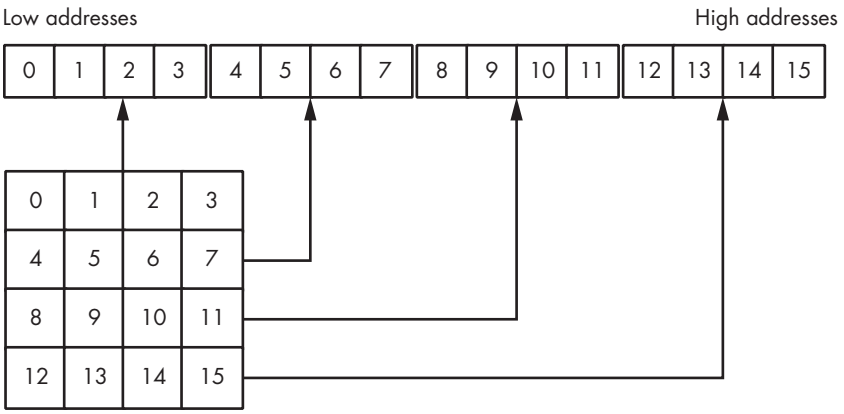


Figure 4-4: Another view of row-major ordering for a 4×4 array

The function that converts a list of index values into an offset is a slight modification of the formula for computing the address of an element of a single-dimensional array. The formula to compute the offset for a two-dimensional row-major ordered array is as follows:

```
Element_Address =
    Base_Address +
      (colindex × row_size + rowindex) × Element_Size
```

As usual, *Base_Address* is the address of the first element of the array (`A[0][0]` in this case), and *Element_Size* is the size of an individual element of the array, in bytes. *colindex* is the leftmost index, and *rowindex* is the rightmost index into the array. *row_size* is the number of elements in one row of the array (4, in this case, because each row has four elements). Assuming *Element_Size* is 1, this formula computes the following offsets from the base address:

Column Index	Row	Offset into Array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

For a three-dimensional array, the formula to compute the offset into memory is the following:

```
Address =
    Base + ((depthindex × col_size + colindex) × row_size +
             rowindex) × Element_Size
```

col_size is the number of items in a column, and *row_size* is the number of items in a row.

In C/C++, if you've declared the array as

```
type A[i][j][k];
```

then *row_size* is equal to `k` and *col_size* is equal to `j`.

For a four-dimensional array, declared in C/C++ as

```
type A[i][j][k][m];
```

the formula for computing the address of an array element is shown here:

```
Address =
    Base +
      (((LeftIndex × depth_size + depthindex) × col_size +
         colindex) × row_size + rowindex) × Element_Size
```

depth_size is equal to `j`, *col_size* is equal to `k`, and *row_size* is equal to `m`. *LeftIndex* represents the value of the leftmost index.

By now you're probably beginning to see a pattern. A generic formula will compute the offset into memory for an array with *any* number of dimensions; however, you'll rarely use more than four.

Another convenient way to think of row-major arrays is as arrays of arrays. Consider the following single-dimensional Pascal array definition

```
A: array [0..3] of sometype;
```

where *sometype* is the type *sometype* = `array [0..3] of char;` and `A` is a single-dimensional array. Its individual elements happen to be arrays, but you can safely ignore that for the time being.

Here is the formula to compute the address of an element in a single-dimensional array:

```
Element_Address = Base + Index × Element_Size
```

In this case, *Element_Size* happens to be 4 because each element of `A` is an array of four characters. Therefore, this formula computes the base address of each row in this 4×4 array of characters (see Figure 4-5).

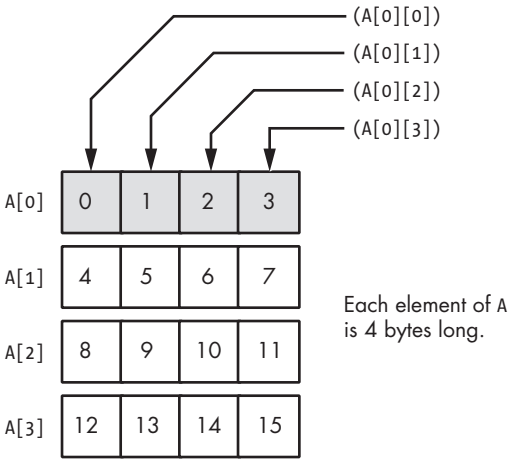


Figure 4-5: Viewing a 4×4 array as an array of arrays

Of course, once you compute the base address of a row, you can reapply the single-dimensional formula to get the address of a particular element. While this doesn't affect the computation, it's probably a little easier to deal with several single-dimensional computations rather than a complex multi-dimensional array computation.

Consider a Pascal array defined as follows:

```
A:array [0..3, 0..3, 0..3, 0..3, 0..3] of char;
```

You can view this five-dimensional array as a single-dimensional array of arrays. The following Pascal code provides such a definition:

```
type
 OneD = array[0..3] of char;
 TwoD = array[0..3] of OneD;
 ThreeD = array[0..3] of TwoD;
 FourD = array[0..3] of ThreeD;
var
 A: array[0..3] of FourD;
```

The size of `OneD` is 4 bytes. Because `TwoD` contains four `OneD` arrays, its size is 16 bytes. Likewise, `ThreeD` is four `TwoD`s, so it is 64 bytes long. Finally, `FourD` is four `ThreeD`s, so it is 256 bytes long. To compute the address of `A[b, c, d, e, f]`, you could use the following steps:

1. Compute the address of `A[b]` as *Base* + `b` × *size*. Here *size* is 256 bytes. Use this result as the new base address in the next computation.
2. Compute the address of `A[b, c]` by the formula *Base* + `c` × *size*, where *Base* is the value obtained in the previous step and `size` is 64. Use the result as the new base in the next computation.
3. Compute the base address of `A [b, c, d]` by *Base* + `d` × *size*, where *Base* comes from the previous computation and *size* is 16. Use the result as the new base in the next computation.

4. Compute the address of `A[b, c, d, e]` with the formula *Base* + `e` × *size*, using *Base* from the previous step and a size of 4. Use this value as the base for the next computation.
5. Finally, compute the address of `A[b, c, d, e, f]` by using the formula *Base* + `f` × *size*, where *Base* comes from the previous computation and *size* is 1 (obviously, you can ignore this final multiplication). The result you obtain at this point is the address of the desired element.

One of the main reasons you won't find higher-dimensional arrays in assembly language is that assembly language emphasizes the inefficiencies associated with such access. It's easy to enter something like `A[b, c, d, e, f]` into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier—they see the mess you wind up with when you use higher-dimensional arrays. Indeed, good assembly language programmers try to avoid two-dimensional arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory.

4.7.4.2 Column-Major Ordering

Column-major ordering is the other function HLLs frequently use to compute the address of an array element. FORTRAN and various dialects of BASIC (for example, older versions of Microsoft BASIC) use this method.

In row-major ordering, the rightmost index increases the fastest as you move through consecutive memory locations. In column-major ordering, the leftmost index increases the fastest. Pictorially, a column-major ordered array is organized as shown in Figure 4-6.

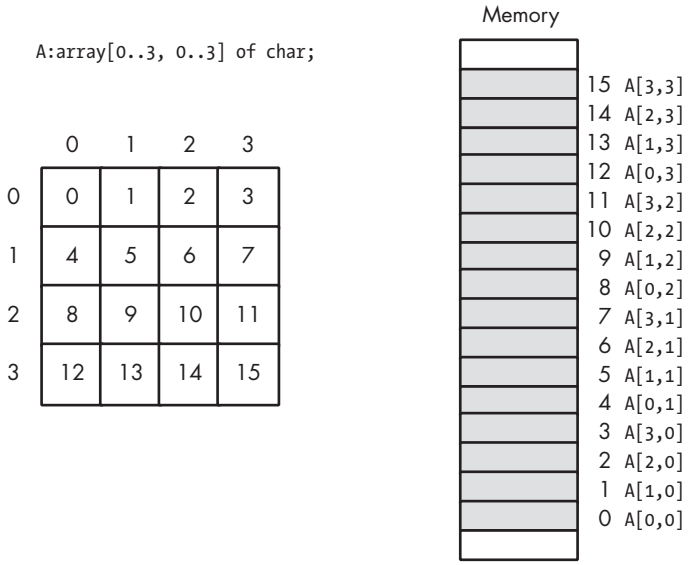


Figure 4-6: Column-major ordering of array elements

The formula for computing the address of an array element when using column-major ordering is similar to that for row-major ordering. You reverse the indices and sizes in the computation.

For a two-dimensional column-major array:

```
Element_Address = Base_Address +
                  (rowindex × col_size + colindex) × Element_Size
```

For a three-dimensional column-major array:

```
Address = Base + ((rowindex × col_size + colindex) ×
        depth_size + depthindex) × Element_Size
```

For a four-dimensional column-major array:

```
Address =
    Base + (((rowindex × col_size + colindex) × depth_size +
      depthindex) × Left_size + Leftindex) × Element_Size
```

The formulas for higher-dimension arrays progress in a like fashion.

4.7.4.3 Storage Allocation for Multidimensional Arrays

If you have an $m \times n$ array, it will have $m \times n$ elements and require $m \times n \times$ *Element_Size* bytes of storage. To allocate storage for an array, you must reserve this memory. As usual, you can accomplish this task in several ways. The most common way to declare a multidimensional array in Gas is to use the `.space` directive:

```
ArrayName: .space size1 * size2 * size3 * ... * sizen * (Element_Size)
```

Here, $size_1$ to $size_n$ are the sizes of each of the dimensions of the array, and (*Element_Size*) is the size (in bytes) of a single element. I recommend putting parentheses around the *Element_Size* component of this expression to emphasize that it is not another dimension in the multidimensional array.

For example, here is a declaration for a 4×4 array of characters:

```
GameGrid: .space 4 * 4 // Element_Size is 1.
```

Here is another example that shows how to declare a three-dimensional array of strings (assuming the array holds 64-bit pointers to the strings):

```
NameItems: .space 2 * 3 * 3 * (8)  // dword NameItems[2, 3, 3]
```

As with single-dimensional arrays, you may initialize every element of the array to a specific value by following the declaration with the values of the array constant. Array constants ignore dimension information; all that matters is that the number of elements in the array constant corresponds to the number of elements in the actual array. The following example shows the `GameGrid` declaration with an initializer:

```
GameGrid: .byte 'a', 'b', 'c', 'd'
          .byte 'e', 'f', 'g', 'h'
          .byte 'i', 'j', 'k', 'l'
          .byte 'm', 'n', 'o', 'p'
```

This example was laid out to enhance readability. Gas does not interpret the four separate lines as representing rows of data in the array; humans do, which is why it's good to write the data in this manner. If you have a large array, an array with really large rows, or an array with many dimensions, there is little hope for winding up with something readable; in this case, comments that carefully explain everything come in handy.

The use of a constant expression to compute the number of array elements rather than simply using the constant 16 (4×4) more clearly suggests that this code is initializing each element of a 4×4 element array than does the simple literal constant 16.

4.7.4.4 How to Access Elements of Multidimensional Arrays

To access elements of a multidimensional array, you'll need to be able to multiply two values; this is done using the `mul` (multiply) and `madd` (multiply and add) instructions.

The `mul` and `madd` instructions have the following syntax

```
mul  reg_d, reg_1, reg_r        // reg_d = reg_1 * reg_r
madd reg_d, reg_1, reg_r, reg_a  // reg_d = reg_1 * reg_r + reg_a
```

where reg_d is the destination register (32 or 64 bits), reg_1 and reg_r are source registers (left- and right-hand operands), and reg_a is a third source operand. These instructions perform the calculations described in the comments.

These instructions do not have a form with an `s` suffix and therefore do not update the flags after their execution. An n-bit × n-bit multiplication can produce a $2 \times n$ bit result; however, these instructions maintain only n bits in the destination register. Any overflow is lost. Sadly, these instructions do not allow immediate operands, though this would be useful.

The multiply instruction has several other variants that are used for other purposes. These are covered in Chapter 6.

Now that you've seen the formulas for computing the address of a multidimensional array element, it's time to see how to access elements of those arrays with assembly language. The `ldr`, `lsl`, and `mul`/`madd` instructions make short work of the various equations that compute offsets into multidimensional arrays. First, consider a two-dimensional array:

```
        .data
i:      .word  .-.
j:      .word  .-.
TwoD:   .word  4 * 8  * (4)
          .
          .
          .
```

```
// To perform the operation TwoD[i,j] := 5;
// you'd use code like the following.
// Note that the array index computation is (i * 4 + j) * 4.

        lea x0, i
        ldr w0, [x0]    // Clears HO bits of X0
        lsl x0, x0, #2  // Multiply i by 4.
        lea x1, j
        ldr w1, [x1]
        add w0, w0, w1  // W0 = i * 4 + j
        lea x1, TwoD    // X1 = base
        mov w2, #5      // [TwoD + (i * 4 + j) * 4] = 5
        str w2, [x1, x0, lsl #2] // Scaled by 4 (element size)
```

Now consider a second example that uses a three-dimensional array:

```
        .data
i:      .word  .-.
j:      .word  .-.
k:      .word  .-.
ThreeD: .space 3 * 4 * 5 * (4) // word ThreeD[3, 4, 5]
          .
          .
          .
// To perform the operation ThreeD[i,j,k] := W7;
// you'd use the following code that computes
// ( (i * 4 + j) * 5 + k ) * 4 as the address of ThreeD[i,j,k].

          lea  x0, i
          ldr  w0, [x0]
          lsl  w0, w0, #2  // Four elements per column
          lea  x1, j       // Add in j.
          ldr  w1, [x1]
          add  w0, w0, w1
          mov  w1, #5      // Five elements per row
          lea  x2, k
          ldr  w2, [x2]
          madd w0, w0, w1, w2 // ( (i * 4 + j) * 5 + k )
          lea  x1, ThreeD
          str  w7, [x1, w0, uxtw #2] // ThreeD[i,j,k] = W7
```

This code uses the `madd` instruction to multiply the value in W0 by 5 and add in the `k` index at the same time. Because the `lsl` instruction can multiply a register by only a power of 2, we must resort to a multiplication here. While there are ways to multiply the value in a register by a constant other than a power of 2, the `madd` instruction is more convenient, especially as it handles an addition operation at the same time.

4.8 Structs

Another major composite data structure is the Pascal *record* or C/C++/C# *struct*. The Pascal terminology is probably better, because it tends to avoid confusion with the more general term *data structure*. However, this book

uses the term *struct*, as C-based languages are more commonly used these days. (Records and structures also go by other names in other languages, but most people recognize at least one of these names.)

Whereas an array is homogeneous, with elements that are all the same type, the elements in a struct can have different types. Arrays let you select a particular element via an integer index. With structs, you must select an element, known as a *field*, by offset (from the beginning of the struct).

The whole purpose of a struct is to let you encapsulate different, though logically related, data into a single package. The Pascal record declaration for a hypothetical student is a typical example:

```
student =
     record
          sName:    string[64];
          Major:    integer;
          SSN:      string[11];
          Midterm1: integer;
          Midterm2: integer;
          Final:    integer;
          Homework: integer;
          Projects: integer;
     end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name, the next 2 bytes hold the `Major` code (assuming a 16-bit integer), the next 12 bytes hold the Social Security number, and so on. (Strings require an extra byte, in addition to all the characters in the string, to encode the length.) The `John` variable declaration allocates 89 bytes of storage laid out in memory, as shown in Figure 4-7 (assuming no padding or alignment of fields).

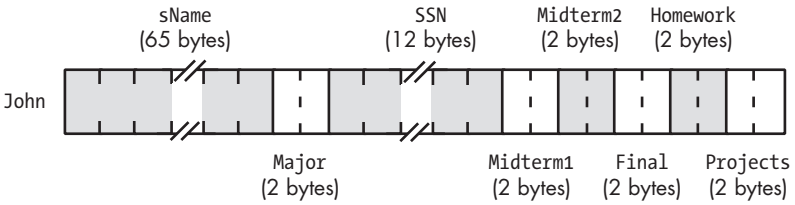


Figure 4-7: Student data structure in memory

If the label `John` corresponds to the base address of this record, the `sName` field is at offset `John + 0`, the `Major` field is at offset `John + 65`, the `SSN` field is at offset `John + 67`, and so on. In assembly language, if X0 holds the base address of the `John` structure, you could access the `Major` field by using the following instruction:

```
ldrh  w0, [x0, #65]
```

This loads W0 with the 16-bit value at the address specified by `John + 65`.

4.8.1 Dealing with Limited Gas Support for Structs

Unfortunately, Gas provides only the smallest amount of support for structures via the `.struct` directive (see "Linux `.struct` Directive" on page 217). Even more unfortunately, the macOS assembler doesn't support `.struct`.

To use structures under macOS and Linux together, you'll need a way to specify the offsets to all the fields of a structure for use in the register indirect-plus-offset addressing mode (such as in the last example line of the previous section). In theory, you could manually use equates to define all the offsets:

```
.equ sName, 0
.equ Major, 65
.equ SSN, 67
.equ Mid1, 79
.equ Mid2, 81
.equ Final, 83
.equ Homework, 85
.equ Projects, 87
```

However, this is an absolutely horrible, error-prone, and difficult-to-maintain approach. The ideal method would be to supply a structure name (the type name) and a list of the field names and their types. From this, you'd aim to get offsets for all the fields, plus the size of the entire structure (which you can use with the `.space` directive to allocate storage for the structure).

The *aoaa.inc* include file contains several macro definitions that can help you declare and use structures in your assembly language source files. These macros aren't amazingly robust, but when used carefully, they get the job done. Table 4-1 lists these macros and their arguments. Field names must be unique throughout the program, not just in the structure definition. Also note that the `struct`/`ends` macros do not support nesting.

Table 4-1: The *aoaa.inc* Macros for Defining Structures

Macro	Argument(s)	Description
`struct`	*name*, *offset*	Begin a structure definition. The offset field is optional and can be either a (small) negative number or 0. The default (and most commonly used) value is 0.
`ends`	*name*	Ends a structure definition. The *name* argument must match the name supplied in the `struct` invocation.
`byte`	*name*, *elements*	Create a field of type byte. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.
`hword`	*name*, *elements*	Create a field of type hword. *name* is the (unique) field name. *elements* is optional (default value is 1) and specifies the number of array elements.
`word`	*name*, *elements*	Create a field of type word. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.
`dword`	*name*, *elements*	Create a field of type dword. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.
`qword`	*name*, *elements*	Create a field of type qword. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.

Macro	Argument(s)	Description
single	*name*, *elements*	Create a field of type single. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.
double	*name*, *elements*	Create a field of type double. *name* is the unique field name. *elements* is optional (default value is 1) and specifies the number of array elements.

For strings, you'd specify either a `dword` field (to hold a pointer to the field) or a `byte` field with a sufficient number of elements to hold all the characters in the string.

The student example from the previous section could be encoded as follows:

```
struct student
    byte  sName, 65 // Includes zero-terminating byte
    hword Major
    byte  SSN, 12   // Includes zero-terminating byte
    hword Midterm1
    hword Midterm2
    hword Final
    hword Homework
    hword Projects
ends student
```

You would declare a variable of type `student` like this:

```
student John
```

The `ends` macro automatically generates a macro with the same name as the structure name, so you can use that like a directive to allocate sufficient space to hold an instance of the structure type.

You could access fields of `John` as follows:

```
lea  x0, John
ldrh w1, [x0, #Midterm1]
ldrh w2, [x0, #Midterm2]
ldrh w3, [x0, #Final]   // And so on ...
```

This macro package has a couple of issues. First of all, the field names must be unique throughout the assembly language source file (unlike standard structures, where the field names are local to the structure itself). As a result, these structures tend to suffer from *namespace pollution*, which happens when you try to reuse some of the field names for other purposes. For example, `sName` will likely be used again elsewhere in the source file, since it's a common identifier. A quick-and-dirty solution to this problem is to always prefix the field names with the structure name and a period. For example:

```
struct student
    byte  student.sName, 65
    hword student.Major
```

```
    byte  student.SSN, 12
    hword student.Midterm1
    hword student.Midterm2
    hword student.Final
    hword student.Homework
    hword student.Projects
ends student
```

This requires a bit more typing, but it resolves the namespace pollution issue most of the time.

Consider the `student John` macro invocation/declaration given in this section. This macro expands into

```
John:  .fill student.size
```

where `student.size` is an extra field that the `struct` macro generates, specifying the total size of the structure (in bytes).

The `struct` macro accepts a second (optional) parameter: the *starting offset* for fields in the structure. By default, this is 0. If you supply a negative number here, the directive/macro that `struct` generates works a little differently. Consider the following structure definition:

```
struct HLAstring, -4
word   HLAstring.len
byte   HLAstring.chars, 256
ends   HLAstring
```

HLA strings are actually a bit different from the structure provided here, but this does serve as a good example of negative starting offsets.

The `HLAstring` macro that `struct` generates does the following:

```
           HLAstring myString
         // Expands to
           .fill      4
myString:  .fill      256
```

This expansion places the `myString` label after the first 4 bytes of the beginning of the structure. This is because the `HLAstring.len` field's offset is –4, meaning that the length field starts 4 bytes before the base address of the structure (and the structure variable's name is always associated with the base address). You'll see some important uses for this feature in the next chapter.

The `struct` macro does not allow positive offsets (values greater than 0). It will generate an error at assembly time if you specify a positive value.

LINUX .STRUCT DIRECTIVE

The Gas `.struct` directive (available only under Linux) doesn't begin a structure definition in the sense of an HLL like C/C++. Instead, it begins a new section (similar to `.text`, `.data`, or `.section`). However, rather than creating a memory section that can be relocated in memory when the OS loads the program, the `.struct` section is an absolute section located in memory at the address specified by *constExpression*. Furthermore, this is a phantom section, insofar as Gas doesn't actually write any data to the object file in response to this directive; `.struct` exists only for the purpose of associating offsets with symbols created within the section.

Consider the following simple example:

```
        .struct     0
f1:     .byte       0
f2:     .hword      0
f3:     .word       0
f4:     .dword      0
size    =           .
```

During assembly, Gas will set the location counter of the `.struct` section to 0 (because of the `0` operand after `.struct`). Therefore, symbol `f1` will have the offset 0 associated with it, as it is the first symbol defined in the section. Because `f1` is a byte (and consumes 1 byte of memory), the location counter advances to 1 when Gas encounters symbol `f2`. Therefore, `f2` has the offset 2 associated with it. Similarly, `f3` has the offset 3, and `f4` has the offset 7 (the offsets are the sums of the sizes of all the prior objects in the section). The symbol size is given the value of the location counter at the end of the sequence, so it has the value 15.

You can use the symbol names defined in a `.struct` section as offsets in an address expression. For example, if you've defined the structure object `s1` as `s1: .space size`, you can access the `f3` field of `s1` as follows

```
lea x0, s1
ldr w0, [x0, #f1]
```

where `#f1` is the offset into the struct from its base address (held in X0).

One issue with the `struct` macro is that it doesn't provide a way to initialize the fields of the structure. To learn how to do that, keep reading.

4.8.2 Initializing Structs

The `struct` macro definitions do not provide any way to initialize the fields of a structure at compile time. You'll have to either assign the values at

runtime or manually build up the structure variable by using Gas directives. For example:

```
John:    .asciz  "John Somebody"   // sName
         .space  65 - (.-John)     // Must be 65 bytes long!
         .hword  0                 // Major
         .asciz  "123-45-6578"     // SSN-Exactly 12 bytes long
         .hword  75                // Midterm1
         .hword  82                // Midterm2
         .hword  90                // Final
         .hword  72                // Homework
         .hword  80                // Projects
```

This initializes the fields of the structure to the corresponding values.

4.8.3 Creating Arrays of Structs

A common pattern in program design is to create an array of structures. To do so, create a `struct` type and multiply its size by the number of array elements when declaring the array variable, as shown in the following example:

```
numStudents = 30
   .
   .
   .
Class:  .fill student.size * numStudents
```

To access an element of this array, use the standard array-indexing techniques. Because `class` is a single-dimensional array, you'd compute the address of an element of this array by using the formula `baseAddress + index × student.size`. For example, to access an element of `class`, you'd use code like the following:

```
// Access field Final, of element i of class:
// X1 := i * student.size + offset Final

    lea  x1, i
    ldr  x1, [x1]
    mov  x2, #student.size
    mov  x3, #student.Final
    madd x1, x1, x2, x3   // Include offset to field.
    lea  x2, class
    ldrh w0, [x2, x1]     // Accesses class[i].Final
```

You must sum in the offset to the field you want to access. Sadly, the scaled-indexed addressing mode doesn't include an offset component as part of the addressing mode, but `madd` saves us an instruction by working in this addition as part of the multiplication.

Naturally, you can create multidimensional arrays of structs as well, using the row-major or column-major order functions to compute the

address of an element within such structs. The only real change is that the size of each element is the size of the struct object:

```
        .data

numStudents = 30
numClasses  = 2

// student Instructor[numClasses][numStudents]

Instructor:   .fill  numStudents * numClasses * (student.size)
whichClass:   .dword 1
whichStudent: .dword 10
          .
          .
          .
// Access element [whichClass,whichStudent] of class
// and load Major into W0:

     lea  x0, whichClass
     ldr  x1, [x0]
     mov  x2, #numStudents    // X1 = whichClass * numStudents
     mul  x1, x1, x2
     lea  x0, whichStudent
     ldr  x2, [x0]            // X1 = (whichClass * numStudents +
     add  x1, x1, x2          //  numStudents)
     mov  x2, #student.size   //   * sizeStudent + offset Major
     mov  x3, #Major
     madd x1, x1, x2, x3

     lea  x0, Instructor // W0 =  Instructor[whichClass]
     ldrh w0, [x0, x1]   //         [whichStudent].Major
```

This demonstrates how to access fields of an array of structs.

4.8.4 Aligning Fields Within a Struct

To achieve maximum performance in your programs, or to ensure that Gas structures properly map to records or structures in an HLL, you will often need to be able to control the alignment of fields within a struct. For example, you might want to ensure that a double-word field's offset is a multiple of 4. You can use the `salign` macro to do this. The following creates a structure with aligned fields:

```
struct tst
byte bb
salign 2   // Aligns offset to next 4-byte boundary
byte c
ends tst
```

As for the `.align` directive, the `salign` macro aligns the structure's offset to 2^n, where n is the value specified as the `salign` argument. In this example, `c`'s offset is set to 4 (the macro rounds up the field offset from 1 to 4).

Field alignment is up to you when you're creating your own structure variables. However, if you're linking with code written in an HLL that uses structures, you'll need to determine field alignment for that particular language. Most modern HLLs use *natural alignment*: fields are aligned on a boundary that is the size of that field (or an element of that field). The structure itself is aligned at an address rounded to the size of the largest object in the structure. See section 4.11, "For More Information," on page 221 for appropriate links.

4.9 Unions

Unions (in an HLL like C/C++) are similar to structures insofar as they create an aggregate data type containing several fields. Unlike structures, however, the fields of a union all occupy the same offset in the data structure.

Programmers typically use unions for one of two reasons: to conserve memory or to create aliases. Memory conservation is the intended use of this data structure facility. To see how this works, consider the following struct type:

```
struct numericRec
       word  i
       word  u
       dword q
ends   numericRec
```

If you declare a variable, say `n`, of type `numericRec`, you access the fields as `n.i`, `n.u`, and `n.q`. A struct assigns different offsets to each field, effectively allocating separate storage to each field. A union, on the other hand, assigns the same offset (typically 0) to each of these fields, allocating the same storage to each.

For `struct`, then, `numericRec.size` is 16 because the struct contains two word fields and a double-word field. The size of the corresponding `union`, however, would be 8. This is because all the fields of a union occupy the same memory locations, and the size of a union object is the size of the largest field of that object (see Figure 4-8).

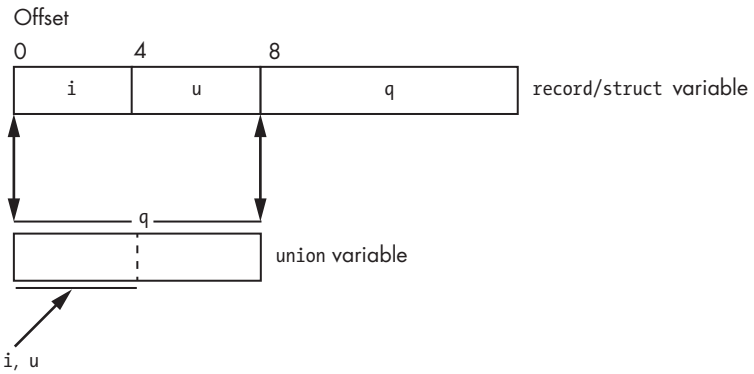


Figure 4-8: The layout of a union versus a structure variable

Programs use unions for several purposes: preserving memory, overlaying data types, and creating *variant types* (dynamically typed values whose type can change during execution). Because you probably won't use unions that often in an assembly language program, I've not bothered creating a union macro in the *aoaa.inc* include file. However, if you really need a union macro, you could take the information in Chapter 13 and the source code to the `struct` macro in *aoaa.inc* and write your own.

4.10 Moving On

This chapter concludes the machine organization component of this book, which dealt with the organization of memory, constants, data, and data types. It discussed memory variables and data types, arrays, row-major and column-major ordering, structs and unions, and strings, including zero-terminated, length-prefixed, and descriptor-based strings. It also covered issues you may encounter when using pointers, including uninitialized pointers, illegal pointer values, dangling pointers, memory leaks, and type-unsafe access.

Now it's time to begin studying assembly language *programming* in earnest. The next section of the book will begin discussing procedures and functions (Chapter 5), arithmetic (Chapter 6), low-level control structures (Chapter 7), and advanced arithmetic (Chapter 8).

4.11 For More Information

- For additional information about data structure representation in memory, consider reading my book *Write Great Code*, Volume 1, 2nd edition (No Starch Press, 2020). For an in-depth discussion of data types, consult a textbook on data structures and algorithms such as *Introduction to Algorithms*, 3rd edition (MIT Press, 2009), by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- You can find information about the GNU assembler (including the `.struct` directive) in the manual at *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html*.
- As noted in Chapter 3, you can find more information about ARM CPUs at the developer website at *https://developer.arm.com*. To learn more about field alignment in particular, see *https://developer.arm.com/documentation/dui0491/i/C-and-C---Implementation-Details/Structures--unions--enumerations--and-bitfields?lang=en*.
- For more on dangling pointers, see *https://en.wikipedia.org/wiki/Dangling_pointer*.
- For more on the High-Level Assembler, see the online resources at my website, *https://www.randallhyde.com/AssemblyLanguage/HighLevelAsm/*.

TEST YOURSELF

1. What is a manifest constant?
2. Which directive(s) would you use to create a manifest constant?
3. What is a constant expression, and how would you determine the number of data elements in the operand field of a byte directive?
4. What is the location counter?
5. What operator returns the current location counter?
6. How would you compute the number of bytes between two declarations in the `.data` section?
7. What is a pointer and how is it implemented?
8. How do you dereference a pointer in assembly language?
9. How do you declare pointer variables in assembly language?
10. What are the five common problems encountered when using pointers in a program?
11. What is a dangling pointer?
12. What is a memory leak?
13. What is a composite data type?
14. What is a zero-terminated string?
15. What is a length-prefixed string?
16. What is a descriptor-based string?
17. What is an array?
18. What is the base address of an array?
19. Provide an example of an array declaration.
20. Describe how to create an array whose elements you initialize at assembly time.
21. What is the formula for accessing elements of a:
    a. Single-dimensional array `dword A[10]`?
    b. Two-dimensional array `word W[4, 8]`?
    c. Three-dimensional array `double R[2, 4, 6]`?
22. What is row-major order?
23. What is column-major order?
24. Provide an example of a two-dimensional array declaration (word array `W[4, 8]`).
25. What is a struct (record)?

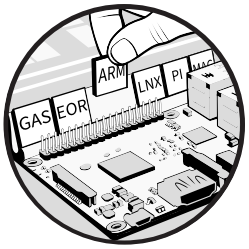
26. How do you declare a struct data structure?
27. How do you access fields of a struct?
28. What is a union?
29. What is the difference between the memory organization of fields in a union versus those in a struct?

PART II

BASIC ASSEMBLY LANGUAGE

5

PROCEDURES



In a procedural programming language, the basic unit of code is the procedure. A *procedure* is a set of instructions that compute a value or take an action, such as printing or reading a character value. This chapter discusses how Gas implements procedures, parameters, and local variables. By the end of this chapter, you should be well versed in writing your own procedures and functions. You'll also fully understand parameter passing and the ARM application binary interface (ABI) calling convention.

This chapter covers several topics, including the following:

- An introduction to assembly language programming style, along with some *aoaa.inc* macros to improve the readability of your programs

- Gas procedures/functions and their implementation (including the use of the `bl`, `br`, and `ret` instructions), along with more *aoaa.inc* macros to allow the better declaration of procedures in your source files
- Activation records, automatic variables, local symbols, register preservation, and the ARM stack
- Various ways to pass parameters to a procedure, including pass by value and pass by reference, and how to use procedure pointers and procedural parameters

This chapter also discusses how to return function results to a caller and how to call and use recursive functions.

5.1 Assembly Language Programming Style

Up until this chapter, I've not stressed good assembly language programming style for two reasons. First, this book assumes you're already familiar with the need for good programming style based on your experience with HLLs. Second, the programs quoted up to this point have been relatively trivial, and programming style doesn't matter much with trivial code. However, as you begin to write more advanced ARM assembly language programs, style becomes more important.

As you can probably tell by now, ARM assembly language code is nowhere near as readable as code written in an HLL such as C/C++, Java, or Swift. Therefore, as an assembly language programmer, you must expend extra effort to write assembly code that is as readable and maintainable as possible. As I've pointed out, the GNU assembler was written not as a tool for assembly language programmers but as a backend to the GCC compiler to process the compiler's output. Because of this and the fact that Gas attempted to absorb as many features as possible from a huge number of assembly languages (for many CPUs, not just the ARM), writing high-quality code with Gas is a difficult task.

Fortunately, you can use Gas's macro processor (and ability to take advantage of the CPP) to modify the Gas assembly language somewhat, accessing features that can help improve your programming style. The *aoaa.inc* include file contains a fair number of predefined macros and symbol definitions to help achieve this goal. Chapter 13 covers the contents of *aoaa.inc* line by line and explains how you can use these macros, and create macros of your own, to improve the readability of your ARM assembly language programs.

When you write assembly language source files, feel free to include *aoaa.inc* in that code or incorporate any features from that code in your assembly language source files. Even if you don't require the cross-platform portability offered by *aoaa.inc*, its macros and other definitions can help you write more readable and maintainable code. The *aoaa.inc* header file is open source and covered by the Creative Commons 4.0 Attribution license (see section 5.12, "For More Information," on page 290).

As an example of using macros to make code more readable, consider the `.code` macro from *aoaa.inc*. It expands into the following two statements:

```
.text
.align 2
```

As a general rule, you should always ensure that the `.text` section is aligned on a word boundary (code could get misaligned if you've declared some data in the previous code section whose length is not a multiple of 4). It's good programming style to always align a `.text` section; just to be sure an instruction begins at a proper address. Rather than clutter up your code with a bunch of extra `.align` directives, I recommend using the `.code` directive to automatically handle the alignment. Having less clutter makes your code easier to read.

The *aoaa.inc* header file contains several additional macros I will present throughout the rest of this chapter that take the 1960s-style Gas syntax and attempt to provide features found in more modern assemblers (such as the Microsoft Macro Assembler, or MASM, and the HLA assemblers available for the x86 processor family). Using these features (such as formal procedure declarations and local variable declarations) can help produce easier-to-read assembly language source code.

Even when writing traditional assembly language source code, you can follow certain rules to produce more readable code. Throughout this book, I've generally organized assembly language statements as follows (braces surround optional items and don't appear in the actual source code):

```
{Label:}   {{instruction}     operands}  {// Comment}
```

As a general rule, I try to put all label definitions in column 1 and to line up all the instruction mnemonics in column 2. I try to start the operands in column 3. The exact number of spaces between these columns is not important, but be sure that the mnemonics are generally lined up together, in one column, and that the operands tend to start in the next column. This is the traditional assembly language programming style and the format that most assembly language programmers will want to see when reading your code.

NOTE *For formatting reasons, this book often compresses the amount of space between the columns and sometimes varies the position of each column within the same listing. This was done to ensure source lines fit on one line within the book. In a normal source file, you should try to keep all the columns aligned (two 4-character tab positions for column 2, column 3 around character position 16, and so on).*

In general, don't try to indent statements as you would blocks in an HLL. Assembly language is not a block-structured language and does not lend itself to the same indentation techniques that work well for block-structured languages. If you need to set apart a sequence of statements, the best approach is to insert two or more blank lines before and after that

sequence of statements. Comments are also useful for differentiating two separate, loosely coupled blocks of code.

Gas usually expects an entire assembly language instruction to reside on a single line of source code. In theory, you could use the backslash character immediately before a newline character to break a single statement across two lines:

```
b.al   \
  targetLabel
```

However, there's almost never a good argument for doing this. Keep your instructions on a single line unless you have a really good reason to split them across multiple lines (for example, if the source line becomes inordinately long for some reason, which is rare). The label field is an exception to this rule: labels may appear on a line by themselves even if they are associated with the next machine instruction in the program.

Gas (under Linux) allows putting multiple assembly language instructions on the same line, separated by a semicolon. However, putting multiple statements on the same source line is an even worse idea in assembly language than it is in HLLs—don't do it. In any case, the macOS assembler does not support this feature.

With a few assembly language style guidelines out of the way, it's time to consider the main topic of this chapter: procedures (functions) in assembly language.

5.2 Gas Procedures

Most procedural programming languages implement procedures by using the call/return mechanism. The code calls a procedure, the procedure performs whatever actions it was written to do, and then the procedure returns to the caller. The call and return operations provide the ARM's *procedure invocation mechanism*. The calling code calls a procedure with the `bl` instruction, and the procedure returns to the caller with the `ret` instruction. For example, the following ARM instruction calls the C stdlib library `printf()` function:

```
bl printf
```

Alas, the C stdlib does not supply all the routines you'll need. Most of the time, you'll have to write your own Gas procedures. A basic Gas procedure declaration takes the following form:

```
procName:
    Procedure statements
    ret
```

Technically, the procedure does not need to end with a `ret` instruction; the `ret` could be somewhere in the middle of the procedure, with a `b.al`

instruction at the end. However, it's considered good programming style to use a `ret` instruction (or an equivalent) as the last instruction of a procedure's body.

Procedure declarations appear in the `.text` section of your program. In the preceding syntax example, *`procName`* represents the name of the procedure you wish to define. This can be any valid (and unique) Gas identifier.

Here is a concrete example of a Gas procedure declaration. This procedure stores 0s into the 256 words at which X0 points upon entry into the procedure:

```
zeroBytes:
          mov  x1, #256*4     // 1,024 bytes = 256 words
repeatlp: subs x1, x1, #4
          str  wzr, [x0, x1]  // Store *after* subtraction!
          bne  repeatlp       // Repeat while X1 >= 0.
          ret
```

As you've probably noticed, this simple procedure doesn't bother with the "magic" instructions that add and subtract a value to and from the SP register. Those instructions are a requirement of the ARM ABI when the procedure will be calling other C/C++ code (or other code written in an ARM ABI–compliant language). Because this little function doesn't call any other procedures, it doesn't bother executing such code.

Also note that this code uses the loop index to count down from 1,024 down to 0 by 4, filling in the 256-word array backward (from end to beginning) rather than filling it in from beginning to end. This is a common technique in assembly language. Finally, this code decrements X1 by 4 before storing the 0 into memory. This is because the loop index (X1) is initialized just beyond the end of the array pointed at by X0. The `str` instruction does not affect the flags, so the `bne` instruction responds to the flags set by the `subs` instruction.

You can use the ARM `bl` instruction to call this procedure. When, during program execution, the code falls into the `ret` instruction, the procedure returns to whoever called it and begins executing the first instruction beyond `bl`. Listing 5-1 provides an example of a call to the `zeroBytes` routine.

```
// Listing5-1.S
//
// Simple procedure call example

#include "aoaa.inc"

stackSpace  =       64
saveLR      =       56

            .section .rodata, ""
ttlStr:     .asciz   "Listing 5-1"
```

```
            .data
wArray:     .space   256 * (4), 0xff // Fill with 0xFF.


            .text
            .align   2

// getTitle
//
// Return program title to C++ program:

            .global getTitle
getTitle:
            lea     x0, ttlStr
            ret

// zeroBytes
//
// Here is the user-written procedure
// that zeros out a 256-word buffer.
// On entry, X0 contains the address
// of the buffer.

zeroBytes:
            mov     x1, #256 * 4
repeatlp:   subs    x1, x1, #4
            str     wzr, [x0, x1] // Store *after* subtraction!
            bne     repeatlp      // Repeat while X1 != 0.
            ret

// Here is the asmMain function:

            .global asmMain
asmMain:
            sub     sp, sp, #stackSpace // Reserve stack storage.
            str     lr, [sp, #saveLR]

            lea     x0, wArray
            bl      zeroBytes

            ldr     lr, [sp, #saveLR]   // Restore return address.
            add     sp, sp, #stackSpace // Clean up stack.
            ret     // Returns to caller
```

I won't bother with a build or run command, as this program doesn't produce any real output beyond saying that it ran and terminated.

The Gas language doesn't really have a syntactical concept of a program component we think of as a procedure (or function). It has labels you can call with the `bl` instruction, along with the `ret` instruction, which you can use to return from a procedure. However, it has no syntactical entity you can use to delineate one procedure from another in your assembly language source file.

So far, the few procedures in this book have delineated the code in the procedure by using a label and a return statement. For example, the following procedure begins with `zeroBytes` and ends with `ret`:

```
zeroBytes:
          mov  x1, #256 * 4
repeatlp: subs x1, x1, #4
          str  wzr, [x0, x1] // Store *after* subtraction!
          bge  repeatlp      // Repeat while X1 >= 0.
          ret
```

A comment immediately before the procedure might help separate it from previous code. However, the person reading the code has to work to differentiate the `zeroBytes` label from the `repeatlp` label. In fact, there's no reason you couldn't use both labels as entry points for a procedure (`zeroBytes` would always zero out 256 words starting at the address passed in X0, and `repeatlp` would zero out the number of words specified in X1/4). Of course, a procedure isn't required to use just a single `ret` instruction (or any at all, since there are other ways to return from a procedure). The last instruction of a procedure also doesn't have to be a `ret`. Therefore, relying on a statement label and a `ret` instruction to delineate the procedure is not always appropriate.

Though it's always a good idea to put comments at the beginning and end of your Gas procedures to clarify what's happening, the best way to solve this problem would be to use *syntactical sugar*—statements that clarify meaning without generating any code—to delineate procedures. Although Gas does not provide such statements, you can write your own macros for the same purpose. The *aoaa.inc* include file provides a couple of these macros: `proc` and `endp`. Here is their syntax:

```
proc procedureName {, public}  // Braces denote optional item.

     Body of the procedure

endp procedureName
```

Here, *`procedureName`* will be the name of the procedure, and you must supply the same name in the `proc` and `endp` statements. The `, public` argument is optional, as denoted by the meta-symbol braces. If the `public` argument is present, the `proc` macro will automatically generate a `.global` directive for the procedure.

Here's a very simple example of using the `proc` and `endp` macros with the `getTitle` function:

```
proc    getTitle, public
lea     x0, ttlStr
ret
endp    getTitle
```

These macros generate the usual statements for the `getTitle` procedure:

```
            .global     getTitle  // Generated by public
getTitle:   // Generated by proc
            lea         x0, ttlStr
            ret
```

The `endp` macro doesn't generate anything in the program. It simply checks the identifier passed as an argument to ensure that it matches the procedure's name in the `proc` macro invocation.

Because the `proc` and `endp` statements neatly isolate a procedure's body from other code in the program, this book uses them for procedures from this point forward. I suggest you take advantage of these macros to help make your own future procedures more readable too.

Procedures and functions in an HLL provide useful features in the form of local symbols. The next section covers the limited form of local labels supported by Gas.

5.2.1 Gas Local Labels

Unlike HLLs, Gas does not support *lexically scoped symbols*. Labels you define in a procedure are not limited in scope to that procedure. Except for one special case, symbols you define in a Gas procedure, including those defined with `proc`/`endp`, are visible throughout the source file.

However, Gas does support a limited form of *local labels*, which consist of a single numeric digit followed by a colon (`0:` through `9:`). In your code, refer to these symbols by using `Nb` or `Nf`, where `N` is the digit (`0` through `9`). A symbol of the form `Nb` references the previous `N:` label in the source file (`b` is for *backward*). A symbol of the form `Nf` references the next `N:` symbol in the source file (`f` is for *forward*).

Here's an example of a Gas local label in the `zeroBytes` procedure (rewritten from the previous section):

```
          proc zeroBytes
          mov  x1, #256 * 4
0:        subs x1, x1, #4
          str  wzr, [x0, x1] // Store *after* subtraction!
          bne  0b            // Repeat while X1 != 0.
          ret
          endp zeroBytes
```

Local labels are useful when there is no compelling reason to use a more meaningful name. Be careful about using these local symbols, though. When used sparingly, they help reduce the distraction of meaningless labels in your program, but using too many will destroy the readability of your programs ("to which `0` label is this code jumping?").

When you use local labels, your target label should be only a few instructions away; if the code jumps any great distance, you run the risk of inserting that same local label between the source and targets when

enhancing your code later. This would produce undesirable consequences, and Gas won't notify you of the error.

5.2.2 bl, ret, and br

Once you can declare a procedure, the next problem is how to call (and return from) a procedure. As you've seen many times throughout this book, you call procedures by using `bl` and return from those procedures by using `ret`. This section covers those instructions (as well as the `br` instruction) in more detail, including the effects of their use.

The ARM `bl` instruction does two things: it copies the (64-bit) address of the instruction immediately following the `bl` to the LR register, and then it transfers control to the address of the specified procedure. The value that `bl` copies to LR is known as the *return address*.

When a procedure wants to return to the caller and continue execution with the first statement following the `bl` instruction, that procedure commonly returns to its caller by executing a `ret` instruction. The `ret` instruction transfers control indirectly to that address held in the LR register (X30).

The ARM `ret` instruction takes two forms

```
ret
ret reg64
```

where *reg*$_{64}$ is one of the ARM's thirty-two 64-bit registers. If a 64-bit register operand appears, the CPU uses the address held in that register as the return address; if no register is present, the default is X30 (LR).

The `ret` instruction is actually a special case of the `br` (branch indirect through register) instruction. The `br` syntax is

```
br  reg64
```

where *reg*$_{64}$ is one of the ARM's thirty-two 64-bit registers. This instruction also transfers control to the address held in the specified register. Whereas the `ret` *reg*$_{64}$ instruction provides a hint to the CPU that this is an actual return-from-subroutine, the `br` *reg*$_{64}$ instruction offers no such hint. In some circumstances, the ARM can execute the code faster if it's given the hint. Chapter 7 covers some uses for the `br` instruction.

The following is an example of the `minimal` Gas procedure:

```
proc minimal
ret
endp minimal
```

If you call this procedure with the `bl` instruction, `minimal` will simply return to the caller. If you fail to put the `ret` instruction in the procedure, the program will not return to the caller upon encountering the `endp` statement. Instead, the program will fall through to whatever code happens to follow the procedure in memory.

Listing 5-2 demonstrates this problem. The main program calls `noRet`, which falls straight through to `followingProc` (printing the `followingProc was called` message).

```
// Listing5-2.S
//
// A procedure without a ret instruction

#include "aoaa.inc"

stackSpace  =           64
saveLR      =           56

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 5-2"
fpMsg:      .asciz      "followingProc was called\n"

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// noRet
//
// Demonstrates what happens when a procedure
// does not have a return instruction

            proc    noRet
            endp    noRet

            proc    followingProc
            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]

            lea     x0, fpMsg
            bl      printf

            ldr     lr, [sp, #saveLR]
            add     sp, sp, #stackSpace
            ret
            endp    followingProc

// Here is the asmMain function:

            proc    asmMain, public
            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]
```

```
            bl      noRet

            ldr     lr, [sp, #saveLR]
            add     sp, sp, #stackSpace
            ret
            endp    asmMain
```

As you can see, there is no `ret` instruction in `noRet`, so when the main program (`asmMain`) calls `noRet`, it will fall straight through into `followingProc`. Here's the `build` command and sample execution:

```
$ ./build Listing5-2
$ ./Listing5-2
Calling Listing5-2:
followingProc was called
Listing5-2 terminated
```

Although this behavior might be desirable in certain rare circumstances, it usually represents a defect in most programs. Therefore, always remember to explicitly return from the procedure by using the `ret` instruction.

5.3 Saving the State of the Machine

Listing 5-3 attempts to print 20 lines of 40 spaces and an asterisk.

```
// Listing5-3.S
//
// Preserving registers (failure) example

#include "aoaa.inc"

stackSpace  =           64
saveLR      =           56
saveX19     =           48

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 5-3"
space:      .asciz      " "
asterisk:   .asciz      "*, %d\n"

            .data
loopIndex:  .word       .-.     // Used to print loop index value

            .code
            .extern     printf

// getTitle
//
// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
```

```
            ret
            endp    getTitle

// print40Spaces
//
// Prints out a sequence of 40 spaces
// to the console display

            proc    print40Spaces
            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]

            mov     w19, #40
printLoop:  lea     x0, space
            bl      printf
            subs    w19, w19, #1
            bne     printLoop // Until W19 == 0
            ldr     lr, [sp, #saveLR]
            add     sp, sp, #stackSpace
            ret
            endp    print40Spaces

// Here is the asmMain function:

            proc    asmMain, public

            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]   // Save return address.
            str     x19, [sp, #saveX19] // Must preserve nonvolatile register.

            mov     w19, #20
astLp:      bl      print40Spaces
            lea     x0, loopIndex
            str     w19, [x0]
            lea     x0, asterisk
            vparm2  loopIndex
            bl      printf
            subs    w19, w19, #1
            bne     astLp

            ldr     x19, [sp, #saveX19]
            ldr     lr, [sp, #saveLR]
            add     sp, sp, #stackSpace
            ret     // Returns to caller
            endp    asmMain
```

Unfortunately, a subtle bug creates an infinite loop. The main program uses the `bne printLoop` instruction to create a loop that calls `Print40Spaces` 20 times. This function uses W19 to count off the 40 spaces it prints, and then returns with W19 containing 0. The main program prints an asterisk and a newline, decrements W19, and then repeats because W19 isn't 0 (it will always contain –1 at this point).

The problem here is that the `print4OSpaces` subroutine doesn't preserve the W19 register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the `print4OSpaces` subroutine preserved the contents of the W19 register, Listing 5-3 would have functioned properly. There is no need to build and run this program; it just runs in an infinite loop.

Consider the following code for `print4OSpaces`:

```
            proc    print4OSpaces
            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]
            str     x19, [sp, #saveX19]

            mov     w19, #4O
printLoop:  lea     xO, space
            bl      printf
            subs    w19, w19, #1
            bne     printLoop // Until W19 == O
            ldr     lr, [sp, #saveLR]
            ldr     x19, [sp, #saveX19]
            add     sp, sp, #stackSpace
            ret
            endp    print4OSpaces
```

This variant of `print4OSpaces` saves and restores X19 on the stack, along with the LR register. Because X19 is a nonvolatile register (in the ARM ABI), it is the responsibility of the callee (the procedure) to preserve it.

Note that `print4OSpaces` uses X19 rather than one of the X0 to X15 registers specifically because it is nonvolatile. The `printf()` function does not have to preserve X0 to X15 because they are *volatile* registers in the ARM ABI. Any attempt to use those registers would have likely failed because `printf()` doesn't have to preserve their values.

In general, either the caller (the code containing the call instruction) or the callee (the subroutine) can take responsibility for preserving the registers. When following the ARM ABI, it is the caller's responsibility to preserve volatile registers and the callee's responsibility to preserve nonvolatile registers. Of course, when writing your own procedures that won't be called by ABI-compliant functions and don't call any ABI-compliant functions, you can choose whichever register preservation scheme you prefer.

Listing 5-4 shows the corrected version of the program in Listing 5-3, which properly preserves X19 in the call to `print4OSpaces`.

```
// Listing5-4.S
//
// Preserving registers (successful) example

#include "aoaa.inc"

stackSpace  =           64
saveLR      =           56
saveX19     =           48
```

```
            .section    .rodata, ""
ttlStr:     .asciz      "Listing 5-4"
space:      .asciz      " "
asterisk:   .asciz      "*, %d\n"

            .data
loopIndex:  .word       .-.     // Used to print loop index value

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// print40Spaces
//
// Prints out a sequence of 40 spaces
// to the console display

            proc    print40Spaces
            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]
            str     x19, [sp, #saveX19]

            mov     w19, #40
printLoop:  lea     x0, space
            bl      printf
            subs    w19, w19, #1
            bne     printLoop // Until W19 == 0
            ldr     lr, [sp, #saveLR]
            ldr     x19, [sp, #saveX19]
            add     sp, sp, #stackSpace
            ret
            endp    print40Spaces

// Here is the asmMain function:

            proc    asmMain, public

            sub     sp, sp, #stackSpace
            str     lr, [sp, #saveLR]   // Save return address.
            str     x19, [sp, #saveX19] // Must preserve nonvolatile register.

            mov     w19, #20
astLp:      bl      print40Spaces
            lea     x0, loopIndex
            str     w19, [x0]
            lea     x0, asterisk
            vparm2  loopIndex
            bl      printf
```

```
        subs    w19, w19, #1
        bne     astLp

        ldr     lr, [sp, #saveLR]
        ldr     x19, [sp, #saveX19]
        add     sp, sp, #stackSpace
        ret     // Returns to caller
        endp    asmMain
```

Here's the `build` command and sample output for Listing 5-4:

```
$ ./build Listing5-4
$ ./Listing5-4
Calling Listing5-4:
                                        *, 20
                                        *, 19
                                        *, 18
                                        *, 17
                                        *, 16
                                        *, 15
                                        *, 14
                                        *, 13
                                        *, 12
                                        *, 11
                                        *, 10
                                        *, 9
                                        *, 8
                                        *, 7
                                        *, 6
                                        *, 5
                                        *, 4
                                        *, 3
                                        *, 2
                                        *, 1
Listing5-4 terminated
```

As you can see, this program executes properly without entering an infinite loop.

Callee preservation has two advantages: space and maintainability. If the callee (the procedure) preserves all affected registers, only one copy of the `str` and `ldr` instructions exists—those that the procedure contains. If the caller saves the values in the registers, the program needs a set of preservation instructions around every call. This makes your programs not only longer but also harder to maintain. It's not easy to remember which registers to save and restore on each procedure call.

On the other hand, a subroutine may unnecessarily preserve some registers if it preserves all the registers it modifies. If the caller is preserving the registers, the subroutine doesn't have to save registers it doesn't care about.

One big problem with having the caller preserve registers is that your program may change over time. You may modify the calling code or the procedure to use additional registers. Such changes, of course, may change

the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate *every* call to the routine and verify that the subroutine does not change any registers that the calling code uses.

Assembly language programmers typically use a common convention with respect to register preservation: unless there is a good reason (performance) for doing otherwise, most programmers will preserve each register that a procedure modifies (and doesn't explicitly return a value in a modified register). This reduces the likelihood of defects occurring in a program because a procedure modifies a register the caller expects to be preserved. Of course, you could follow the rules concerning the ARM ABI with respect to volatile and nonvolatile registers; however, such calling conventions impose their own inefficiencies on both programmers and other programs. This book generally adheres to the ARM ABI with respect to volatile and nonvolatile registers, though many examples preserve all affected registers in a procedure.

There's more to preserving the environment than preserving registers. You can also preserve variables and other values that a subroutine might change.

5.4 Call Trees, Leaf Procedures, and the Stack

Imagine a procedure, A, that calls two other procedures B and C. Also assume that B calls two procedures D and E, and procedure C calls two other procedures F and G. We can diagram this calling sequence by using a *call tree*, as shown in Figure 5-1.

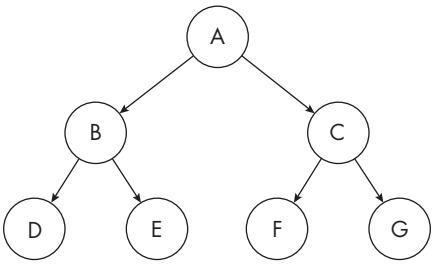


Figure 5-1: A call-tree diagram

This entire call graph is the tree, and the procedures at the bottom that do not call any other procedures—in this case, D, E, F, and G—are known as *leaf procedures*.

Leaf procedures are different from non-leaf procedures in ARM assembly language because they can leave the return address in the LR register rather than saving it to memory (the stack). As leaf procedures don't make any other calls via the `bl` instruction, the procedure won't disturb the value in LR upon entry into the procedure. (This assumes that the procedure doesn't explicitly modify LR, but generally, there is no good reason for doing so.) Therefore, leaf procedures can be slightly more efficient than

non-leaf procedures, as they are spared the need to preserve the value in the LR register. Leaf procedures can also make full use of the volatile register set without worrying about their values being scrambled during a call to another procedure.

Non-leaf procedures must preserve the value in the LR register because calls they make (via `bl`) will overwrite the value in LR. A procedure can preserve LR in a few places: in another register, on a stack, or in a global memory location, as our examples did before Chapter 3 introduced the stack.

I've already pointed out that using global variables to preserve LR is a poor choice in nearly every case. That scheme can handle only one level of calls and completely fails when using recursion (see section 5.8, "Recursion," on page 277) or writing multithreaded applications. It's also slower, uses more code, and is less convenient to use than other schemes.

You could use another register to temporarily hold the return address while calling another procedure. Of course, that register must be nonvolatile (or, at least, the procedure you're calling must not modify that register's value) so that it will still contain the saved return address whenever the procedure you call returns. Using a register to preserve LR like this is very fast. Unfortunately, guaranteeing that other procedures won't modify the saved value often means you have to preserve that value in memory within the second procedure to be called. Since you still have to write the value to memory (and read it back), you may as well have saved LR directly to memory in the first place.

The most common place to save the return address in LR is on the stack. Usually, one of the first instructions in a procedure will move the contents of the LR register into the stack. This is typically done in one of two ways. The first is to directly push the LR register onto the stack:

```
str lr, [sp, #-16]!
```

The second is to adjust the stack down in memory and store LR into the storage area just created:

```
sub sp, sp, #someAmount   // Make room for LR on stack.
str lr, [sp, #someOffset] // Store LR into space allocated on stack.
```

Here, *someAmount* is a multiple of 16 (or another value that keeps the stack 16-byte aligned), and *someOffset* is an index into the space just allocated on the stack by the `sub` instruction.

Notice that the former example uses the pre-indexed addressing mode to adjust SP downward and store LR into the vacated space (because of stack alignment issues, this actually reserves 16 bytes, though it uses only 8 of them). The latter example uses the indirect-plus-offset addressing mode to simply store the return address into the storage allocated by the `sub` instruction. This book most commonly uses the latter form because the cost of `sub` is often shared by other code that uses the stack.

Wasting 8 bytes by using the pre-indexed addressing mode won't turn out to be an issue. As you'll see shortly, most of the time you'll want to

preserve the value of the FP register along with the return address, so you'll commonly use an `stp` instruction, like one of the following, that won't waste any memory:

```
stp fp, lr, [sp, #-16]!
```

or

```
sub sp, sp, #someAmount
stp fp, lr, [sp, #someOffset]
```

The following subsections cover the use of the stack in procedures, including activation records, accessing data within activation records (local and automatic variables as well as parameters), how the ARM ABI influences activation records and passing parameters, and how to build and destroy activation records.

5.4.1 Activation Records

When you call a procedure, the program associates certain information with that procedure call, including the return address, parameters, and automatic local variables (which I'll discuss in later sections). To do so, it uses a data structure called an *activation record*, also known as a *stack frame*. The program creates an activation record when calling (activating) a procedure, and the data in the record is organized in a manner identical to structs.

This section covers traditional activation records created by a hypothetical compiler, ignoring the parameter-passing conventions of the ARM ABI. A later section of this chapter presents the ARM ABI conventions.

Construction of an activation record begins in the code that calls a procedure. The caller makes room for the parameter data (if any) on the stack and copies the data onto the stack. The `bl` instruction then passes the return address into the procedure. At this point, construction of the activation record continues within the procedure itself. The procedure typically pushes the value in LR onto the stack along with other registers and other important state information, then makes room in the activation record for local variables. The procedure might also update the FP register (X29) so that it points at the base address of the activation record.

To see what a traditional activation record looks like, consider the following C++ procedure declaration:

```
void ARDemo(unsigned i, int j, unsigned k)
{
     int a;
     float r;
     char c;
     bool bb;
```

```
    short w;
     .
     .
     .
}
```

Whenever a program calls this `ARDemo` procedure, it begins by pushing the data for the parameters onto the stack. In the original C/C++ calling convention (ignoring the ARM ABI), the calling code pushes all the parameters onto the stack in the opposite order in which they appear in the parameter list, from right to left. Therefore, the calling code pushes first the value for the `k` parameter, then the value for the `j` parameter, and finally the data for the `i` parameter (with possible padding for the parameters to keep the stack aligned).

Next, the program calls `ARDemo`. Immediately upon entry into the `ARDemo` procedure, the stack contains these three items arranged as shown in Figure 5-2. Since the program pushes the parameters in reverse order, they appear on the stack in the correct order, with the first parameter at the lowest address in memory.

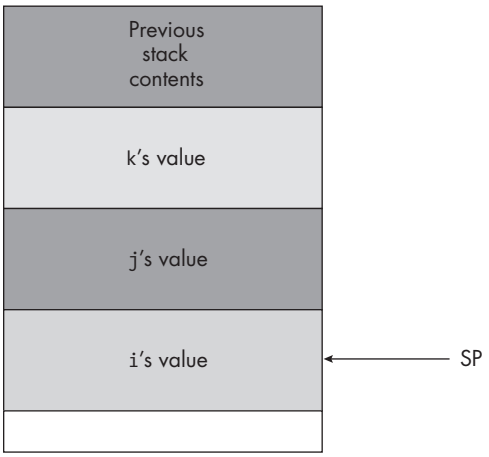


Figure 5-2: Stack organization immediately upon entry into `ARDemo`

The first few instructions in `ARDemo` push the current values of LR and FP onto the stack, then copy the value of SP into FP. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 5-3.

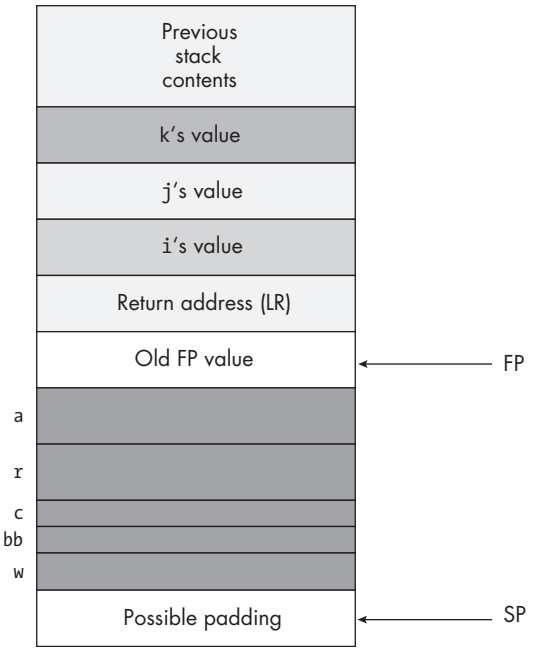


Figure 5-3: The activation record for `ARDemo`

Because local variables can be any size in the activation record, their total storage might not be a multiple of 16 bytes. However, the entire block of local variables must be a multiple of 16 bytes so that SP remains aligned on a 16-byte boundary as required by the ARM CPU—hence the presence of possible padding in Figure 5-3.

5.4.2 Objects in the Activation Record

To access objects in the activation record, you can use offsets from the FP register to the desired object. The two items of immediate interest to you are the parameters and the local variables. You can access the parameters at positive offsets from the FP register; you can access the local variables at negative offsets from the FP register, as Figure 5-4 shows (the figure assumes that the `i`, `j`, and `k` parameters are all 64-bit integers with appropriate padding to 8 bytes each).

ARM specifically reserves the X29/FP register for use as a pointer to the base of the activation record. This is why you should avoid using the FP register for general calculations. If you arbitrarily change the value in the FP register, you could lose access to the current procedure's parameters and local variables.

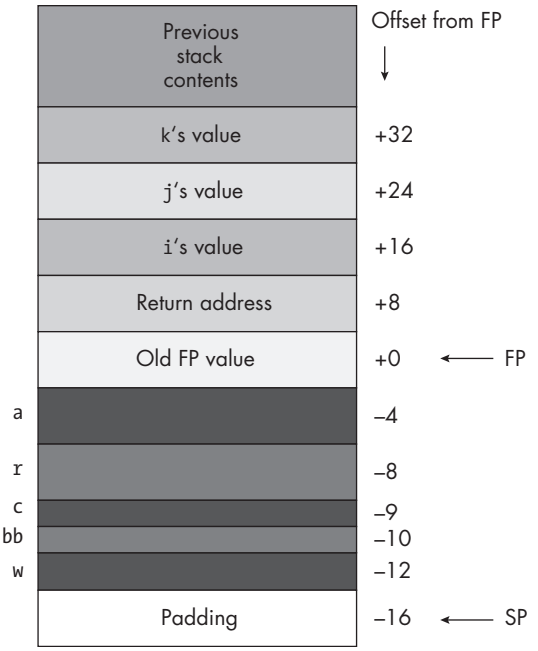


Figure 5-4: Offsets of objects in the `ARDemo` activation record

The local variables are aligned on offsets that are equal to their native size: chars are aligned on 1-byte addresses; shorts/hwords are aligned on 2-byte addresses; longs, ints, unsigned, and words are aligned on 4-byte addresses; and so forth. In the `ARDemo` example, all the locals just happen to be allocated on appropriate addresses (assuming a compiler allocates storage in the order of declaration).

5.4.3 ARM ABI Parameter-Passing Conventions

The ARM ABI makes several modifications to the activation record model:

- The caller passes the first eight (non-floating-point) parameters in registers (X0 through X7) rather than on the stack.
- Parameters are always 8-byte values, either in registers or on the stack (if the formal parameter is fewer than 8 bytes in size, the unused HO bits are undefined).
- Structures and unions greater than 16 bytes in size are passed by value on the stack above any other parameters, but with a pointer to the value in the normal parameter position (in a register or on the stack). Structs and unions that are 8 bytes (or fewer) are passed in a 64-bit register; those that are 9 to 16 bytes are passed in two consecutive registers.

You must follow these conventions only when calling ARM ABI–compliant code. For assembly language procedures that you write and call, you can use any convention you like.

Apple's calling conventions for macOS (iOS, iPadOS, and so on) vary a little from the standard ARM ABI. This will affect your assembly code if you're doing the following:

- Passing more than eight parameters to a procedure
- Passing parameters to a variadic procedure

When passing parameters on the stack—that is, when you're passing more than eight arguments to a function—Apple packs them on the stack, meaning it doesn't simply allocate 8 bytes for each parameter on the stack. It does ensure that each value is aligned in memory on its natural size (chars = 1 byte, half words = 2 bytes, words = 4 bytes, and so on).

Variadic procedures are those with a variable number of parameters, such as the C `printf()` function. Apple passes all variadic parameters on the stack and allocates 8 bytes for each parameter, regardless of type. This is the purpose behind the `vparm2`, `vparm3`, . . . , macros in *aoaa.inc*: calls to `printf()` under macOS must pass the arguments on the stack, while the same calls on Linux pass the first eight parameters in registers.

The `vparm2`, `vparm3`, and so on, macros automatically generate the appropriate code based on the OS (either putting the parameters in the stack or passing them in registers).

5.4.4 Standard Entry Sequence

The caller of a procedure is responsible for allocating storage for parameters on the stack and moving the parameter data to its appropriate location. In the simplest case, this just involves moving the data onto the stack by using `str` or `stp` instructions. It is the procedure's responsibility to construct the rest of the activation record. You can accomplish this by using the following assembly language *standard entry sequence* code:

```
stp fp, lr, [sp, #-16]!   // Save LR and FP values.
mov fp, sp                // Get activation record ptr in FP.
sub sp, sp, #NumVars      // Allocate local storage.
```

The `mov fp, sp` instruction copies the current address held in SP into the FP register. As SP is currently pointing at the old value of FP pushed on the stack, FP will point at the original FP value after the execution of this instruction, as shown in Figure 5-4. When using the `stp` instruction in the standard entry sequence, make sure to specify the FP register as the first argument so that it is stored at location `[SP]` and LR is stored at location `[SP, #8]`. This ensures that FP will point at the old FP value after the `mov` instruction.

In the third instruction, *`NumVars`* represents the number of bytes of local variables needed by the procedure, a constant that should be a multiple of 16 so that the SP register remains aligned on a 16-byte boundary. If the number of bytes of local variables in the procedure is not a multiple of 16, round up the value to the next higher multiple of 16 before subtracting this constant from SP. Doing so will slightly increase the amount of storage the procedure uses for local variables but will not otherwise affect the

operation of the procedure. If the procedure doesn't have any local variables or call any other functions, the

```
sub sp, sp, #NumVars
```

instruction isn't necessary.

In theory, you could use any register to access the data in the stack frame. However, the OS, and especially debugger applications, often depend on the activation record being built with FP pointing at the old FP value in the activation record.

If an ARM ABI–compliant program calls your procedure, the stack will be aligned on a 16-byte boundary immediately prior to the execution of the `bl` instruction. Pushing LR and FP onto the stack (before copying SP into FP) adds another 16 bytes to the stack so that SP remains 16-byte aligned. Therefore, assuming the stack was 16-byte aligned prior to the call, and the number you subtract from SP is a multiple of 16, the stack will be 16-byte aligned after allocating storage for local variables.

The `ARDemo` activation record from the previous section has only 12 bytes of local storage. Therefore, subtracting 12 from SP for the local variables will not leave the stack 16-byte aligned. The entry sequence in the `ARDemo` program must subtract 16 (which will include 4 bytes of padding) to keep the stack properly aligned (as shown in Figure 5-4).

A possible alternate entry code sequence that is equivalent to the earlier example takes this form:

```
sub sp, sp, #numVars + 16   // Space for locals and SP/LR
stp fp, lr, [sp, #numVars]
add fp, sp, #numVars
```

The ARM ABI calling convention suggests saving the LR and FP values *below* the local variables. However, it is often convenient to allocate parameter space for additional procedure calls (from the current procedure) while allocating local variables. If you save the LR and FP values at the bottom of the activation record in memory, you will need an extra instruction to make room for those parameters, and cleaning up the activation record will take more effort when the procedure returns.

Because you'll so often use the standard entry sequence, the *aoaa.inc* include file provides a macro to generate this sequence for you:

```
enter numVars
```

The single constant argument is the amount of stack space to allocate (for local variables and other memory objects) in addition to the 16 bytes set aside to preserve the LR and FP registers. This macro generates the following sequence of instructions for the entry sequence:

```
stp fp, lr, [sp, #-16]!
mov fp, sp
sub sp, sp, #(numVars + 15) & 0xFFFFFFFFFFFFFFF0
```

The final expression involving *numVars* ensures that the space allocated on the stack is a multiple of 16 bytes, to keep the stack 16-byte aligned.

5.4.5 Standard Exit Sequence

The standard exit sequence for an assembly language program is the following:

```
mov sp, fp              // Deallocates storage for all the local vars
ldp fp, lr, [sp], #16   // Pop FP and return address.
ret                     // Return to caller.
```

In the *aoaa.inc* include file, the `leave` macro expands to the original standard exit sequence.

5.5 Local Variables

Procedures and functions in most HLLs let you declare local variables (also known as *automatic variables*). The previous sections mentioned that procedures maintain local variables in an activation record, but they didn't really define how to create and use them. This section (and the subsections that follow) defines local variables and describes how to allocate storage for them and use them.

Local variables possess two special attributes in HLLs: scope and lifetime. The *scope* of an identifier determines where that identifier is visible (accessible) in the source file during compilation. In most HLLs, the scope of a procedure's local variable is the body of that procedure; the identifier is inaccessible outside that procedure. Sadly, Gas does not support locally scoped variables in a procedure, since Gas has no syntax to determine the bounds of a procedure.

Whereas scope is a compile-time attribute of a symbol, lifetime is a runtime attribute. The *lifetime* of a variable is a range of time, from that point when storage is first bound to the variable until the point where the storage is no longer available for that variable. Static objects (those you declare in the `.data`, `.rodata`, `.bss`, and `.text` sections) have a lifetime equivalent to the total runtime of the application. The program allocates storage for such variables when the program first loads into memory, and those variables maintain that storage until the program terminates.

Local variables, more properly known as *automatic variables*, have their storage allocated upon entry into a procedure. That storage is then returned for other use when the procedure returns to its caller. The name *automatic* refers to the program automatically allocating and deallocating storage for the variable on procedure invocation and return.

Under Linux, a procedure can access any global `.data`, `.bss`, or `.rodata` object exactly the same way the main program accesses such variables: by referencing the name, using the PC-relative addressing mode (sadly, macOS's PIE format doesn't allow easy access to non-`.text` section objects). Accessing global objects is convenient and easy. However, accessing global

objects makes your programs harder to read, understand, and maintain, so you should avoid using global variables within procedures.

Although accessing global variables within a procedure may sometimes be the best solution to a given problem, you likely won't be writing such code at this point, so carefully consider your options before doing so. (An example of a legitimate use of global variables might be when sharing data between threads in a multithreaded application, a bit beyond the scope of this chapter.)

This argument against accessing global variables does not apply to other global symbols, however. It is perfectly reasonable to access global constants, types, procedures, and other objects in your programs.

5.5.1 Low-Level Implementation of Automatic Variables

Your program accesses local variables in a procedure by using negative offsets from the activation record base address (FP). Consider the Gas procedure in Listing 5-5, which is intended primarily to demonstrate the use of local variables.

```
// Listing5-5.S
//
// Accessing local variables

#include "aoaa.inc"

               .text

// local_vars
//
// Word a is at offset -4 from FP.
// Word bb is at offset -8 from FP.
//
// On entry, W0 and W1 contain values to store
// into the local variables a & bb (respectively).

            proc    local_vars
            enter   8

            str     w0, [fp, #-4]   // a = W0
            str     w1, [fp, #-8]   // bb = W1

    // Additional code here that uses a & bb

            leave
            endp    local_vars
```

This program isn't runnable, so I won't bother providing a `build` command for it. The `enter` macro will actually allocate 16 bytes of storage, rather than the 8 specified by the argument (for locals `a` and `bb`), in order to keep the stack 16-byte aligned.

The activation record for `local_vars` appears in Figure 5-5.

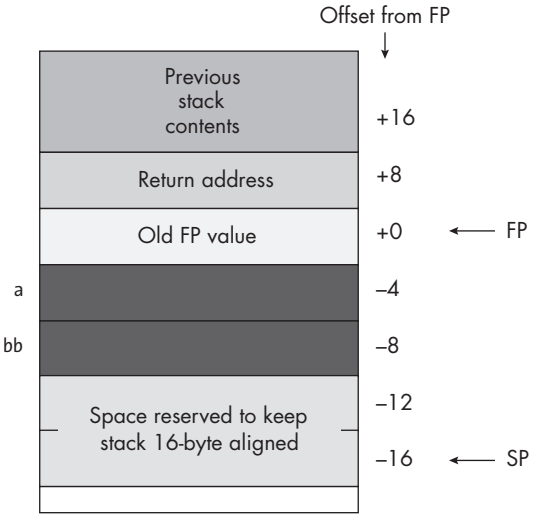


Figure 5-5: The activation record for the `local_vars` procedure

Of course, having to refer to the local variables by the numeric offset from the FP register is truly horrible. This code is not only difficult to read (Is `[FP, #-4]` the `a` or the `bb` variable?) but also hard to maintain. For example, if you decide you no longer need the `a` variable, you'd have to go find every occurrence of `[FP, #-8]` (accessing the `bb` variable) and change it to `[FP, #-4]`.

A slightly better solution is to create equates for your local variable names. Consider the modification to Listing 5-5 shown in Listing 5-6.

```
// Listing5-6.S
//
// Accessing local variables #2

#include "aoaa.inc"

            .code

// local_vars
//
// Demonstrates local variable access
//
// Word a is at offset -4 from FP.
// Word bb is at offset -8 from FP.
//
// On entry, W0 and W1 contain values to store
// into the local variables a & bb (respectively).

#define a [fp, #-4]
#define bb [fp, #-8]

            proc    local_vars
            enter   8
```

```
        str     w0, a
        str     w1, bb

    Additional code here that uses a & bb.

        leave
        endp    local_vars
```

In Listing 5-6, the CPP replaces `a` and `bb` with the appropriate indirect-plus-offset addressing mode to access those local variables on the stack. This is considerably easier to read and maintain than the program in Listing 5-5. However, this approach still requires some manual work to set the local variable offsets in the `#define` statements, and modifying the code (when adding or removing local variables) can create maintenance issues. I'll provide a better solution in the next section.

One big advantage to automatic storage allocation is that it efficiently shares a fixed pool of memory among several procedures. For example, say you call three procedures in a row, like this:

```
bl ProcA
bl ProcB
bl ProcC
```

In this example, `ProcA` allocates its local variables on the stack. Upon return, `ProcA` deallocates that stack storage. Upon entry into `ProcB`, the program allocates storage for `ProcB`'s local variables by using the same memory locations just freed by `ProcA`. Likewise, when `ProcB` returns and the program calls `ProcC`, `ProcC` uses the same stack space for its local variables that `ProcB` recently freed up. This memory reuse makes efficient use of the system resources and is probably the greatest advantage to using automatic variables.

Now that you've seen how assembly language allocates and deallocates storage for local variables, it's easy to understand why automatic variables do not maintain their values between two calls to the same procedure. Once the procedure returns to its caller, the storage for the automatic variable is lost, and, therefore, the value is lost as well. Thus, *you must always assume that a local variable object is uninitialized upon entry into a procedure.* If you need to maintain the value of a variable between calls to a procedure, you should use one of the static variable declaration types.

5.5.2 The locals Macro

Using equates to maintain local variable references is a lot of work. Granted, it's better than using magic numbers in all your local variable references, but even when using equates, inserting and deleting local variables in a procedure takes time and effort. What would be really nice is a declaration section that lets you declare your local variables in an HLL-like fashion and leave it up to the assembler to maintain all the offsets into the activation record. The *aoaa.inc* header file provides a set of macros you

can use to automate the creation of local variables. This section describes those macros.

The activation record is a *record* (structure). In theory, you could use the `struct` macro from Chapter 4 to define an activation record. However, it's easy enough to modify the `struct`/`ends` macros to create something even better for local variables. To achieve that, the *aoaa.inc* include file includes two additional macros for declaring local variables: `locals` and `endl`. Use these in a manner almost identical to the `struct`/`ends` macros

```
locals procName
  declarations (same as for struct)
endl   procName
```

where *procName* is an identifier (usually the name of the procedure that the local variables are associated with).

Like the `ends` macro, `endl` generates a symbol with the name *procName*`.size` that is an equate set to the size of the local variable space. You can supply this value to the `enter` macro to specify the amount of space to reserve for the local variables:

```
    proc   myProc

    locals myProc
    dword  mp.ptrVar
    word   mp.counter
    byte   mp.inputChar
    salign 4
    word   mp.endIndex
    endl   myProc

    enter  myProc.size

Insert procedure's body here.

    leave
    endp   myProc
```

The `locals`/`endl` declarations create a set of equates whose values correspond to the offsets of the symbols within an activation record. For example, the symbols in the previous example have the following values:

`mp.ptrVar` –8

`mp.counter` –12

`mp.inputChar` –13

`mp.endIndex` –20

You can use these offsets with the `[FP, #`*offset*`]` addressing mode to reference these local variables in the activation record. For example:

```
ldr w0, [fp, #mp.counter]
ldr x1, [fp, #mp.ptrVar]
str w0, [x1]
```

This is a whole lot easier than accessing global variables in the `.data` section!

When allocating offsets for variables between the `locals` and `endl` macros, the declaration macros first decrease the offset counter by the size of the variable's declaration and then assign the decremented offset value to the symbol. Specifying the `salign` directive will then adjust the offset to the specified boundary (2^n, where n is the `salign` operand's value). The next declaration will not use this offset, but rather it will first decrement the running offset counter by the size of the declaration and assign that offset to the variable. In the earlier example, the `salign` directive set the running offset to –16 (because 13 bytes of variables were allocated at that point). The following variable's offset is –20, because `mp.endIndex` consumes 4 bytes.

As I mentioned earlier, Gas does not support the concept of lexically scoped local variable names, which are private to a procedure. Therefore, all symbols you declare within the `locals`/`endl` block are visible throughout the source file. This can lead to *namespace pollution*, where you wind up creating names in one procedure and cannot reuse those names in a different procedure.

In the examples of this section, I use a convention that I continue throughout this book to alleviate namespace pollution: I use local variable names of the form *`proc.local`*, where *`proc`* is the procedure's name (or an abbreviation of the procedure's name) and *`local`* is the specific local variable name I want to use. For example, `mp.ptrVar` is the `ptrVar` local variable within the `myProc` (`mp`) procedure.

5.6 Parameters

Although many procedures are totally self-contained, most procedures require input data and return data to the caller (parameters).

The first aspect to consider when discussing parameters is how we pass them to a procedure. If you are familiar with Pascal or C/C++, you've probably seen two ways to pass parameters: pass by value and pass by reference. Anything you can do with an HLL can be done in assembly language (obviously, as HLL code compiles into machine code), but you have to provide the instruction sequence to access those parameters in an appropriate fashion.

Another concern when dealing with parameters is *where* you pass them. There are many places to pass parameters: in registers, on the stack, in the code stream, in global variables, or a combination of these. The following subsections cover several of the possibilities.

5.6.1 Passing by Value

A parameter *passed by value* is just that—the caller passes a value to the procedure. Pass-by-value parameters are input-only parameters. You can

pass them to a procedure, but the procedure cannot return values through them. Consider this C/C++ function call:

```
CallProc(I);
```

If you pass `I` by value, `CallProc()` does not change the value of `I`, regardless of what happens to the parameter inside `CallProc()`.

Because you must pass a copy of the data to the procedure, you should use this method only for passing small objects like bytes, words, double words, and quad words. Passing large arrays and records by value is inefficient, because you must create and pass a copy of the object to the procedure.

5.6.2 Passing by Reference

To pass a parameter *by reference*, you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures. Because pointers on the ARM are 64 bits wide, a parameter that you pass by reference will consist of a double-word value, typically in one of the general-purpose registers.

You can use the `lea` macro to take the address of any static variable you've declared in your `.data`, `.bss`, `.rodata`, or `.text` sections. Listing 5-7 demonstrates how to obtain the address of a static variable (`staticVar`) and pass that address to a procedure (`someFunc`) in the X0 register.

```
// Listing5-7.S
//
// Demonstrate obtaining the address
// of a variable by using the lea instruction.

#include "aoaa.inc"

            .data
staticVar:  .word   .-.

            .code
            .extern someFunc

            proc    get_address
            enter   0
            lea     x0, staticVar
            bl      someFunc
            leave
            endp    get_address
```

Calculating the address of a nonstatic variable is a bit more work. Unfortunately, the `adr` and `adrp` instructions compute only the address of a PC-relative memory access. If your variable is referenced by one of the other

ARM addressing modes, you'll have to manually compute the effective address yourself.

Table 5-1 describes the process for effective address calculation. In the table, the [Xn, #const] (scaled form) addressing mode describes a machine encoding, not an assembler syntax. In source code, the scaled and unscaled variants share the same syntax: [Xn, #const]. The assembler will pick the correct machine encoding based on the value of the constant.

Table 5-1: Effective Address Calculations

Addressing mode	Effective address	Description
[Xn]	Xn	For the register-indirect addressing mode, the effective address is just the value held in the register.
[Xn, #const]	Xn + *const*	For the indirect-plus-offset addressing mode, the sum of the Xn register and the constant is the effective address. This assumes the constant is –256 to +255 and the shift is 0.
[Xn, #const]	Xn + *const* (scaled)	For the scaled-indirect-plus-offset mode (where the scaling factor is determined by the size of the data being loaded or stored), the constant has to be multiplied by the size of the memory operand prior to adding with the Xn register. For strb/ldrb, the multiplier is 1; for strh/ldrh, the multiplier is 2; for str/ldr (word register), the multiplier is 4; and for str/ldr (dword register), the multiplier is 8. For strb/ldrb, the constant must be in the range 0–4,096. For strh/ldrh, the constant must be in the range 0–8,191 and must be an even number. For ldr/str with a word-sized register operand, the constant must be in the range 0–16,383 and must be a multiple of 4. For ldr/str with a dword-sized register operand, the constant must be in the range 0–32,767 and must be a multiple of 8.
[Xn, #const]!	Xn + *const*	For the pre-indexed addressing mode, the effective address is the sum of the Xn register and the constant.
[Xn], #const	Xn	For the post-indexed addressing mode, the effective address is just the value in the Xn register.
[Xn, Xm]	Xn + Xm	For the scaled-indexed addressing mode, with a scaling factor of 1, the effective address is the sum of the two registers (sign- or zero-extend Xm, if specified).
[Xn, Xm, extend #s]	Xn + (Xm << *s*)	For the scaled-indexed addressing mode with a shift extension, the effective address is the sum of Xn plus the value in Xm shifted to the left *s* positions (with Xm zero- or sign-extended, if specified).

Suppose that a procedure has a local variable and you want to pass that on to a second procedure by reference. Because you access local variables by using the [FP, #*offset*] addressing mode, the effective address is FP + *offset*. You would have to use the following instruction to compute the address of that variable (leaving the address in X0):

```
add x0, fp, #offset
```

Listing 5-8 demonstrates passing a local variable as a reference parameter to a procedure.

```
// Listing5-8.S
//
// Demonstrate passing a local variable
// by reference to another procedure.

#include "aoaa.inc"

            .data
staticVar:  .word   .-.

            .code
            .extern aSecondFunction

            proc    demoPassLclByRef

            locals  ga
            word    ga.aLocalVariable
            endl    ga

            enter   ga.size
            add     x0, fp, #ga.aLocalVariable // Pass parameter in X0.
            bl      aSecondFunction

            leave
            endp    demoPassLclByRef
```

Pass by reference is usually less efficient than pass by value. You must dereference all pass-by-reference parameters on each access; this is slower than simply using a value because it typically requires at least two instructions: one to fetch the address into a register and one to fetch the value indirectly through that register.

However, when passing a large data structure, pass by reference is faster because you do not have to copy the large data structure before calling the procedure. Of course, you'd probably need to access elements of that large data structure (such as an array) by using a pointer, so little efficiency is lost when you pass large arrays by reference.

5.6.3 Using Low-Level Parameter Implementation

A parameter-passing mechanism is a contract between the caller and the callee (the procedure). Both parties have to agree on where the parameter data will appear and what form it will take (for example, value or address).

If your assembly language procedures are being called only by other assembly language code that you've written, you control both sides of the contract negotiation and get to decide where and how you're going to pass parameters. However, if external code is calling your procedure, or your procedure is calling external code, your procedure will have to adhere to whatever calling convention that external code uses.

Before discussing the particular calling conventions, this section considers the situation of calling code that you've written (and, therefore,

have complete control over its calling conventions). The following sections describe the various ways you can pass parameters in pure assembly language code (without the overhead associated with the ARM or macOS ABIs).

5.6.3.1 Passing Parameters in Registers

Having touched on *how* to pass parameters to a procedure, the next topic to discuss is *where* to pass parameters. This depends on the size and number of those parameters. If you are passing a small number of parameters to a procedure, the registers are an excellent place to pass them. If you are passing a single parameter to a procedure, pass that data in X0, as described in Table 5-2.

Table 5-2: Parameter Size and Location

Parameter size	Location
Byte	Pass a byte parameter in the LO byte of W0.
Half word	Pass a halfword parameter in the LO half-word of W0.
Word	Pass a word in W0.
Dword	Pass a dword in X0.
> 8 bytes	I suggest passing a pointer to the data structure in X0, or the value in X0/X1 if 16 bytes or fewer.

When passing fewer than 32 bits in X0, the macOS ABI requires that the value be zero- or sign-extended throughout the X0 register. The ARM ABI does not require this. Of course, when passing data to a procedure you've written in assembly language, it is up to you to define what must be done with the HO bits. The safest course of action, portable everywhere, is to zero-extend or sign-extend the value into the HO bits (depending on whether the value is unsigned or signed).

If you need to pass more than 8 bytes as a parameter, you could also pass that data in multiple registers (for example, under macOS and Linux, the C/C++ compiler will pass a 16-byte structure in two registers). Whether you pass the argument as a pointer or in multiple registers is up to you.

For passing parameters to a procedure in registers, the ARM ABI reserves X0 to X7. Of course, in pure assembly language code (that won't call, or be called by, ARM ABI–compliant code), you can use whichever registers you choose. However, X0 through X7 should probably be your first choice unless you can provide a good reason for using other registers.

Eight parameters probably cover 95 percent of the procedures ever written. If you are passing more than eight parameters to a pure assembly procedure, nothing is stopping you from using additional registers (for example, X8 through X15). Likewise, nothing is stopping you from passing large objects in multiple registers, if you really want to do that.

5.6.3.2 Passing Parameters in the Code Stream

You can also pass parameters in the code stream immediately after the `bl` instruction. Consider the following `print` routine that prints a literal string constant to the standard output device:

```
bl      print
.asciz "This parameter is in the code stream..."
```

Normally, a subroutine returns control to the first instruction immediately following the `bl` instruction. Were that to happen here, the ARM would attempt to interpret the ASCII codes for `"This..."` as an instruction. This would produce undesirable results. Fortunately, you can skip over this string before returning from the subroutine.

One big issue arises with the design of the ARM CPU, however: all instructions must be word-aligned in memory. Therefore, the parameter data appearing in the code stream must be a multiple of 4 bytes long (I chose the string in this example to contain 39 characters so that the zero-terminating byte made the whole sequence 40 bytes).

So how do you gain access to these parameters? Easy: the return address in LR points at them. Consider the implementation of `print` in Listing 5-9.

```
// Listing5-9.S
//
// Demonstrate passing parameters in the code stream

#include "aoaa.inc"

            .text
            .pool
ttlStr:     .asciz      "Listing 5-9"
            .align      2

// getTitle
//
// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// print
//
// Here's the print procedure.
// It expects a zero-terminated string
// to follow the call to print:

rtnAdrs     =       8           // Offset to rtn adrs from FP.

            proc    print
```

```
          ❶ locals  print
            qword   print.x0X1Save   // Register save area.
            qword   print.x2X3Save
            qword   print.x4X5Save
            qword   print.x6X7Save
            qword   print.x8X9Save
            qword   print.x10X11Save
            qword   print.x12X13Save
            qword   print.x14X15Save
            endl    print

            enter   print.size

// Assembly language convention--save all the registers
// whose values we change. Spares caller from having to
// preserve volatile registers.
// Note: this code calls ABI function write, so you must
// preserve all the volatile registers.

            stp     x0, x1, [fp, #print.x0X1Save]
            stp     x2, x3, [fp, #print.x2X3Save]
            stp     x4, x5, [fp, #print.x4X5Save]
            stp     x6, x7, [fp, #print.x6X7Save]
            stp     x8, x9, [fp, #print.x8X9Save]
            stp     x10, x11, [fp, #print.x10X11Save]
            stp     x12, x13, [fp, #print.x12X13Save]
            stp     x14, x15, [fp, #print.x14X15Save]

// Compute the length of the string immediately following
// the call to this procedure:

          ❷ mov     x1, lr              // Get pointer to string.
search4_0:  ldrb    w2, [x1], #1        // Get next char.
            cmp     w2,  #0             // At end of string?
            bne     search4_0           // If not, keep searching.
            sub     x2, x1, lr          // Compute string length.

// LR now points just beyond the 0 byte. We need to
// make sure this address is 4-byte aligned:

          ❸ add     x1, x1, #3
            and     x1, x1, #-4         // 0xfff...fff0

// X1 points just beyond the 0 byte and padding.
// Save it as the new return address:

          ❹ str     x1, [fp, #rtnAdrs]

// Call write to print the string to the console.
//
// write( fd, bufAdrs, len );
//
// fd in X0 (this will be 1 for stdout)
// bufAdrs in X1
```

```
// len in X2

          ❺ mov     x0, #1          // stdout = 1
            mov     x1, lr          // Pointer to string
            bl      write

// Restore the registers we used:

          ❻ ldp     x0, x1, [fp, #print.x0X1Save]
            ldp     x2, x3, [fp, #print.x2X3Save]
            ldp     x4, x5, [fp, #print.x4X5Save]
            ldp     x6, x7, [fp, #print.x6X7Save]
            ldp     x8, x9, [fp, #print.x8X9Save]
            ldp     x10, x11, [fp, #print.x10X11Save]
            ldp     x12, x13, [fp, #print.x12X13Save]
            ldp     x14, x15, [fp, #print.x14X15Save]
            leave                   // Return to caller.
            endp    print

// Here is the asmMain function:

            proc    asmMain, public
            enter   64

// Demonstrate passing parameters in code stream
// by calling the print procedure:

            bl      print
          ❼ .asciz  "Hello, world!!\n"

            leave   // Returns to caller
            endp    asmMain
```

The `print` procedure ❶ saves all the registers it modifies (even the volatile registers, because the call to `write()` might overwrite them). This is a normal assembly language convention, but it's especially important for `print` because you want to be able to print (debug) messages without saving register values across your calls.

LR points at the string to print upon entry into the `print` procedure ❷. This code scans through that string to find the zero-terminating byte; this scan produces both the length and the (approximate) return address.

Because code must be aligned on a 4-byte boundary, the return address isn't necessarily the byte after the zero-terminating byte. Instead, the code may need to pad the end of string pointer by 1 to 3 bytes to advance to the next word boundary in the `.text` section ❸. Adding `3` and then ANDing the result with 0xFFFFFFFFFFFFFFFC (`-4`) pads the return address up to the appropriate boundary. The code then stores the return address over the original on the stack ❹.

Once you have the string length, you can call the C stdlib `write` function to print it ❺ (if the first argument is `0`, this prints the string to the standard output device). On exit, the code restores the registers you saved earlier ❻.

For this listing, I included two exclamation marks ❼ so that the length of the string (including the zero-terminating byte) is a multiple of four characters. This ensures that the following instruction is aligned on a 4-byte boundary.

To avoid a bus fault, the length of the data following the call to `print` must be a multiple of 4 bytes so that the next instruction is properly aligned on a 4-byte boundary. The length of the string itself doesn't have to be a multiple of 4 bytes; arbitrary padding after the zero-terminating byte is fine. Rather than counting the number of characters in the string, you could use the Gas `.p2align` directive. This directive will pad the location counter to a boundary that is a multiple of 2^n bytes, where n is the (first) value in the `.p2align` operand field. For example

```
.p2align 2
```

pads the location counter to the next word boundary.

Using the `.p2align 2` directive, you can call the print procedure with an arbitrary-length string as follows:

```
bl        print
.asciz    "Hello, world!\n"
.p2align  2
```

Remembering to put the `.p2align 2` directive in the code can be difficult, not to mention that having to type it is a pain, and it clutters up your code. To resolve this, the *aoaa.inc* include file includes a `wastr` (word-aligned string) macro that automatically adds the padding for you:

```
bl       print
wastr    "Hello, world!\n"
```

Besides showing how to pass parameters in the code stream, the `print` routine also exhibits another concept: *variable-length parameters* (the length of the string can be arbitrarily long). The string following the `bl` can be any practical length. The zero-terminating byte marks the end of the parameter list. You can handle variable-length parameters in two easy ways: either use a special terminating value (like 0) or pass a special length value that tells the subroutine the number of parameters you are passing. Both methods have advantages and disadvantages.

Using a special value to terminate a parameter list requires that you choose a value that never appears in the list. For example, `print` uses `0` as the terminating value, so it cannot print the NUL character (whose ASCII code is 0). Sometimes this isn't a limitation. Specifying a length parameter is another mechanism you can use to pass a variable-length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare; this is especially true if the parameter list changes frequently.

Despite the convenience afforded by passing parameters in the code stream, this method also has disadvantages. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get confused. Consider the `print` example. It prints a string of characters up to a zero-terminating byte and then returns control to the first instruction following that byte. If you leave off the zero-terminating byte, the `print` routine happily prints the following opcode bytes as ASCII characters until it finds a 0 byte. Because 0 bytes often appear in the middle of an instruction, the `print` routine might return control into the middle of another instruction, which will probably crash the machine.

On the ARM, you must ensure that the parameters you pass in the code stream are a multiple of 4 bytes long. The instructions following the parameters must lie on a word boundary. Problems notwithstanding, however, the code stream is an efficient place to pass parameters whose values do not change.

5.6.3.3 Passing Parameters on the Stack

Most HLLs use the stack to pass a large number of parameters because this method is fairly efficient. Although passing parameters on the stack is slightly less efficient than doing so in registers, the register set is limited (especially if you're limiting yourself to the eight registers the ARM ABI sets aside for this purpose). The stack, on the other hand, allows you to pass a large amount of parameter data without difficulty. This is the reason that most programs pass their parameters on the stack (at least, when passing more than eight parameters).

To manually pass parameters on the stack, push them immediately before calling the subroutine (just remember to keep the stack 16-byte aligned). The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following HLL function call:

```
CallProc(i,j,k);
```

Because keeping SP aligned on a 16-byte boundary is crucial, you can't simply push one argument at a time with a `str` instruction, nor can you push values smaller than 32 bits. Assuming that `i`, `j`, and `k` are 32-bit integers, you would need to somehow marshal them together into a 128-bit package (including an extra 32 bits of unused data) and push 16 bytes onto the stack. This is so inconvenient that ARM code almost never pushes individual (or even pairs of) register values onto the stack.

The common solution in ARM assembly language is first to drop the stack down by however many bytes you need (plus any padding, to make sure the stack is aligned properly), and then to simply store your parameters into the stack space so allocated. For example, to call `CallProc`, you might use code like the following:

```
sub sp, sp, #16   // Allocate space for parameters.
str w0, [sp]      // Assume i is in W0,
```

```
str w1, [sp, #4]  // j is in W1, and
str w2, [sp, #8]  // k is in W2.
bl  CallProc
add sp, sp, #16   // Caller must clean up stack.
```

The `sub` instruction allocates 16 bytes on the stack; you need only 12 for the three 32-bit parameters, but you must allocate 16 to keep the stack aligned.

The three `str` instructions store the parameter data (which is presumed to be in W0, W1, and W2 by this code) into the 12 bytes from SP + 0 through SP + 11. The `CallProc` will simply ignore the extra 4 bytes allocated on the stack.

In this example, the three 32-bit integers are packed into memory, each consuming 4 bytes on the stack. So the `i` parameter is found at SP + 0, the `j` parameter is found at SP + 4, and the `k` parameter is found at SP + 8 upon entry into `CallProc` (see Figure 5-6).

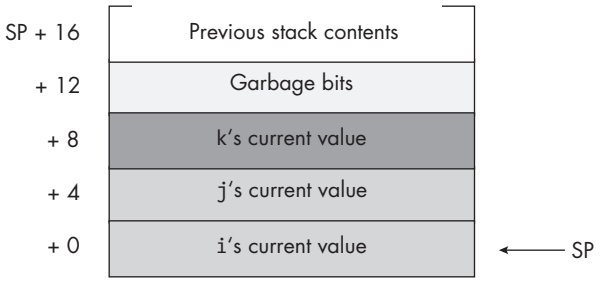


Figure 5-6: Stack layout upon entry into `CallProc`

If your procedure includes the standard entry and exit sequences, you may directly access the parameter values in the activation record by indexing off the FP register. Consider the layout of the activation record for `CallProc` that uses the following declaration:

```
proc  CallProc
enter 0         // No local variables
  .
  .
  .
leave
endp  CallProc
```

At this point, `i`'s value can be found at `[FP, #16]`, `j`'s value can be found at `[FP, #20]`, and `k`'s value can be found at `[FP, #24]` (see Figure 5-7).

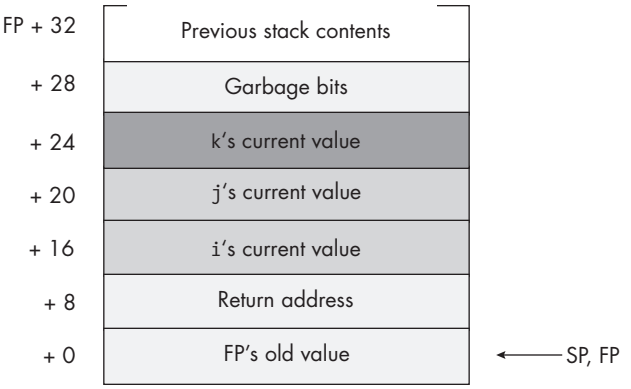


Figure 5-7: `CallProc` activation record after standard entry sequence

Within the `CallProc` procedure, you can access the parameter values with these instructions:

```
ldr w0, [fp, #16]
ldr w1, [fp, #20]
ldr w2, [fp, #24]
```

Of course, using magic numbers such as these to reference the parameter offsets is still a bad idea. It would be far better to use equates or, even better, create a declaration macro similar to `struct` and `locals` to define the parameters for a procedure. The *aoaa.inc* file contains just such a macro: `args` (and `enda`). Listing 5-10 demonstrates the use of this macro.

```
// Listing5-10.S
//
// Accessing a parameter on the stack

#include "aoaa.inc"

            .data
value1:     .word   20
value2:     .word   30
pVar:       .word   .-.

ttlStr:     .asciz  "Listing 5-10"
fmtStr1:    .asciz  "Value of parameter: %d\n"

            .code
            .extern printf

// getTitle
//
// Return program title to C++ program.

            proc    getTitle, public
            lea     x0, ttlStr
```

```
            ret
            endp    getTitle

// valueParm
//
// Passed a single parameter (vp.theParm) by value

            proc    valueParm

            args    vp          // Declare the
            word    vp.theParm  // parameter.
            enda    vp

            enter   64          // Alloc space for printf.

// vparms macro accepts only global variables.
// Must copy parameter to that global to print it:

            lea     x0, fmtStr1
            ldr     w1, [fp, #vp.theParm]
            lea     x2, pVar
            str     w1, [x2]
            vparm2  pVar
            bl      printf

            leave
            ret
            endp    valueParm

// Here is the asmMain function:

            proc    asmMain, public
            enter   64

            lea     x0, value1
            ldr     w1, [x0]
            str     w1, [sp]        // Store parameter on stack.
            bl      valueParm

            lea     x0, value2
            ldr     w1, [x0]
            str     w1, [sp]        // Store parameter on stack.
            bl      valueParm

            leave
            endp    asmMain
```

The `args` macro requires an argument list name, which can be the procedure name or an abbreviation of it, and an optional second argument with a starting offset. The second argument defaults to 16, which is an appropriate value if the procedure uses the standard entry sequence (pushing the LR and FP registers on the stack). Offsets associated with the parameters you declare are offsets from FP in the procedure.

Here's the `build` command and sample output for Listing 5-10:

```
% ./build Listing5-10
% ./Listing5-10
Calling Listing5-10:
Value of parameter: 20
Value of parameter: 30
Listing5-10 terminated
```

If your procedure does not use the standard entry sequence, you can specify an explicit offset as the second argument. For example:

```
args procName, 0
```

If you aren't pushing anything on the stack in the procedure (or allocating local variables), 0 is a good value to use; then the offsets are SP based rather than FP based.

5.6.3.4 Removing Parameters in Callee vs. Caller Stack Cleanup

When passing parameters on the stack, ultimately those parameters must be removed from the stack. The ARM ABI specifies that the caller is responsible for removing all parameters it pushes onto the stack. Most of the example programs in this book thus far have (implicitly) done this.

Removing the parameters after every procedure call is slow and inefficient. Fortunately, an easy optimization eliminates the need to allocate and deallocate parameter storage for each function call. Upon entry into a procedure, when allocating storage for local variables, include additional storage to be used for parameters the procedure passes to other functions. This, in fact, has been the whole purpose of the "magic stack allocation" instructions at the beginning of most procedures in this book up to this point. The examples thus far have typically reserved 64 or 256 bytes of storage on the stack (enough for between eight and thirty-two 64-bit parameters, respectively).

Functions that pass parameters on the stack, such as `printf()` running on macOS, can store data into this area prior to calling the function. Upon return from the function, your code does not have to worry about cleaning up the parameters. That stack space is now available for the next function you want to call that requires stack parameters.

Ultimately, of course, the parameters must be deallocated from the stack. That happens when the procedure executes the `leave` macro (or manually copies FP into SP, which is part of `leave`'s expansion). When using `enter` and `leave` to allocate this stack space for the parameters, along with any local variables a procedure might need, you need to allocate and deallocate the stack space only once, not for each individual procedure call.

If your procedure doesn't have any local variables, you can easily allocate stack space for parameters by using code like the following:

```
proc  myProc
enter 64    // Allocate 64 bytes for parameter usage.
  .
  .
  .
leave       // Deallocate storage and returns.
endp  myProc
```

If your procedure requires local variable storage, just specify the extra stack space as a dummy local variable at the end of your `locals` declaration:

```
proc   myProc

locals mp
word   mp.local1
dword  mp.local2
byte   mp.local3
byte   mp.stack, 64 // Allocate 64 bytes for parms.
endl   mp

enter  mp.size      // Allocate locals and stack space.
  .
  .
  .
leave               // Deallocate storage and returns.
endp   myProc
```

Remember that `enter` always allocates a multiple of 16 bytes, so we know that the stack storage will be aligned on a 16-byte boundary.

5.6.3.5 Passing Parameters to the C/C++ printf() Function

Under Linux, you pass the first eight `printf()` parameters in registers, just as you would any other nonvariadic function. On macOS, those parameters are always passed on the stack, each occupying a dword. Until now, this book has used the `vparms`*n* macros to handle the difference in the way parameters are passed (and, of course, to avoid dealing with the stack, which the book hadn't covered in the earlier chapters).

In this book, I strived to write code that is portable between Linux and macOS, resorting to OS-specific code only as necessary; this was part of the motivation for using the `vparms`*n* macros when calling `printf()`. Now that you've learned how these two OSes expect you to pass variadic parameters, you'll probably want to pass parameters in a more flexible manner than using the `vparms`*n* macros. Nevertheless, there is great benefit (at least for the source code in this book) to writing portable code. Fortunately, with a little sleight of hand, it is possible to directly pass the parameters to `printf()` without using `vparms`*n* and still have the code assemble and run on both OSes.

The first rule is to load each `printf()` argument into X0 through X7. This puts the arguments into the locations where Linux expects them. Once the arguments are in these registers, you'll also store them into the

stack storage area at SP + 0, SP + 8, SP + 16, . . . , SP + 56 (which is where macOS expects them). Here's a typical call to `printf()` printing the values in X0, X5, and X7:

```
locals mp
Byte   mp.stack, 24
endl   mp
 .
 .
 .
enter mp.size
 .
 .
 .
mov  x1, x0         // Put data in appropriate registers first.
mov  x2, x5
mov  x3, x7
lea  x0, fmtStr
str  x1, [sp]       // For macOS, store the arguments
str  x2, [sp, #8]   // onto the stack in their
str  x3, [sp, #16]  // appropriate locations.
bl   printf         // Then call printf.
```

Strictly speaking, the `str` instructions aren't necessary when running under Linux. To allow the creation of slightly more efficient code, I've provided the following `mstr` macro in the *aoaa.inc* include file:

```
mstr register, memory
```

This macro assembles to nothing under Linux and to the corresponding `str` instruction under macOS. If you rewrite the former code by using `mstr`, it will not generate any excess code under Linux:

```
locals mp
Byte   mp.stack, 24
endl   mp
 .
 .
 .
enter mp.size
 .
 .
 .
mov  x1, x0         // Put data in appropriate registers first.
mov  x2, x5
mov  x3, x7
lea  x0, fmtStr
mstr x1, [sp]       // For macOS, store the arguments
mstr x2, [sp, #8]   // onto the stack in their
mstr x3, [sp, #16]  // appropriate locations.
bl   printf         // Then call printf.
```

Of course, if you're writing code only for Linux and don't care at all about macOS portability, you can drop the `mstr` instructions altogether to remove some clutter.

5.6.4 Accessing Reference Parameters on the Stack

Because you pass the addresses of objects as reference parameters, accessing the reference parameters within a procedure is slightly more difficult than accessing value parameters, as you must dereference the pointers to the reference parameters.

Consider Listing 5-11, which demonstrates a single pass-by-reference parameter.

```
// Listing5-11.S
//
// Accessing a reference parameter on the stack

#include "aoaa.inc"

            .data
value1:     .word   20
value2:     .word   30

ttlStr:     .asciz  "Listing 5-11"
fmtStr1:    .asciz  "Value of reference parameter: %d\n"

            .code
            .extern printf

// getTitle
//
// Return program title to C++ program.

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// refParm
//
// Expects a pass-by-reference parameter on the stack

            proc    refParm

            args    rp
            dword   rp.theParm
            enda    rp

            enter   64              // Alloc space for printf.

            lea     x0, fmtStr1
          ❶ ldr     x1, [fp, #rp.theParm]
            ldr     w1, [x1]
```

```
          ❷ mstr    x1, [sp]
            bl      printf

            leave
            endp    refParm

// Here is the asmMain function:

            proc    asmMain, public
            enter   64

// Pass the address of the arguments on the
// stack to the refParm procedure:

          ❸ lea     x0, value1
            str     x0, [sp]        // Store address on stack.
            bl      refParm

            lea     x0, value2
            str     x0, [sp]        // Store address on stack.
            bl      refParm

            leave

            endp    asmMain
```

The `refParm` procedure fetches the reference parameter (a 64-bit pointer) into X1 ❶ and then immediately dereferences this pointer by fetching the 32-bit word at the address in X1. The `mstr` macro ❷ stores the second parameter onto the stack (under macOS). To pass a variable by reference to `refParm` ❸, you must compute its effective address and pass that.

Here is the `build` command and sample output for the program in Listing 5-11:

```
$ ./build Listing5-11
$ ./Listing5-11
Calling Listing5-11:
Value of reference parameter: 20
Value of reference parameter: 30
Listing5-11 terminated
```

As you can see, accessing (small) pass-by-reference parameters is a little less efficient than accessing value parameters, because you need an extra instruction to load the address into a 64-bit pointer register (not to mention that you have to reserve a 64-bit register for this purpose). If you access reference parameters frequently, these extra instructions can really begin to add up, reducing the efficiency of your program.

Furthermore, it's easy to forget to dereference a reference parameter and use the address of the value in your calculations. Therefore, unless you really need to affect the value of the actual parameter, you should use pass by value to pass small objects to a procedure.

Passing large objects, like arrays and records, is where using reference parameters becomes efficient. When passing these objects by value, the calling code has to make a copy of the actual parameter; if it is a large object, the copy process can be inefficient. Because computing the address of a large object is just as efficient as computing the address of a small scalar object, no efficiency is lost when passing large objects by reference. Within the procedure, you must still dereference the pointer to access the object, but the efficiency loss due to indirection is minimal when you contrast this with the cost of copying that large object.

Listing 5-12 demonstrates how to use pass by reference to initialize an array of structures.

```
// Listing5-12.S
//
// Passing a large object by reference

#include "aoaa.inc"

NumElements =       24

// Here's the structure type:

            struct  Pt
            byte    pt.x
            byte    pt.y
            ends    Pt

            .data

ttlStr:     .asciz  "Listing 5-12"
fmtStr1:    .asciz  "refArrayParm[%d].x=%d"
fmtStr2:    .asciz  "refArrayParm[%d].y=%d\n"

            .code
            .extern printf

// getTitle
//
// Return program title to C++ program.

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// refAryParm
//
// Passed the address of an array of Pt structures
// Initializes each element of that array

            proc    refAryParm
```

```
            args    rap
            dword   rap.ptArray     // Reference parameter
            enda    rap

            enter   0               // No stack space needed!

// Get the base address of the array into X1:

          ❶ ldr     x1, [fp, #rap.ptArray]

// While X0 < NumElements, initialize each
// array element. x = X0/8, y = X0 % 8:

            mov     x0, xzr             // Index into array.
ForEachEl:  cmp     x0, #NumElements    // While we're not done
            bhs     LoopDone

// Compute address of ptArray[X0].
// Element adrs = base address (X1) + index (X19) * size (2):

          ❷ add     x3, x1, x0, lsl #1  // X3 = X1 + X0 * 2

// Store index / 8 into x field:

            lsr     x2, x0, #3          // X2 = X0 / 8
            strb    w2, [x3, #pt.x]     // ptArray[X0].x = X0/8

// Store index % 8 (mod) into y field:

            and     x2, x0, #0b111      // X2 = X0 % 8
            strb    w2, [x3, #pt.y]     // ptArray[X0].y = X0 % 8

// Increment index and repeat:

            add     x0, x0, #1
            b.al    ForEachEl

LoopDone:   leave
            endp    refAryParm

// Here is the asmMain function:

            proc    asmMain, public

// Easier to access local variables than globals, so let's
// make everything a local variable:

            locals  am
            word    saveX19
            byte    Pts, NumElements * (Pt.size)
            byte    stackSpace, 64
            endl    am

            enter   am.size             // Reserve space.
```

```
            str     x19, [fp, #saveX19] // Save nonvolatile reg.

// Initialize the array of points:

          ❸ add     x0, fp, #Pts    // Compute address of Pts.
            str     x0, [sp]        // Pass address on stack.
            bl      refAryParm

// Display the array:

            mov     x19, xzr        // X19 is loop counter.
dispLp:     cmp     x19, #NumElements
            bhs     dispDone

// Print the x field:

            lea     x0, fmtStr1
            mov     x1, x19
            mstr    x1, [sp]
            add     x3, fp, #Pts         // Get array base address.
            add     x3, x3, x19, lsl #1  // Index into array.
            ldrb    w2, [x3, #pt.x]      // Get ptArray[X0].x.
            mstr    x2, [sp, #8]
            bl      printf

// Print the y field:

            lea     x0, fmtStr2
            mov     x1, x19
            mstr    x1, [sp]
            add     x3, fp, #Pts         // Get array base address.
            add     x3, x3, x19, lsl #1  // Index into array.
            ldrb    w2, [x3, #pt.y]      // Get ptArray[X0].x.
            mstr    x2, [sp, #8]
            bl      printf

// Increment index and repeat:

            add     x19, x19, #1
            b.al    dispLp

dispDone:
            ldr     x19, [fp, #saveX19]  // Restore X19.
            leave
            endp    asmMain
```

The code computes the address of the `Pts` array and passes this array (by reference) to the `refAryParm` procedure ❸. It loads this address into X1 ❶ and uses this pointer value as the base address of the array that `refAryParm` processes ❷.

Here's the `build` command and sample output:

```
$ ./build Listing5-12
$ ./Listing5-12
```

```
Calling Listing5-12:
refArrayParm[0].x=0 refArrayParm[0].y=0
refArrayParm[1].x=0 refArrayParm[1].y=1
refArrayParm[2].x=0 refArrayParm[2].y=2
refArrayParm[3].x=0 refArrayParm[3].y=3
refArrayParm[4].x=0 refArrayParm[4].y=4
refArrayParm[5].x=0 refArrayParm[5].y=5
refArrayParm[6].x=0 refArrayParm[6].y=6
refArrayParm[7].x=0 refArrayParm[7].y=7
refArrayParm[8].x=1 refArrayParm[8].y=0
refArrayParm[9].x=1 refArrayParm[9].y=1
refArrayParm[10].x=1 refArrayParm[10].y=2
refArrayParm[11].x=1 refArrayParm[11].y=3
refArrayParm[12].x=1 refArrayParm[12].y=4
refArrayParm[13].x=1 refArrayParm[13].y=5
refArrayParm[14].x=1 refArrayParm[14].y=6
refArrayParm[15].x=1 refArrayParm[15].y=7
refArrayParm[16].x=2 refArrayParm[16].y=0
refArrayParm[17].x=2 refArrayParm[17].y=1
refArrayParm[18].x=2 refArrayParm[18].y=2
refArrayParm[19].x=2 refArrayParm[19].y=3
refArrayParm[20].x=2 refArrayParm[20].y=4
refArrayParm[21].x=2 refArrayParm[21].y=5
refArrayParm[22].x=2 refArrayParm[22].y=6
rRefArrayParm[23].x=2 refArrayParm[23].y=7
Listing5-12 terminated
```

This output shows how the `refAryParm` procedure initialized the array.

5.7 Functions and Function Return Results

Functions are procedures that return a result to the caller. In assembly language, few syntactical differences exist between a procedure and a function. This is why *aoaa.inc* doesn't provide a specific macro declaration for a function. Nevertheless, semantic differences exist; although you can declare them the same way in Gas, you use them differently.

Procedures are a sequence of machine instructions that fulfill a task. The end result of the execution of a procedure is the accomplishment of that activity. Functions, on the other hand, execute a sequence of machine instructions specifically to compute a value to return to the caller. Of course, a function can perform an activity as well, and procedures can undoubtedly compute values, but the main difference is that the purpose of a function is to return a computed result; procedures don't have this requirement.

In assembly language, you don't specifically define a function by using special syntax. In Gas, everything is a procedure. A section of code becomes a function when the programmer explicitly decides to return a function result via the procedure's execution.

The registers are the most common place to return function results. The `strlen()` routine in the C stdlib is a good example of a function that returns a value in one of the CPU's registers. It returns the length of the string (whose address you pass as a parameter) in the X0 register.

By convention, programmers try to return 8-, 16-, and 32-bit results in the W0 register and 64-bit values in the X0 register. This is where most HLLs return these types of results, and it's where the ARM ABI states that you should return function results. The exception is floating-point values; I discuss floating-point function results in Chapter 6.

There is nothing particularly sacred about the W0/X0 register. You can return function results in any register if it's more convenient to do so. Of course, if you're calling an ARM ABI–compliant function, such as `strlen()`, you have no choice but to expect the function's return result in the X0 register. The `strlen()` function returns an integer in X0, for example.

If you need to return a function result that is larger than 64 bits, you obviously must return it somewhere other than in X0 (which can hold only 64-bit values). For values slightly larger than 64 bits (for example, 128 bits or maybe even as many as 256 bits), you can split the result into pieces and return those parts in two or more registers. It is not uncommon to see functions returning 128-bit values in the X1:X0 register pair. Just keep in mind that these schemes are not ARM ABI compliant, so they're practical only when calling code you've written.

If you need to return a large object as a function result (say, an array of 1,000 elements), you obviously are not going to be able to return the function result in the registers. When returning function results greater than 64 bits, the ARM ABI specifies that the caller allocate storage for the result and pass a pointer to that storage in X8. The function places the result in that storage, and the caller retrieves the data from that location upon return.

5.8 Recursion

Recursion occurs when a procedure calls itself. The following, for example, is a recursive procedure:

```
proc  Recursive
enter 0
bl    Recursive
leave
endp  Recursive
```

Of course, the CPU will never return from this procedure. Upon entry into `Recursive`, this procedure will immediately call itself again, and control will never pass to the end of the procedure. In this case, runaway recursion results in a logical infinite loop that produces stack overflow, at which point the OS will raise an exception and stop the program.

Like a looping structure, recursion requires a termination condition in order to stop infinite recursion. `Recursive` could be rewritten with a termination condition as follows:

```
          proc  Recursive
          enter 0
```

```
          subs  x0, x0, #1
          beq   allDone
          bl    Recursive
allDone:
          leave
          endp  Recursive
```

This modification to the routine causes `Recursive` to call itself the number of times appearing in the X0 register. On each call, `Recursive` decrements the X0 register by 1 and then calls itself again. Eventually, `Recursive` decrements X0 to 0 and returns from each call until it returns to the original caller.

So far in this section, there hasn't been a real need for recursion. After all, you could efficiently code this procedure as follows:

```
          proc  Recursive
          enter 0
iterLp:
          subs  x0, x0, #1
          bne   iterLp
          leave
          endp  Recursive
```

Both of these last two examples would repeat the body of the procedure the number of times passed in the X0 register. (The latter version will do it considerably faster because it doesn't have the overhead of the `bl`/`ret` instructions.) As it turns out, you cannot implement only a few recursive algorithms in an iterative fashion. However, many recursively implemented algorithms are more efficient than their iterative counterparts, and most of the time the recursive form of the algorithm is much easier to understand.

The *quicksort algorithm* is probably the most famous algorithm that usually appears in recursive form. Listing 5-13 shows a Gas implementation of this algorithm.

```
// Listing5-13.S
//
// Recursive quicksort

#include "aoaa.inc"

numElements =       10

            .data
ttlStr:     .asciz  "Listing 5-13"
fmtStr1:    .asciz  "Data before sorting: \n"
fmtStr2:    .ascii  "%d"   // Use nl and 0 from fmtStr3
fmtStr3:    .asciz  "\n"
fmtStr4:    .asciz  "Data after sorting: \n"
fmtStr5:    .asciz  "ary=%p, low=%d, high=%d\n"

theArray:   .word   1,10,2,9,3,8,4,7,5,6
```

```
            .code
            .extern printf

// getTitle
//
// Return program title to C++ program.

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// quicksort
//
// Sorts an array using the quicksort algorithm
//
// Here's the algorithm in C, so you can follow along:
//
// void quicksort(int a[], int low, int high)
// {
//     int i,j,Middle;
//     if( low < high )
//     {
//         Middle = a[(low + high)/2];
//         i = low;
//         j = high;
//         do
//         {
//             while( a[i] <= Middle ) i++;
//             while( a[j] > Middle ) j--;
//             if( i <= j )
//             {
//                 swap( a[i],a[j] );
//                 i++;
//                 j--;
//             }
//         } while( i <= j );
//
//         // Recursively sort the two subarrays:
//
//         if( low < j ) quicksort( a,low,j );
//         if( i < high ) quicksort( a,i,high );
//     }
// }
//
// Args:
//    X19 (_a):       Pointer to array to sort
//    X20 (_lowBnd):  Index to low bound of array to sort
//    X21 (_highBnd): Index to high bound of array to sort
//
// Within the procedure body, these registers
// have the following meanings:
//
// X19: Pointer to base address of array to sort
```

```
// X20: Lower bound of array (32-bit index)
// X21: Higher bound of array (32-bit index)
//
// X22: index (i) into array
// X23: index (j) into array
// X24: Middle element to compare against
//
// Create definitions for variable names as registers
// to make the code more readable:

#define array x19
#define lowBnd x20
#define highBnd x21
#define i x22
#define j x23
#define middle w24

            proc    quicksort

            locals  qsl
            dword   qsl.saveX19
            dword   qsl.saveX20
            dword   qsl.saveX21
            dword   qsl.saveX22
            dword   qsl.saveX23
            dword   qsl.saveX24
            dword   qsl.saveX0
            byte    qsl.stackSpace, 32
            endl    qsl

            enter   qsl.size

// Preserve the registers this code uses:

            str     x0, [fp, #qsl.saveX0]
            str     x19, [fp, #qsl.saveX19]
            str     x22, [fp, #qsl.saveX22]
            str     x23, [fp, #qsl.saveX23]
            str     x24, [fp, #qsl.saveX24]

            cmp     lowBnd, highBnd
            bge     endif3

            mov     i, lowBnd       // i = low
            mov     j, highBnd      // j = high

// Compute a pivotal element by selecting the
// physical middle element of the array:
//
// Element address = ((i + j) / 2) * 4 (4 is element size)
//                 = ((i + j) * 2)

            add     x0, i, j
            lsr     x0, x0, #1
```

```
// Middle = ary[(i + j) / 2]:

            ldr     middle, [array, x0, lsl #2]

// Repeat until the i and j indices cross each
// other (i works from the start toward the end
// of the array, j works from the end toward the
// start of the array):

rptUntil:

// Scan from the start of the array forward,
// looking for the first element greater or equal
// to the middle element:

            sub     i, i, #1        // To counteract add, below
while1:     add     i, i, #1        // i = i + 1
            ldr     w1, [array, i, lsl #2]
            cmp     middle, w1      // While middle <= ary[i]
            bgt     while1

// Scan from the end of the array backward, looking
// for the first element that is less than or equal
// to the middle element:

            add     j, j, #1     // To counteract sub, below
while2:     sub     j, j, #1     // j = j - 1
            ldr     w1, [array, j, lsl #2]
            cmp     middle, w1   // while middle >= a[j]
            blt     while2

// If you've stopped before the two pointers have
// passed over each other, you have two
// elements that are out of order with respect
// to the middle element, so swap these two elements:

            cmp     i, j        // If i <= j
            bgt     endif1

            ldr     w0, [array, i, lsl #2]
            ldr     w1, [array, j, lsl #2]
            str     w0, [array, j, lsl #2]
            str     w1, [array, i, lsl #2]

            add     i, i, #1
            sub     j, j, #1

endif1:     cmp     i, j        // Until i > j
            ble     rptUntil

// The code has just placed all elements in the array in
// their correct positions with respect to the middle
// element of the array. Unfortunately, the
// two halves of the array on either side of the pivotal
```

```
// element are not yet sorted. Call quicksort recursively
// to sort these two halves if they have more than one
// element in them (if they have zero or one elements,
// they are already sorted).

            cmp     lowBnd, j   // If lowBnd < j
            bge     endif2

            // Note: a is still in X19,
            // Low is still in X20.

            str     highBnd, [fp, #qsl.saveX21]
            mov     highBnd, j
            bl      quicksort   // ( a, low, j )
            ldr     highBnd, [fp, #qsl.saveX21]

endif2:     cmp     i, highBnd  // If i < high
            bge     endif3

            // Note: a is still in X19,
            // High is still in X21.

            str     lowBnd, [fp, #qsl.saveX20]
            mov     lowBnd, i
            bl      quicksort   // ( a, i + 1, high )
            ldr     lowBnd, [fp, #qsl.saveX20]

// Restore registers and leave:

endif3:
            ldr     x0,  [fp, #qsl.saveX0]
            ldr     x19, [fp, #qsl.saveX19]
            ldr     x22, [fp, #qsl.saveX22]
            ldr     x23, [fp, #qsl.saveX23]
            ldr     x24, [fp, #qsl.saveX24]
            leave
            endp    quicksort

// printArray
//
// Little utility to print the array elements

            proc    printArray

            locals  pa
            dword   pa.saveX19
            dword   pa.saveX20
            endl    pa

            enter   pa.size
            str     x19, [fp, #pa.saveX19]
            str     x20, [fp, #pa.saveX20]

            lea     x19, theArray
            mov     x20, xzr
```

```
whileLT10:  cmp     x20, #numElements
            bge     endwhile1

            lea     x0, fmtStr2
            ldr     w1, [x19, x20, lsl #2]
            mstr    w1, [sp]
            bl      printf

            add     x20, x20, #1
            b.al    whileLT10

endwhile1:  lea     x0, fmtStr3
            bl      printf

            ldr     x19, [fp, #pa.saveX19]
            ldr     x20, [fp, #pa.saveX20]
            leave
            endp    printArray

// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            dword   am.savex19
            dword   am.savex20
            dword   am.savex21
            byte    am.stackSpace, 64
            endl    am

            enter   am.size

            str     array, [fp, #am.saveX19]
            str     lowBnd, [fp, #am.saveX20]
            str     highBnd, [fp, #am.saveX21]

// Display unsorted array:

            lea     x0, fmtStr1
            bl      printf
            bl      printArray

// Sort the array:

            lea     array, theArray
            mov     lowBnd, xzr               // low = 0
            mov     highBnd, #numElements - 1 // high = 9
            bl      quicksort                 // (theArray, 0, 9)

// Display sorted results:

            lea     x0, fmtStr4
            bl      printf
            bl      printArray
```

```
        ldr     array, [fp, #am.saveX19]
        ldr     lowBnd, [fp, #am.saveX20]
        ldr     highBnd, [fp, #am.saveX21]
        leave
        endp    asmMain
```

Here's the `build` command and output for Listing 5-13:

```
$ ./build Listing5-13
$ ./Listing5-13
Calling Listing5-13:
Data before sorting:
1
10
2
9
3
8
4
7
5
6

Data after sorting:
1
2
3
4
5
6
7
8
9
10

Listing5-13 terminated
```

This output shows the contents of the array prior to sorting and after the quicksort procedure has sorted the array.

5.9 Procedure Pointers and Procedural Parameters

The ARM `bl` instruction supports an indirect form: `blr`. This instruction has the following syntax:

```
blr reg64  // Indirect call through reg64
```

This instruction fetches the address of a procedure's first instruction from this specified register. It is equivalent to the following pseudo-instructions:

```
add lr, pc, #4  // Set LR to return address (PC is pointing at mov).
mov pc, reg64   // Transfer control to specified procedure.
```

Gas treats procedure names like static objects. Therefore, you can compute the address of a procedure by using the `lea` macro along with the procedure's name. For example

```
lea x0, procName
```

loads the address of the very first instruction of the `procName` procedure into X0. The following code sequence winds up calling the `procName` procedure:

```
lea x0, procName
blr x0
```

Because the address of a procedure fits in a 64-bit object, you can store such an address into a double-word variable; in fact, you can initialize a double-word variable with the address of a procedure by using code like the following:

```
       proc  p
        .
        .
        .
       endp  p
        .
        .
        .
       .data
ptrToP:
       .dword  p
        .
        .
        .
       lea  x0, ptrToP
       ldr  x0, [x0]
       blr  x0    // Calls p if ptrToP has not changed
```

Note that although macOS does not allow you to initialize a dword variable in the `.text` section with the address of an object outside the `.text` section, it will allow you to initialize a pointer (in any section) with the address of some code within the `.text` section.

As with all pointer objects, you should not attempt to indirectly call a procedure through a pointer variable unless you've initialized that variable with an appropriate address. You can initialize a procedure pointer variable in two ways: you can create dword variables with initializers in the `.data`, `.text`, and `.rodata` sections, or you can compute the address of a routine (as a 64-bit value) and store that 64-bit address directly into the procedure pointer at runtime. The following code fragment demonstrates both ways to initialize a procedure pointer:

```
            .data
ProcPointer: .dword  p    // Initialize ProcPointer with
                          // the address of p.
```

```
                .
                .
                .
                lea  x0, ProcPointer
                ldr  x0, [x0]
                blr  x0  // First invocation calls p.

// Reload ProcPointer with the address of q:

                lea  x0, q
                lea  x1, ProcPointer
                str  x0, [x1]
                .
                .
                .
                lea  x0, ProcPointer
                ldr  x0, [x0]
                blr  x0  // This invocation calls q.
```

Although all the examples in this section use static variable declarations (`.data`, `.text`, `.bss`, and `.rodata`), you aren't limited to declaring simple procedure pointers in the static variable declaration sections. You can also declare procedure pointers (which are just dword variables) as local variables, pass them as parameters, or declare them as fields of a record or a union.

Procedure pointers are also invaluable in parameter lists. Selecting one of several procedures to call by passing the address of a procedure is a common operation. A *procedural parameter* is just a double-word parameter containing the address of a procedure, so passing a procedural parameter is really no different from using a local variable to hold a procedure pointer (except, of course, that the caller initializes the parameter with the address of the procedure to call indirectly).

5.10 A Program-Defined Stack

Using the pre- and post-indexed addressing modes, along with one of the ARM's 64-bit registers, it is possible to create software-controlled stacks that don't use the SP register. Since the ARM CPU provides a hardware stack pointer register, it may not be obvious why you'd consider using another stack. As you've learned, one limitation of the ARM's hardware stack is that it must be 16-byte aligned at all times. Return addresses and other values you might want to preserve on the stack are generally 8 bytes or smaller. For example, you cannot push the LR register onto the stack by itself without causing a bus error fault. However, if you create your own stack, you won't have this issue.

Perhaps you're wondering why anyone would ever want to use a second stack in their programs. If the normal hardware stack works fine, why add the complexity of a second stack? Having two stacks is useful in several situations. Particularly, coroutines, generators, and iterators can make use of an extra stack pointer. See section 5.12, "For More Information," on page 290 for a Wikipedia link on this subject. Of course, as just pointed out, not having to

16-byte align the stack pointer is another good reason for using a program-defined stack.

Creating your own stack has two drawbacks: you must dedicate one of the ARM's registers for this purpose, and you must explicitly allocate storage for that stack yourself (the OS automatically allocates the hardware stack when it runs your program).

You can easily allocate storage in the `.data` segment. A typical stack will have at least 128 to 256 bytes of storage. The following is a simple example that allocates a 256-byte stack:

```
            .data
smallStk:   .fill  256
endSmallStk:
```

You may need more than 256 bytes of storage if you use automatic variables in your procedures; see section 5.4.1, "Activation Records," on page 244 and section 5.5, "Local Variables," on page 250.

Normally, stacks start at the end of their allocated space in memory and grow downward toward smaller memory addresses. Having the `endSmallStk` label at the end of the stack in this example gives you a handle with which to initialize your stack pointer.

Because the ARM uses SP for its hardware stack pointer, you must use a different register for your program-defined stack pointer. This needs to be a nonvolatile register—you don't want a function call like `printf()` to mess with your stack. As X30 is already used for LR and X29 is reserved for FP (see Chapter 1), X28 is a good choice for a user-defined stack pointer (USP). You can initialize it to point at the end of `smallStk` as follows:

```
#define usp  x28   // Use a reasonable name for the user SP.
         .
         .
         .
        lea usp, endSmallStk
```

This leaves USP pointing just beyond the end of the stack, which is exactly what you want; the stack pointer should point at the current top of the stack, and when the stack is empty, as it is after initialization, the stack pointer isn't pointing at a valid stack address.

To push and pop data on the stack, use the same `str` and `ldr` instructions, along with the pre-indexed and post-indexed addressing modes, just as you would with the hardware stack. The only differences are that you specify the USP register (X28) and you don't have to keep the stack aligned to 16 bytes (in fact, you technically don't have to keep it aligned to anything, but it will be faster if you keep it word or dword aligned). Here's how you can push the LR register into the user stack and pop it off:

```
str lr, [usp, #-8]!  // Pre-decrement addressing mode
 .
 .
 .
```

```
ldr lr, [usp], #8    // Post-increment addressing mode
ret
```

Listing 5-14 is a rewrite of Listing 5-4 using a software stack.

```
// Listing5-14.S
//
// Demonstrating a software stack

#include "aoaa.inc"

#define usp x28     // Program-defined stack pointer

stackSpace  =           64  // Space on the HW stack
saveLRUSP   =           48  // 16 bytes to hold LR and USP

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 5-14"
space:      .asciz      " "
asterisk:   .asciz      "*, %ld\n"

            .data
loopIndex:  .dword       .-.     // Used to print loop index value

// Here's the software-based stack this program will use
// to store return addresses and the like:

            .align      3
smallStk:   .fill       256, .-.
endSmallStk:

            .code
            .extern     printf

// getTitle
//
// Return program title to C++ program.

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// print40Spaces
//
// Prints out a sequence of 40 spaces
// to the console display

            proc    print40Spaces
         ❶  stp     lr, x19, [usp, #-16]! // Preserve LR and X19.

            mov     w19, #40
printLoop:  lea     x0, space
            bl      printf
```

```
            subs    w19, w19, #1
            bne     printLoop // Until w19 == 0

          ❷ ldp     lr, x19, [usp], #16 // Restore LR and X19.
            ret
            endp    print40Spaces

// Here is the asmMain function:

            proc    asmMain, public
          ❸ sub     sp, sp, #stackSpace       // HW stack space
            stp     lr, usp, [sp, #saveLRUSP] // Save on HW stack.

          ❹ lea     usp, endSmallStk   // Initialize USP.
          ❺ str     x19, [usp, #-16]!  // Save X19 on SW stk.

            mov     x19, #20
astLp:      bl      print40Spaces
            lea     x0, loopIndex
            str     x19, [x0]
            lea     x0, asterisk
            vparm2  loopIndex
            bl      printf
            subs    x19, x19, #1
            bne     astLp

          ❻ ldr     x19, [usp], #16  // Restore from SW stack.
          ❼ ldp     lr, usp, [sp, #saveLRUSP]
            add     sp, sp, #stackSpace
            ret     // Returns to caller
            endp    asmMain
```

Upon entry into `print40Spaces`, the code pushes LR and X19 onto the software stack ❶, using an `stp` instruction to save both registers at the same time. The pre-indexed addressing mode decrements USP by 16; then this instruction stores the two 8-bit registers on the software stack. Just before returning, `print40Spaces` restores the LR and X19 registers from the software stack ❷, using an `lpd` instruction and the post-indexed addressing mode.

Although this program demonstrates using a software-controlled stack, it must still use the hardware stack for a couple of purposes. In particular, the `printf()` function will push its return address (and parameters, as it turns out) onto the hardware stack. Therefore, the main program sets up storage space on the hardware stack for this purpose ❸. The program must also preserve the USP register (X28) before initializing it to point at the end of the `smallStack` data area. The space just allocated on the hardware stack is the perfect place for this. As long as the code is saving USP there, it may as well save LR at the same time, since you must always write 16 bytes to the hardware stack.

Once the code has preserved USP's value (because it is a nonvolatile register), the next step is to initialize USP with the address of the end of the `smallStack` memory buffer ❹. Loading the address of `endSmallStk` into

USP accomplishes this. Once the stack is initialized, the code can use it; for example, this statement pushes nonvolatile register X19 onto the software stack ❺ (to preserve it for the C++ program).

Before leaving, the code pops the X19 nonvolatile register off the software stack to restore its value ❻. Finally, the main program restores USP and LR from the hardware stack (and cleans up allocated storage) before returning to the C++ code ❼.

Just to prove it really works, here's the `build` command and sample output for the program in Listing 5-14:

```
$ ./build Listing5-14
$ ./Listing5-14
Calling Listing5-14:
                                        *, 20
                                        *, 19
                                        *, 18
                                        *, 17
                                        *, 16
                                        *, 15
                                        *, 14
                                        *, 13
                                        *, 12
                                        *, 11
                                        *, 10
                                        *, 9
                                        *, 8
                                        *, 7
                                        *, 6
                                        *, 5
                                        *, 4
                                        *, 3
                                        *, 2
                                        *, 1
Listing5-14 terminated
```

As you can see, Listing 5-14 produces the same output as Listing 5-4.

5.11 Moving On

This chapter covered considerable material, including an introduction to assembly language programming style, basic Gas procedure syntax, local labels, calling and returning from procedures, register preservation, activation records, function results, and more. Armed with this information, you're ready to learn how to write functions that calculate arithmetic results in the next chapter.

5.12 For More Information

- For more details on the Creative Commons 4.0 Attribution license, see *https://creativecommons.org/licenses/by/4.0/*.

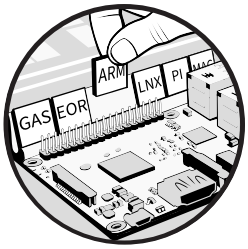
- The ARM developer site has more on the AARCH64 (ARM64) calling convention and ABI at *https://github.com/ARM-software/abi-aa/releases*.
- Wikipedia provides a useful entry on coroutines, generators, and iterators at *https://en.wikipedia.org/wiki/Coroutine*.

TEST YOURSELF

1. Explain, step by step, how the `bl` instruction works.
2. Explain, step by step, how the `ret` instruction works.
3. What is the main disadvantage of caller preservation?
4. What is the main problem with callee preservation?
5. What is an activation record?
6. What register usually points at an activation record, providing access to the data in that record?
7. What is the standard entry sequence for a procedure (the instructions)?
8. What is the standard exit sequence for a procedure (the instructions)?
9. What is an automatic variable?
10. When does the system allocate storage for an automatic variable?
11. What value does a pass-by-value parameter pass to a function?
12. What value does a pass-by-reference parameter pass to a function?
13. When passing four integer parameters to a function, where does the ARM ABI state those parameters are to be passed?
14. When passing more than eight parameters to a function, where does the ARM ABI state the parameters will be passed?
15. What is the difference between a volatile and nonvolatile register in the ARM ABI?
16. Which registers are volatile in the ARM ABI?
17. Which registers are nonvolatile in the ARM ABI?
18. When passing parameters in the code stream, how does a function access the parameter data?
19. What is the best way to pass a large array to a procedure?
20. Where is the most common place to return a function result?
21. What is a procedural parameter?
22. How would you call a procedure passed as a parameter to a function/procedure?
23. If a procedure has local variables, what is the best way to preserve registers within that procedure?

6

ARITHMETIC



This chapter discusses arithmetic computation in assembly language, including floating-point arithmetic on the ARM processor and architectural support for real arithmetic. By the end of this chapter, you should be able to translate arithmetic expressions and assignment statements from HLLs like Pascal, Swift, and C/C++ into ARM assembly language. You'll learn to pass floating-point values as parameters to procedures and return real values as function results.

6.1 Additional ARM Arithmetic Instructions

Before learning to encode arithmetic expressions in assembly language, you should learn the rest of the arithmetic instructions in the ARM instruction

set. Previous chapters have covered most of the arithmetic and logical instructions, so this section covers the remaining few.

6.1.1 Multiplication

Chapter 4 provided a brief introduction to multiplication with the `mul` and `madd` instructions. As a reminder, those instructions are as follows:

```
mul   Xd, Xs1, Xs2       // Xd = Xs1 * Xs2
madd  Xd, Xs1, Xs2, Xs3  // Xd = Xs1 * Xs2 + Xs3
```

As long as overflow doesn't occur, these instructions produce correct results for both unsigned and signed multiplications.

These instructions multiply two 64-bit integers and produce a 64-bit result. The multiplication of two n-bit numbers can actually produce a $2 \times n$–bit result, meaning that multiplying two 64-bit registers could produce up to a 128-bit result. These instructions ignore any overflow and keep only the LO 64 bits of the product (Chapter 8 discusses how to produce a full 128-bit result, if you require that).

You can also specify 32-bit registers for these two instructions:

```
mul   Wd, Ws1, Ws2       // Wd = Ws1 * Ws2
madd  Wd, Ws1, Ws2, Ws3  // Wd = Ws1 * Ws2 + Ws3
```

These instructions produce 32-bit results, ignoring any overflow. There are two additional multiplication instructions: multiply and subtract, and multiply and negate:

```
msub  Wd, Ws1, Ws2, Ws3   // Wd = Ws1 * Ws2 - Ws3
msub  Xd, Xs1, Xs2, Xs3   // Xd = Xs1 * Xs2 + Xs3
mneg  Wd, Ws1, Ws2        // Wd = -(Ws1 * Ws2)
mneg  Xd, Xs1, Xs2        // Xd = -(Xs1 * Xs2)
```

As with the previous instructions, these multiplications ignore any overflow beyond 32 or 64 bits.

The ARM does not provide multiplication instructions that affect the condition code flags. These instructions have no s-suffix versions.

6.1.2 Division and Modulo

The ARM64 CPU provides only two division instructions:

```
sdiv  Xd, Xs1, Xs2  // Xd = Xs1 / Xs2 (signed division)
udiv  Xd, Xs1, Xs2  // Xd = Xs1 / Xs2 (unsigned division)
```

Unlike with multiplication, you must use separate instructions for signed and unsigned integer values.

Division has two special cases that you must consider: division by 0 and dividing the smallest negative number by –1 (which would, mathematically, produce an overflow). A division by 0 produces 0 as the result, with no

indication of the problem. A signed division (sdiv) of 0x8000000000000000 (the smallest 64-bit negative number) by 0xFFFFFFFFFFFFFFFF (–1) will produce the result 0x8000000000000000, also without indication of an error. You'll get similar results for the 32-bit division: 0x80000000 / 0xFFFFFFFF. You must explicitly test for these operands before the division to catch these errors.

There's no single instruction to compute the remainder after a division operation on the ARM64 CPU. You can compute the remainder by combining a division and a multiplication operation:

```
mod( x0, x1 ) = x0 - (x0 / x1) * x1
```

Alternatively, you can compute the same result by using the following two instructions:

```
udiv x2, x0, x1
msub x3, x2, x1, x0
```

After this sequence, X2 and X3 hold the following values

```
x2 = x0 / x1
x3 = x0 % x1  // % is C modulo (remainder) operator.
```

thus providing the modulo in X3.

6.1.3 cmp Revisited

As noted in section 2.10.4, "cmp and Corresponding Conditional Branches," on page 78, the cmp instruction updates the ARM's flags according to the result of the subtraction operation (LeftOperand - RightOperand). Based on the way the ARM sets the flags, you can read this instruction as "compare LeftOperand to RightOperand." You can test the result of the comparison by using conditional branch instructions (see Chapter 2 for the conditional branches or Chapter 7 for more on control structure implementations).

A good place to start when exploring cmp is to look at exactly how it affects the flags. Consider the following cmp instruction:

```
cmp w0, w1
```

This instruction performs the computation W0 – W1 and sets the flags depending on the result of the computation. The flags are set as follows:

Z The zero flag is set if and only if W0 = W1. This is the only time W0 – W1 produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

N The negative (sign) flag is set to 1 if the result is negative. You might think this flag would be set if W0 is less than W1, but this isn't always the case. If W0 = 0x7FFFFFFFh and W1 = –1 (0xFFFFFFFF), then subtracting W1 from W0 produces 0x80000000, which is negative (so

the negative flag will be set). For signed comparisons, at least, the negative flag doesn't contain the proper status. For unsigned operands, consider W0 = 0xFFFFFFFF and W1 = 1. Here, W0 is greater than W1, but their difference is 0xFFFFFFFEh, which is still negative. As it turns out, the negative flag and the overflow flag, taken together, can be used for comparing two signed values.

V The overflow flag is set after a `cmp` operation if the difference of W0 and W1 produces a signed overflow or underflow. As mentioned previously, the sign and overflow flags are both used when performing signed comparisons.

C The carry flag is set after a `cmp` operation if subtracting W1 from W0 requires a borrow (unsigned overflow or underflow). This occurs only when W0 is less than W1, where W0 and W1 are both unsigned values.

Table 6-1 shows how the `cmp` instruction affects the flags after comparing to unsigned or signed values.

Table 6-1: Condition Code Settings After `cmp`

Flag	Unsigned result	Signed result
Zero (Z)	Equality/inequality	Equality/inequality
Carry (C)	Left ≥ right (C = 1) Left < right (C = 0)	No meaning
Overflow (V)	No meaning	See discussion in this section
Sign (N)	No meaning	See discussion in this section

Given that the `cmp` instruction sets the flags in this fashion, you can test the comparison of the two signed operands with the following flags:

```
cmp Left, Right
```

For signed comparisons, the N and V flags, taken together, have the following meanings:

- If [N != V], then *Left* < *Right* for a signed comparison.
- If [N == V], then *Left* ≥ *Right* for a signed comparison.

To understand why these flags are set in this manner, consider the 32-bit examples in Table 6-2. The values easily sign-extend to 64 bits, and the results are the same.

Table 6-2: Sign and Overflow Flag Settings After Subtraction (32-Bit Values)

Left	Minus	Right	N	V
0xFFFFFFFF (–1)	–	0xFFFFFFFE (–2)	0	0
0x80000000 (–2 billion+)	–	0x000000001	0	1
0xFFFFFFFE (–2)	–	0xFFFFFFFF (–1)	1	0
0x7FFFFFFF (2 billion+)	–	0xFFFFFFFF (–1)	1	1

Remember, the cmp operation is really a subtraction; therefore, the first example in Table 6-2 computes (–1) – (–2), which is +1. The result is positive, and an overflow did not occur, so both the N and V flags are 0. Because (N == V), *Left* is greater than or equal to *Right*.

The cmp instruction would compute (–2,147,483,648) – (+1), which is (–2,147,483,649), in the second example. Because a 32-bit signed integer cannot represent this value, the value wraps around to 0x7FFFFFFF (+2,147,483,647) and sets the overflow flag. The result is positive (at least as a 32-bit value), so the CPU clears the negative flag. Because (N == V) here, *Left* is less than *Right*.

In the third example, cmp computes (–2) – (–1), which produces (–1). No overflow occurred, so the V is 0; the result is negative, so N is 1. Because (N != V), *Left* is less than *Right*.

In the final example, cmp computes (+2,147,483,647) – (–1). This produces (+2,147,483,648), setting the overflow flag. Furthermore, the value wraps around to 0x80000000 (–2,147,483,648), so the negative flag is set as well. Because (N == V) is 0, *Left* is greater than or equal to *Right*.

The cmn (compare negative) instruction compares its first source operand against a negated second operand; like cmp, it sets the flags and ignores the result. It is also, like cmp, an alias for a different instruction, add:

```
add wzr, Ws1, Ws2
add xzr, Xs1, Xs2
```

This is because cmp is equivalent to a sub instruction, using WZR/XZR as the destination register; when comparing a negated value, you get the expression left – (–right), which is mathematically equivalent to left + right.

Using add as a synonym for cmn has one issue: add doesn't set the carry flag properly if the second (right) operand is 0. As a result, you cannot use the unsigned condition codes (hs, hi, ls, or lo) after a cmn instruction if there is any possibility that the right operand is 0. This shouldn't generally be a problem because, by definition, you are using cmn to compare signed values and you should be using signed conditionals after the use of the instruction.

Arguably the main reason for the existence of cmn is that Operand2 immediate values must be in the range 0 to 4,095. You cannot compare a register against a negative immediate value by using the cmp instruction. The cmn instruction is also limited to constants in the range 0 to 4,095, but it will negate the immediate value before the comparison, allowing negative constants in the range –1 to –4,095 (–0 is still 0).

6.1.4 Conditional Instructions

In the original, 32-bit ARM architecture, most of the data manipulation instructions were conditional. You could execute an instruction, such as add, conditionally, based on PSTATE condition code flag settings. Alas, the 4 bits required to test the 16 possible conditions (same as the conditional branch instructions) were needed for other encodings in 64-bit mode. Nevertheless, condition instruction execution is useful, so the ARM64 kept a few of the more commonly used condition instructions.

The first condition instruction is csel (conditional select)

```
csel Wd, Ws1, Ws2, cond  // if( cond ) then Wd = Ws1 else Wd = Ws2
csel Xd, Xs1, Xs2, cond  // if( cond ) then Xd = Xs1 else Xd = Xs2
```

where *cond* is one of the following condition specifications

```
cs, cc, eq, ne, mi, pl, vs, vc, hs, hi, ls, lo, gt, ge, lt, le
```

which have the same meanings as for the conditional branch instructions.

The *aoaa.inc* include file provides definitions for the following opposite conditions:

```
nhs, nhi, nls, nlo, ngt, nge, nlt, nle
```

These are synonyms for lo, ls, hi, hs, le, lt, ge, and gt, respectively.

As its name suggests, the csel instruction selects one of the two source operands to copy into the destination register, based on the current flag settings. For example, the following instruction

```
csel x0, x1, x2, eq
```

copies X1 into X0 if the zero flag is set; otherwise, it copies X2 into X0.

The csinc instruction allows for a conditional select (if true condition) or increment (if false condition) operation:

```
csinc Wd, Ws1, Ws2, cond  // if( cond ) then Wd = Ws1 else Wd = Ws2 + 1
csinc Xd, Xs1, Xs2, cond  // if( cond ) then Xd = Xs1 else Xd = Xs2 + 1
```

Using the predefined macro cinc is sometimes more convenient:

```
cinc Wd, Ws1, cond  // csinc  Wd, Ws1, Ws1, invert( cond )
cinc Xd, Xs1, cond  // csinc  Xd, Xs1, Xs1, invert( cond )
```

That is, cinc increments and copies the source into the destination if the condition is true; otherwise, it just copies the source without incrementing it. Of course, the source and destination registers can be the same if you simply want to conditionally increment a specific register. Note that the conditions for the cinc macros are reversed from the csinc instruction.

The next two conditional instructions are csinv and csneg, which conditionally invert or negate values:

```
csinv Wd, Ws1, Ws2, cond  // if( cond ) then Wd = Ws1 else Wd = not Ws2
csinv Xd, Xs1, Xs2, cond  // if( cond ) then Xd = Xs1 else Xd = not Xs2
csneg Wd, Ws1, Ws2, cond  // if( cond ) then Wd = Ws1 else Wd = -Ws2
csneg Xd, Xs1, Xs2, cond  // if( cond ) then Xd = Xs1 else Xd = -Xs2
```

There are also cinv and cneg macros that take only a single source operand (like cinc). The cset and csetm macros are variants of csinc and cinv:

```
cset  Wd, cond  // if( cond ) then Wd = 1 else Wd = 0
cset  Xd, cond  // if( cond ) then Xd = 1 else Xd = 0
csetm Wd, cond  // if( cond ) then Wd = -1 else Wd = 0
csetm Xd, cond  // if( cond ) then Xd = -1 else Xd = 0
```

The cset macro is equivalent to cinc with WZR or XZR as both source operands, and csetm is equivalent to cinv with WZR or XZR as the source operands. These macros are useful for setting a register to a Boolean value (either true/–1 or false/0) based on the condition codes.

Finally, the ARM also supports two conditional compare instructions, ccmp and ccmn (conditional compare negative), each with a few forms:

```
ccmp  Wd, Ws, #nzcv4, cond
ccmp  Xd, Xs, #nzcv4, cond
ccmp  Wd, #imm5, #nzcv4, cond
ccmp  Xd, #imm5, #nzcv4, cond
ccmn  Wd, Ws, #nzcv4, cond
ccmn  Xd, Xs, #nzcv4, cond
ccmn  Wd, #imm5, #nzcv4, cond
ccmp  Xd, #imm5, #nzcv4, cond
```

Whereas ccmp compares by subtracting the second operand from the first, ccmn compares by adding the second operand to the first. These instructions test the provided condition (*cond*). If it is false, these instructions copy the 4-bit immediate value *#nzcv4* directly into the condition codes (bit 3 to N, bit 2 to Z, bit 1 to C, and bit 0 to V).

If the condition specified by *cond* is true, these instructions compare the destination register to the source operand (register or 5-bit unsigned immediate value) and set the condition code bits based on the comparison. As you'll see later in this chapter, the conditional comparisons are useful for evaluating complex Boolean expressions.

6.2 Memory Variables vs. Registers

Before jumping into converting arithmetic expressions into assembly language statements, let's also wrap up the discussion of variables from the last five chapters. As I've pointed out many times, the ARM is based on a load/store architecture. The ARM has been blessed with many general-purpose registers that you can use in lieu of memory locations for your more commonly used variables. With careful planning, you should be able to keep most of your often-used variables in registers.

Consider the following C/C++ statement and its conversion to ARM assembly language:

```
x = y * z;

// Conversion to ARM assembly if x, y, and z are 32-bit
// memory variables in the .data section:

lea  x0, y     // Remember, lea expands to two instructions.
ldr  w0, [x0]
```

```
lea  x1, z
ldr  w1, [x1]
mul  w0, w0, w1
lea  x1, x
str  w0, [x1]
```

If you keep x, y, and z in registers W19, W20, and W21, respectively, the translation of that expression into assembly language would be

```
mul x19, x20, x21
```

which is one-tenth the size and much faster than the conversion just given.

On RISC CPUs like the ARM, it's a much better idea to keep variables in registers rather than in memory. Your job as an assembly language programmer is to carefully choose the variables you keep in registers versus the (less often used) values you will have to maintain in memory. You can do this by counting the number of times you access a variable during execution and keep the most-frequently accessed variables in registers, leaving the least-frequently accessed variables in memory.

6.2.1 Volatile vs. Nonvolatile Register Usage

If you are adhering to the ARM ABI in your assembly code, you must also be cognizant of the difference between volatile and nonvolatile registers in your procedures. Using nonvolatile registers has a cost: if you modify a nonvolatile register's value, you must preserve the register's original value within a procedure. This generally involves allocating storage in the procedure's activation record, storing the nonvolatile register's value on entry to the procedure, and restoring the register's value before returning.

Using volatile registers means you're spared the expense and storage required to preserve them. However, volatile registers may have their contents disturbed if you make calls to other procedures, which aren't known to explicitly preserve the volatile registers. Because it is the caller's responsibility to preserve any volatile register contents across other function calls, you may as well use a nonvolatile register (assuming one is available) if you're making calls to other functions within your procedures.

This assumes, of course, that the functions you're calling adhere to the ARM ABI conventions. If, for example, you're calling assembly language functions that preserve all register values they modify, you don't have to worry about preserving those registers, even if the ARM ABI considers them volatile.

6.2.2 Global vs. Local Variables

If you have to use memory—because you don't have sufficient register resources available or because you have a large data structure to manipulate that won't fit in registers—you can locate the variables you must

maintain in memory. You can put them in either a global, static data section (such as `.data`, `.bss`, and so on) or in an activation record you've created for your current procedure.

When you learned to program in an HLL, you were probably taught to avoid using global variables in your programs. That advice applies even more in ARM assembly language, especially when programming under macOS. Under macOS, as you've seen many times, accessing global data is more expensive than accessing local data in an activation record. To fetch a 32-bit variable from global (`.data`) memory requires code such as the following:

```
lea x0, globalVariable // Remember, this is two instructions.
ldr w0, [x0]
```

Fetching data from a local variable takes only a single instruction (assuming the variable's offset into the activation record is relatively small):

```
ldr w0, [fp, #localVariable]
```

That means accessing local variables takes one-third the number of instructions it takes to access global variables.

Of course, if you're running under Linux and don't need your assembly code to run under macOS as well, you can also access global variables by using a single instruction and the PC-relative addressing mode:

```
ldr w0, globalVariable
```

Just keep in mind that the data must sit within ±1MB of this instruction. Blowing past this limit is pretty easy when writing larger applications.

Local variables are not without their own limitations. In general, the activation record has a limit of about ±256 bytes of storage, a little more if you can use the scaled-indirect-plus-offset addressing mode with half-word, word, and double-word variables. Fortunately, you'll rarely surpass that number of bytes of scalar (non-array/nonstructure) variables in a single procedure. If you do require more space, you'll have to compute the effective address of the variable within the activation record, which winds up taking as many instructions as accessing global variables.

6.2.3 Easy Access to Global Variables

To make it just as easy to access global variables in a `.data` or `.bss` section as it is to access local variables within an activation record, you can create a static activation record. Local variables are easy to access because you use the indirect-plus-offset (or scaled indirect-plus-offset) addressing mode to index off the FP register. What if you had the equivalent of FP pointing into a static data section? Although the ARM doesn't provide an SB (static base) register, nothing is stopping you from creating your own:

```
#define SB X28
```

I chose to use X28 in this example, since it's a nonvolatile register in the ARM ABI and is right below the FP (X29) register.

Listing 6-1 demonstrates using the SB register (X28) to efficiently access global variables.

```
// Listing6-1.S
//
// Demonstrate using X28 as a "static base"
// register to conveniently access global
// variables.

        #include    "aoaa.inc"

#define sb X28          // Use X28 for SB register.

// Declaration of global variables:

        struct  globals_t
        word    g1
        dword   g2
        hword   g3
        byte    g4,128
        ends    globals_t

        .data

        globals_t globals       // Global variables go here.

        .text
        .pool
ttlStr: wastr  "Listing 6-1"

        proc    getTitle, public
        lea     x0, ttlStr
        ret
        endp    getTitle

        proc    asmMain, public

        locals  am
        dword   saveSB              // Save X28 here.
        byte    stackSpace, 64      // Generic stack space
        endl    am

        enter   am.size             // Reserve space for locals.
        str     sb, [fp, #saveSB]   // Preserve SB register.
        lea     sb, globals         // Initialize with address.

        mov     w0, #55             // Just demonstrate the
        str     w0, [sb, #g1]       // use of the static
        add     x0, x0, #44         // base record in the
        str     x0, [sb, #g2]       // .data section.
```

```
        and     w0, w0, #0xff
        strh    w0, [sb, #g3]

        ldr     sb, [fp, #saveSB]  // Restore SB register.
        leave                      // Return to caller.
        endp    asmMain
```

Keep in mind that the `[sb, #offset]` addressing mode is limited to ±256 bytes (or up to 1KB when using the scaled indirect-plus-offset modes), so it's best to keep nonscalar (composite) variables outside the static record.

As written, the `globals` record in Listing 6-1 provides access to only 256 bytes of storage (because all the `struct` field offsets are positive or 0). The following declaration starts the offsets at –256, providing an additional 256 bytes of storage in the static record:

```
struct  globals_t, -256
word    g1
dword   g2
hword   g3
byte    g4,128
ends    globals_t
```

However, if you do this, you must adjust the value you load into SB appropriately, as shown here

```
lea sb, globals+256  // Initialize with address.
```

so that SB will point into the correct place in the `globals_t` structure.

6.3 Arithmetic Expressions

The biggest shock to beginners facing assembly language for the first time will likely be the lack of familiar arithmetic expressions. Arithmetic expressions in most HLLs look similar to their algebraic equivalents. For example, in C you could write the following algebraic-like statement:

```
x = y * z;
```

In assembly language, you'll need several statements to accomplish this same task if these variables are sitting in memory locations (assume they're local variables):

```
ldr w0, [fp, #y]
ldr w1, [fp, #z]
mul w0, w0, w1
str w0, [fp, #x]

// If you can keep x, y, and z in registers:

mul x0, x1, x2  // Assume x = X0, y = X1, and z = X2.
```

Obviously, the HLL version is much easier to type, read, and understand. Although a lot of typing is involved, converting an arithmetic expression into assembly language isn't difficult. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break any arithmetic expression into an equivalent sequence of assembly language statements.

6.3.1 Simple Assignments

The easiest expressions to convert to assembly language are *simple assignments*, which copy a single value into a variable and take one of two forms:

```
variable = constant
```

or

```
var1 = var2
```

If your variables are sitting in registers, converting these statements to assembly language is simple:

```
mov variable, #constant  // Assumption: constant fits in 16 bits.
mov var1, var2
```

This `mov` instruction copies the source constant or register into the destination register.

If the constant is too large, you'll either have to use the `movk` sequence (see section 2.20.2, "`movk`," on page 112) or the constant form of `ldr`:

```
ldr register, =constant
```

If the source variable is in memory, you must use the `ldr` instruction to fetch the data from memory, as shown in the following examples:

```
ldr register, [fp, #offset]  // Assuming a local variable
ldr register, [sb, #offset]  // Assuming variable is in static record

lea reg64, GlobalVariable     // Global variable in arbitrary memory
ldr register, [reg64]
```

If the destination is a memory variable, you must first load the source variable or constant into a register (if it isn't already in a register) and use the `str` instruction to store the value into the memory variable:

```
str register, [fp, #offset]
str register, [sb, #offset]

lea reg64, GlobalVariable
str register, [reg64]
```

Clearly, the most efficient code occurs when both variables are in a register or the destination is a register and the source value is a small constant, in which case a single `mov` instruction suffices.

6.3.2 Simple Expressions

The next level of complexity is a *simple expression*, which takes the form

```
var1 = term1 op term2;
```

where *var1* is a variable, *term1* and *term2* are variables or constants, and *op* is an arithmetic operator (addition, subtraction, multiplication, and so on). Most expressions take this form. It should come as no surprise, then, that the ARM architecture was optimized for just this type of expression.

Assuming *var1*, *term1*, and *term2* are all in registers, a typical conversion for this type of expression takes the form

```
op  var1, term1, term2
```

where *op* is the mnemonic that corresponds to the specified operation (for example, + is `add`, – is `sub`, and so forth).

Note that the simple expression

```
var1 = const1 op const2;
```

is easily handled with a compile-time expression and a single `mov` instruction. For example, to compute

```
var1 = 5 + 3;
```

you would use the single instruction:

```
mov var1, #5 + 3
```

If *term2* is a (small enough) constant, you can typically use an instruction of the following form:

```
op  var1, term1, #constant
```

Exceptions exist, however. Certain instructions, such as `mul` and `udiv`/`sdiv`, do not allow immediate operands. In such cases, you'll need to use the two instructions

```
mov someReg, #constant
op  var1, term1, someReg
```

where *someReg* is an available temporary register.

If *term1* is a constant and *term2* is a register, you can get away with simply swapping the two source operands in the instruction for commutative operations. For example

```
x0 = 25 + x1;
```

becomes this:

```
add x0, x1, #25
```

For noncommutative operations, such as subtraction and division, this scheme doesn't work. You may have to load the constant into a register prior to the operation.

Of course, if the constant is too large (generally 12 bits for arithmetic instructions), you'll have to first load that constant into a register by using the `mov`, `movk`, or `ldr` instructions.

If your terms are memory variables rather than registers (or constants), you will need to use the `ldr` instruction to move the memory variable(s) into register(s) prior to the operation. Likewise, if the destination variable is in memory, you will have to use a `str` instruction to store the value after the operation is complete. For example

```
x = y + z;  // x, y, and z are all 32-bit memory variables.
```

becomes this:

```
ldr w0, [fp, #y]   // Assuming y is a local variable
ldr w1, [sb, #z]   // Assuming z is in the static base record
add w2, w0, w1
lea x3, globalVar  // Assuming globalVar is a global variable
str w2, [x3]       // in the .data section
```

Here are some examples of common simple expressions (assume `x`, `y`, and `z` are in W0, W1, and W2):

```
// x = y + z;  // Signed or unsigned

          add w0, w1, w2

// x = y - z;  // Signed or unsigned

          sub w0, w1, w2

// x = y * z;  // Signed or unsigned

          mul w0, w1, w2

// x = y / z;  // Unsigned div

          udiv w0, w1, w2
```

```
// x = y / z;  // Signed div

          sdiv w0, w1, w2

// x = y % z;  // Unsigned remainder

          udiv  x0, x1, x2
          msub  x0, x0, x2, x1

// x = y % z;  // Signed remainder

          sdiv  x0, x1, x2
          msub  x0, x0, x2, x1
```

If any of the operands are memory variables, you will first have to load them into registers by using the `ldr` instruction. If any operands are constants, follow the guidelines from the previous section.

6.3.3 Complex Expressions

A *complex expression* is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in an HLL. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, and so on. This section outlines the rules for converting such expressions.

Complex expressions that are easy to convert to assembly language involve three terms and two operators. Here's an example:

```
w = w - y - z;
```

Clearly, the straightforward assembly language conversion of this statement requires two `sub` instructions. However, even with an expression as simple as this, the conversion is not trivial. You can convert the preceding statement into assembly language in two ways (assume `w` is in W0, `y` is in W1, and `z` is in W2):

```
sub w0, w0, w1
sub w0, w0, w2
```

or

```
sub w3, w1, w2
sub w0, w0, w3
```

Both methods can produce different results, with the first conversion largely adhering to C language semantics. The problem is associativity. The second sequence in the preceding example computes `w = w - (y - z)`, which is not the same as `w = (w - y) - z`. The placement of the parentheses around the subexpressions can affect the result.

Precedence, the order in which operations occur, is another issue. Consider this expression:

```
x = w * y + z;
```

Once again, you can evaluate this expression in one of two ways:

```
x = (w * y) + z;
```

or

```
x = w * (y + z);
```

By now, you're probably thinking that this explanation is crazy—everyone knows the correct way to evaluate these expressions is to use the former form. However, this isn't always correct. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another. The "correct" method depends entirely on how you define precedence in your arithmetic system.

Consider this expression:

```
x op1 y op2 z
```

If *op1* takes precedence over *op2*, this evaluates to (x *op1* y) *op2* z. Otherwise, if *op2* takes precedence over *op1*, the expression evaluates to x *op1* (y *op2* z). Depending on the operators and operands involved, these two computations could produce different results.

Most HLLs use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Such programming languages usually compute multiplication and division before addition and subtraction. Those that support exponentiation (for example, FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive because most people learn them before high school.

When converting expressions into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique (assuming multiplication has higher precedence than addition):

```
// w = x + y * z;  // Assume w = W0, x = W1, y = W2, and z = W3.

mul w4, w2, w3  // W4 = W2 * W3
add w0, w1, w4  // W0 = W1 + (W2 * W3)
```

If two operators appearing within an expression have the same precedence, use the associativity rules to determine the order of evaluation. Most operators are *left-associative*, meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left-associative. A *right-associative* operator evaluates from right to left. The exponentiation

operator in FORTRAN is a good example of a right-associative operator. For instance:

```
2**2**3
```

is equal to

```
2**(2**3)
```

not

```
(2**2)**3
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w = x - y - z  // Assume w = W0, x = W1, y = W2, and z = W3.

sub  w0, w1, w2  // Evaluate from left to right.
sub  w0, w0, w3  // W0 = (x - y) - z

// w = x + y * z

mul  w0, w2, w3  // Must compute y * z first.
add  w0, w0, w1  // W0 = (W2 * W3) + W1  (commutative)
```

or, even better

```
madd w0, w2, w3, w1  // W0 = (W2 * W3) + W1

// w = x / y - z

sdiv w0, w1, w2  // Division has highest precedence.
sub  w0, w0, w3  // W0 = (W1 / W2) - W3

// w = x * y * z

mul  w0, w1, w2  // Commutative, so order doesn't matter.
mul  w0, w0, w3
```

The associativity rule has one exception: if an expression involves multiplication and division, it is generally better to perform the multiplication first. For example, given an expression of the form

```
w = x / y * z;  // Note: this is (x / y) * z, not x / (y * z).
```

it is usually better to compute `x * z` and then divide the result by `y`, rather than dividing `x` by `y` and multiplying the quotient by `z`. Doing the multiplication first increases the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5 / 2, you will get the value 2, not 2.5. Computing (5 / 2) × 3 produces 6. However, computing (5 × 3) / 2 gives you the value 7, which is a little closer to the real quotient (7.5).

Therefore, if you encounter an expression of the form

```
w = x / y * z;  // Assume w = W0, x = W1, y = W2, and z = W3.
```

you can usually convert it to the following assembly code:

```
mul  w0, w1, w3   // w = x * z
sdiv w0, w0, w2   // w = (x * z) / y
```

If the multiplication will likely produce an overflow, computing the division operation first may be better.

If the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. The moral of the story is that you should always make sure you fully understand any expression you are converting to assembly language. If the semantics dictate that you must perform the division first, do so.

Consider the following statement:

```
w = x - y * z;  // Assume w = W0, x = W1, y = W2, and z = W3.
```

Because subtraction is not commutative, you cannot compute `y * x` and then subtract `x` from this result. Rather than use a straightforward multiplication-and-subtraction sequence, you'll have to use a temporary register to hold the product. For example, the following two instructions use W4 as a temporary:

```
mul  w4, w2, w3  // temp = y * z
sub  w0, w1, w4  // w = x - (y * z)
```

As your expressions increase in complexity, the need for temporaries grows. Consider the following C statement:

```
w = (a + bb) * (y + z);
```

Following the normal rules of algebraic evaluation, compute the subexpressions inside the parentheses first (that is, the two subexpressions with the highest precedence) and set their values aside. When you've computed the values for both subexpressions, you can compute their product. One way to deal with a complex expression like this is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, you can convert the preceding single expression into the following sequence:

```
temp1 = a + bb;
temp2 = y + z;
w = temp1 * temp2;
```

Since converting simple expressions to assembly language is easy, it's now a snap to compute the former complex expression in assembly, as shown in the following code:

```
// Assume w = W0, y = W1, z = W2, a = W3, and bb = W4.

add  w5, w3, w4  // temp1 (W5) = a + bb
add  w6, w1, w2  // temp2 (W6) = y + z
mul  w0, w5, w6  // w = temp1 * temp2
```

Here's yet another example of a complex arithmetic conversion:

```
x = (y + z) * (a - bb) / 10;
```

You can convert this to a set of four simple expressions:

```
temp1 = (y + z)
temp2 = (a - bb)
temp1 = temp1 * temp2
x = temp1 / 10
```

You can convert these four expressions into the following assembly language statements:

```
// Assume x = W0, y = W1, z = W2, a = W3, and bb = W4.

add  w5, w1, w2  // temp1 (W5) = y + z
sub  w6, w3, w4  // temp2 (W6) = a - bb
mul  w5, w5, w6  // temp1 = temp1 * temp2
mov  w6, #10     // Need a temp to hold constant 10.
sdiv w0, w5, w6  // x = temp1 / 10
```

Most important, make sure you keep temporary values in registers for efficiency. Use memory locations to hold temporaries only if you've run out of registers.

In short, as you've seen, converting a complex expression to assembly language is a little different from solving the expression by hand. Instead of computing the result at each stage of the computation, you write the assembly code that computes the result.

6.3.4 Commutative Operators

If *op* represents an operator, that operator is *commutative* if the following relationship is always true:

```
(A op B) = (B op A)
```

As you learned in the previous section, commutative operators are easy to translate because the order of their operands is immaterial, which lets you rearrange a computation, often making it easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, check whether you can use a better sequence to improve the size or speed of your code.

Table 6-3 lists the commutative operators typically found in HLLs.

Table 6-3: Commutative Dyadic (Two-Operand) Operators

Pascal	C/C++ and similar	Description
`+`	`+`	Addition
`*`	`*`	Multiplication
`and`	`&&` or `&`	Logical or bitwise AND
`or`	`||` or `|`	Logical or bitwise OR
`xor`	`^`	Logical or bitwise exclusive-OR
`=`	`==`	Equality
`<>`	`!=`	Inequality

Table 6-4 lists many of the noncommutative operators.

Table 6-4: Noncommutative Dyadic Operators

Pascal	C/C++ and similar	Description
`-`	`-`	Subtraction
`/` or `div`	`/`	Division
`mod`	`%`	Remainder (modulo)
`<`	`<`	Less than
`<=`	`<=`	Less than or equal
`>`	`>`	Greater than
`>=`	`>=`	Greater than or equal

If you encounter any other operator types, check the associated HLL definition for the operators to determine whether they are commutative or noncommutative and determine their precedence and associativity.

6.4 Logical Expressions

Consider the following logical (Boolean) expression from a C/C++ program:

```
bb = ((x == y) && (a <= c)) || ((z - a) != 5);
```

Here, `bb` is a Boolean variable, and the remaining variables are all integers.

Though it takes only a single bit to represent a Boolean value, most assembly language programmers allocate a whole byte or even a word to represent Boolean variables. Most programmers (and, indeed, some programming languages like C) choose 0 to represent *false* and anything else to represent *true*. Some people prefer to represent true and false with 1 and 0, respectively, and not allow any other values. Others select all 1 bits (0xFFFF_FFFF_FFFF_FFFF, 0xFFFF_FFFF, 0xFFFF, or 0xFF) for true and 0 for false. You could also use a positive value for true and a negative value for false.

All these mechanisms have their advantages and drawbacks. Using only 0 and 1 to represent false and true offers two big advantages. First, the `cset` instruction produces this result, so this scheme is compatible with that instruction. Second, the ARM logical instructions (`and`, `orr`, `eor`, and, to a lesser extent, `mvn`) operate on these values exactly as you would expect. If you have two Boolean variables `a` and `bb`, the following instructions perform the basic logical operations on these two variables:

```
// d = a AND bb;  // Assume d = W0, a = W1, and bb = W2.

and w0, w1, w2

// d = a || bb;

orr w0, w1, w2

// d = a XOR bb;

eor w0, w1, w2

// bb = NOT a;
//
// (NOT 0) does not equal 1.
// The AND instruction corrects this problem.

mvn w2, w1
and w2, w2, #1

// Here's an alternative solution (for NOT) using EOR:

eor w2, w1, #1  // Inverts bit 0
```

The `mvn` instruction will not properly compute logical negation. The bitwise NOT of 0 is 0xFF (assuming a byte value), and the bitwise NOT of 1 is 0FEh. Neither result is 0 or 1. However, ANDing the result with 1 gives you the proper result. You can implement the NOT operation more efficiently by using the `eor` instruction (as shown in the last `eor` example just given) because it affects only the LO bit.

Using 0 for false and anything else for true has a lot of subtle advantages. The test for true or false is often implicit in the execution of any logical instruction. However, this mechanism has a major downside: you

cannot always use the ARM `and`, `orr`, `eor`, and `mvn` instructions to implement the Boolean operations of the same name. Consider the two values 0x55 and 0xAA. They're both nonzero, so they both represent the value true. However, if you logically AND 0x55 and 0xAA together using the ARM `and` instruction, the result is 0. True AND true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing Boolean operations.

A system that uses nonzero values to represent true and 0 to represent false is an *arithmetic logical system*. A system that uses two distinct values like 0 and 1 to represent false and true is called a *Boolean logical system*, or simply a Boolean system. You can use either system as convenient. Consider this Boolean expression:

```
bb = ((x == y) and (a <= d)) || ((z - a) != 5);
```

The resulting simple expressions might be as follows:

```
// Assume bb = W0, x = W1, y = W2, a = W3, d = W4, and z = W5.

cmp  w1, w2
cset w6, eq      // temp1 (W6) = x == y

cmp  w3, w4
cset w7, le      // temp2 (W7) = a <= d
and  w6, w6, w7  // temp1 = (x == y) && (a <= d)

sub  w7, w5, w3  // temp2 = z - a
cmp  w7, #5
cset w7, ne      // temp2 = (z - a) != 5

orr  w0, w6, w7  // W0 = temp1 || temp2
```

When working with Boolean expressions, don't forget that you might be able to optimize your code by simplifying it with algebraic transformations. In Chapter 7, you'll also see how to use control flow to calculate a Boolean result, which can be a bit more efficient than using the methods taught by the examples in this section.

6.5 Conditional Comparisons and Boolean Expressions

The conditional comparison instruction, `ccmp`, is quite useful for encoding complex Boolean expressions in assembly language. Consider the following Boolean expression:

```
bb = (x == y) && (a <= d)
```

Using the logic from the previous section, you could translate this into the following assembly language code:

```
// Assume bb = W0, x = W1, y = W2, a = W3, and d = W4.

cmp  w1, w2
cset w5, eq   // temp1 (W5) = x == y

cmp  w3, w4
cset w6, le   // temp2 (W6) = a <= d
and  w0, w5, w6   // bb = (x == y) && (a <= d)
```

By using the conditional comparison instruction, you can keep the temporary values in the condition code flags to shorten your code:

```
cmp  w1, w2
ccmp w3, w4, #0, eq
cset w0, le
```

The first `cmp` instruction sets the Z flag if `x` is equal to `y`. If that condition is false, the whole logical expression must return false. If it's true, this code has to test whether `a` is less than or equal to `d`.

Assuming that `x` does not equal `y`, the Z flag will be clear after the first `cmp` instruction. In that case, the `ccmp` instruction will not compare W3 (`a`) to W4 (`d`) but will load the flags with 0b0000 instead (because the `ccmp` instruction compares only the first two operands if the condition, `eq`, is true; at this point, it is not). Because all the flags are clear (meaning N == V and Z != 1), the `le` condition for the `cset` is false; therefore, that instruction will store a 0 into W0 (`bb`), exactly what you want.

On the other hand, if `x` is equal to `y`, the `eq` condition for the `ccmp` instruction will be true and will compare the value of W3 (`a`) to W4 (`d`). If `a` is less than or equal to `d`, the N, V, and Z flags will be set in such a way that the `cset` instruction moves a 1 into W0. Otherwise, `cset` will move a 0 into W0, which is again exactly what you want. This sequence with only three instructions does the work of the earlier sequence with five instructions, a huge win.

6.5.1 Implementing Conjunction Using ccmp

Consider this C/C++ logical expression:

```
(a cc1 bb) && (c cc2 d)
```

In general, to convert this expression containing the logical conjunction operator (`&&`) into ARM assembly using conditional comparison instructions, you would use the following five steps:

1. Compare the operands on the left-hand side of the conjunction operator, cc_1 (see Table 6-5).
2. Immediately after the first comparison, execute a `ccmp` instruction, supplying cc_1 as the conditional field.

3. Choose the corresponding #*nzcv* encoding from the opposite column in Table 6-5 to match cc_2. The full `ccmp` instruction should be:

```
ccmp c, d, #nzcv_op, cc_1
```

4. The last instruction in the sequence should test cc_2, as in the following example:

```
cset x0, cc_2
```

5. If cc_1 fails, the `ccmp` instruction will set the flags to the #$nzcv_{op}$ value and not compare `c` against `d`. Since you want the Boolean expression to yield false in this situation, choose an #$nzcv_{op}$ value that is the opposite of cc_2 so that the following test (for example, `cset`) produces a false result. If cc_1 is true upon executing the `ccmp` instruction, `ccmp` will compare `c` and `d` and set the flags.

Table 6-5: Conditional Operators, Opposites, and NZCV Settings

C/C++	Operator	#*nzcv*	Opposite	#$nzcv_{op}$
==	eq	0b0100	ne	0b0000
!=	ne	0b0000	eq	0b0100
> (unsigned)	hi	0b0010	ls	0b0100
>= (unsigned)	hs	0b0110	lo	0b0000
< (unsigned)	lo	0b0000	hs	0b0110
<= (unsigned)	ls	0b0100	hi	0b0010
> (signed)	gt	0b0000	le	0b0101
>= (signed)	ge	0b0100	lt	0b0001
< (signed)	lt	0b0001	ge	0b0100
<= (signed)	le	0b0101	gt	0b0000
Same as hs	cs	0b0010	cc	0b0000
Same as lo	cc	0b0000	cs	0b0010
N/A	vs	0b0001	vc	0b0000
N/A	vc	0b0000	vs	0b0001
N/A	mi	0b1000	pl	0b0000
N/A	pl	0b0000	mi	0b1000

Because keeping the flag settings for the third `ccmp` operand straight in your mind is difficult and error-prone, the *aoaa.inc* include file contains several defines to make it easy to choose these values, as well as some defines for opposite conditions. Table 6-6 lists these defines and their values.

Table 6-6: NZCV Constant Defines

Condition	Define	Value
`eq`	`cceq`	0b0100 (nZcv)
`ne`	`ccne`	0b0000 (nzcv)
`hi`	`cchi`	0b0010 (nzCv)
`hs`	`cchs`	0b0110 (nZCv)
`lo`	`cclo`	0b0000 (nzcv)
`ls`	`ccls`	0b0100 (nZcv)
`gt`	`ccgt`	0b0000 (nzcv)
`ge`	`ccge`	0b0100 (nZcv)
`lt`	`cclt`	0b0001 (nzcV)
`le`	`ccle`	0b0101 (nZcV)
`cs`	`cccs`	0b0010 (nzCv)
`cc`	`cccc`	0b0000 (nzcv)
`vs`	`ccvs`	0b0001 (nzcV)
`vc`	`ccvc`	0b0000 (nzcV)
`mi`	`ccmi`	0b1000 (Nzcv)
`pl`	`ccpl`	0b0000 (nzcv)

Table 6-7 lists some common antonyms (opposite conditions).

Table 6-7: NZCV Antonym Constants

Condition	Define	Same as
Not `hi`	`ccnhi`	`ccls`
Not `hs`	`ccnhs`	`cclo`
Not `lo`	`ccnlo`	`cchs`
Not `ls`	`ccnls`	`cchi`
Not `gt`	`ccngt`	`ccle`
Not `ge`	`ccnge`	`cclt`
Not `lt`	`ccnlt`	`ccge`
Not `le`	`ccnle`	`ccgt`

Using these symbols instead of constants for the immediate `ccmp` instruction operand can make your code easier to read and understand.

Sometimes specifying the opposite condition in one of the conditional instructions can create confusion. It's easy to think that the opposite of "less than" is "greater than" when it's actually "greater than or equal," for example. To help reduce this confusion, the *aoaa.inc* include file also provides defines for several opposite conditions, as listed in Table 6-8.

Table 6-8: Opposite Condition Defines

Condition	Opposite define
lo	nlo (same as hs)
ls	nls (same as hi)
hi	nhi (same as ls)
hs	nhs (same as lo)
gt	ngt (same as le)
ge	nge (same as lt)
lt	nlt (same as ge)
le	nle (same as gt)

By using the *aoaa.inc* definitions, you can make your code easier to read and understand.

6.5.2 Implementing Disjunction Using ccmp

The conditional comparison can also be used to simulate disjunction (logical OR). Consider the following expression:

```
bb = (x == y) || (a <= d)
```

Here's the translation of this expression to assembly language:

```
cmp  w1, w2
ccmp w3, w4, #0b0100, ne  // 0b0100 is .Z.. or use #cceq
cset w0, le               // or #ccle
```

Notice how the conditional compare instruction tests for the not equal condition. If x is equal to y, you don't need to do this comparison. In that case, the ccmp instruction will load 0b0100 into the condition codes, which sets Z to 1 and clears all the other flags. When the cset instruction tests for less than or equal, the equal condition (Z = 1) exists, setting W0 (bb) to 1. Comparing a and d plays no role in the computation of bb's value.

If x does not equal y, the ne condition will exist when the program executes the ccmp instruction. Therefore, ccmp will compare a and d and set the condition code bits on the basis of that comparison. At that point, the cset instruction will set bb's value based on the comparison of a and d.

The following algorithm describes how to convert an expression containing disjunction into ARM assembly language using a conditional comparison:

```
(a cc1 bb) || (c cc2 d)
```

Here are the four steps to follow for this conversion:

1. Compare the operands on the left-hand side of the disjunction operator (operator is cc_1).

2. Immediately after the first `cmp` instruction, execute a `ccmp` instruction, supplying the opposite of cc_1 as the conditional field (return to Table 6-5 to find the opposite conditions).
3. Choose the corresponding `#nzcv` encoding from the regular column in Table 6-5 to match cc_2. The full `ccmp` instruction should be as follows:

```
ccmp c, d, #nzcv, opposite(cc1)
```

4. The last instruction in the sequence should test cc_2. For example:

```
cset x0, cc2
```

If cc_1 succeeds, the `ccmp` instruction will set the flags to the $\#nzcv_{op}$ value and not compare `c` against `d`, because you've chosen the opposite of cc_1 for the `ccmp` condition. As you want the Boolean expression to yield true in this situation, choose an $\#nzcv_{op}$ value that is the same as cc_2 so that the following test (for example, `cset`) produces a true result. If cc_1 is false upon executing the `ccmp` instruction, `ccmp` will compare `c` and `d` and set the flags appropriate for the following test.

6.5.3 Handling Complex Boolean Expressions

You can extend the Boolean expressions by adding additional `ccmp` instructions to the sequence. Just keep in mind that, at least in C/C++, conjunction has higher precedence than disjunction, so you must modify your order of evaluation to handle conjunction first when expressions contain both operators.

Also note that the `ccmp` scheme uses *complete Boolean evaluation* (meaning it evaluates every subterm of the Boolean expression), whereas the C++ programming language uses *short-circuit Boolean evaluation* (which may not compute all subterms). Chapter 7 covers these two forms in greater detail, but for now, just know that the two forms may produce different results.

6.6 Machine and Arithmetic Idioms

An *idiom* is an idiosyncrasy (a peculiarity). Several arithmetic operations and ARM instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as *tricky programming* that you should always avoid in well-written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well known and commonly found in assembly language programs. This section provides an overview of the idioms you'll see most often.

6.6.1 Multiplying Without mul

When multiplying by a constant, you can sometimes write equivalent code by using shifts, additions, and subtractions in place of multiplication

instructions. Although performance differs little between using a `mul` instruction and other arithmetic instructions, some addressing mode variants involving shifts can spare you an extra multiply instruction.

Remember, a `lsl` instruction computes the same result as multiplying the specified operand by 2. Shifting to the left two bit positions multiplies the operand by 4. Shifting to the left three bit positions multiplies the operand by 8. In general, shifting an operand to the left n bits multiplies it by 2^n. You can multiply any value by a constant by using a series of shifts and additions or shifts and subtractions. For example, to multiply the W0 register by 10, you need only multiply it by 8 and then add 2 times the original value. That is, $10 \times W0 = 8 \times W0 + 2 \times W0$. Use the following code to accomplish this:

```
lsl w0, w0, #1          // W0 = W0 * 2
add w0, w0, w0, lsl #2  // W0 = (W0 * 2) + (W0 * 8)
```

The first instruction multiplies W0 by 2, so when the second instruction shifts W0 2 bits to the left, it's actually shifting the original W0 value to the left by 3 bits.

Looking at the instruction timings, you'll see that the multiply instruction executes at the same speed as the `lsl` or `add` instructions, so this second sequence isn't faster. However, if you have to load the constant 10 into a register to do the multiplication by 10, this sequence is no slower. If you've already done the shift as part of another calculation, this sequence could turn out to be faster.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by 7:

```
sub w0, w0, w0, lsl #3  // Actually computes W0 * (-7)
neg w0, w0              // Fix sign.
```

Beginning assembly language programmers commonly make the error of subtracting or adding 1 or 2 rather than $W0 \times 1$ or $W0 \times 2$. The following does not compute $W0 \times 7$:

```
lsl w0, w0, #3
sub w0, w0, #1
```

Rather, this code computes $(8 \times W0) - 1$, which is entirely different (unless, of course, $W0 = 1$). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

The Operand2 addressing mode variations, particularly those involving `lsl`, are quite useful for combining shifts along with other arithmetic operations. For example, consider the following pair of instructions:

```
lsl w0, w0, #3
add w1, w1, w0
```

You can easily replace this by a single instruction:

```
add w1, w1, w0, lsl #3
```

Because RISC CPUs, such as the ARM, tend to execute most instructions in a single CPU clock cycle, using *strength-reduction optimizations* like substituting shifts and adds for multiplication rarely pays off. Generally, a single shift instruction (for a multiplication by a power of 2) may produce better results than `mul`; beyond that, it's unlikely to improve the speed, unless you need those shifts and adds for other calculations.

6.6.2 Dividing Without sdiv or udiv

Just as the `lsl` instruction is useful for simulating a multiplication by a power of 2, the `lsr` and `asr` instructions can simulate a division by a power of 2. Unfortunately, you cannot easily use shifts, additions, and subtractions to perform division by an arbitrary constant. Therefore, this trick is useful only when dividing by powers of 2. Also, don't forget that the `asr` instruction rounds toward negative infinity, unlike the `sdiv` instruction, which rounds toward 0.

On the ARM64 CPU, the division instructions tend to take about twice as long as other instructions to execute. Therefore, if you can simulate a division by a power of 2 by using a single shift-right instruction, your code will run a little faster. You can also divide by a value by multiplying by its reciprocal. This is usually faster than division, since the multiply instruction is faster than the divide instruction.

To multiply by a reciprocal when dealing with integers, you must cheat. If you want to multiply by 1/10, there is no way you can load the value 1/10 into an ARM integer register prior to performing the multiplication. It won't work to multiply 1/10 by 10, perform the multiplication, and divide the result by 10 to get the final result. In fact, this would make performance worse, because you're now doing a multiplication by 10 as well as a division by 10. However, suppose you multiply 1/10 by 65,536 (6,554), perform the multiplication, and then divide by 65,536. Consider the following code that divides W0 by 10:

```
mov w1, #6554
mul w0, w0, w1
lsr w0, w0, #16  // Division by 65,536
```

This code leaves W0 / 10 in the W0 register. To understand how this works, consider what happens when you use the `mul` instruction to multiply W0 by 65,536 (0x1_0000). This moves the LO half word of W0 into the HO half word and sets the LO half word to 0 (a multiplication by 0x1_0000 is equivalent to a shift left by 16 bits). Multiplying by 6,554 (65,536 divided by 10) puts W0 divided by 10 into the HO half word of the W0 register.

Multiplying by a reciprocal works well only when dividing by a constant, such as 10. While you could coerce the calculation with multiple instructions to divide a register by a nonconstant value, the `udiv/sdiv` instructions

would certainly be faster by that point; it's questionable whether multiplying by a reciprocal is faster than a division.

6.6.3 Implementing Modulo-N Counters with AND

To implement a counter variable that counts up to $2^n - 1$ and then resets to 0, use the following code

```
add w0, w0, #1
and w0, w0, #nBits
```

where `nBits` is a binary value containing n bits of 1s right-justified in the number. For example, to create a counter that cycles from 0 to 15 ($2^4 - 1$), you could use the following:

```
add w0, w0, #1
and w0, w0, #0b1111
```

6.6.4 Avoiding Needlessly Complex Machine Idioms

The machine idioms you've just learned work well to improve performance on older complex instruction set computers (CISCs) that typically take a varying number of CPU clock cycles to execute each instruction. Complex instructions like division can take upward of 50 clock cycles on an x86 CPU, for example. RISC CPUs, such as the ARM, try to execute instructions in one clock cycle. While the ARM doesn't always achieve this (`sdiv` and `udiv` are a little slower, for example), the additional time required doesn't justify replacing the instruction with a long sequence of other instructions.

Using machine idioms makes your code harder to read and understand. If using a machine idiom offers no clear performance benefit, stick with using easier-to-understand code. Those who work on your project afterward (including yourself, in the future) will thank you.

6.7 Floating-Point and Finite-Precision Arithmetic

Before discussing how the ARM CPU implements floating-point arithmetic, it is worthwhile to first describe the mathematical theory behind floating-point arithmetic and the problems you will encounter when using it. This section presents a simplified model to explain floating-point arithmetic and why you cannot apply standard algebraic rules to calculations involving it.

6.7.1 Basic Floating-Point Terminology

Integer arithmetic does not let you represent fractional numeric values. Therefore, modern CPUs support an approximation of *real* arithmetic: *floating-point arithmetic*. To represent real numbers, most floating-point formats employ scientific notation and use a certain number of bits to represent a mantissa and a smaller number of bits to represent an exponent.

For example, in the number 3.456e+12, the mantissa consists of 3.456, and the exponent digits are 12. Because the number of bits is fixed in computer-based representations, computers can represent only a certain number of digits (known as *significant digits*) in the mantissa. For example, if a floating-point representation could handle only three significant digits, then the fourth digit in 3.456e+12 (the 6) could not be accurately represented with that format, as three significant digits can represent only 3.45e+12 or 3.46e+12 correctly.

Because computer-based floating-point representations also use a finite number of bits to represent the exponent, that exponent also has a limited range of values, approximately ranging from $10^{\pm 38}$ for the single-precision format to about $10^{\pm 308}$ for the double-precision format. This is known as the *dynamic range* of the value. Denormalized numbers (which I'll define shortly) can represent values as small as $\pm 4.94066 \times 10^{-324}$.

6.7.2 Limited-Precision Arithmetic and Accuracy

A big problem with floating-point arithmetic is that it does not follow the standard rules of algebra. Normal algebraic rules apply only to *infinite-precision* arithmetic. Therefore, if you translate an algebraic formula into code, that code might produce different results from what you would (mathematically) expect. This can introduce defects in your software.

Consider the simple statement $x = x + 1$, where x is an integer. On any modern computer, this statement follows the normal rules of algebra *as long as overflow does not occur*. That is, this statement is valid only for certain values of x ($minint \leq x < maxint$). Most programmers do not have a problem with this because they are well aware that integers in a program do not follow the standard algebraic rules (for example, 5 / 2 does not equal 2.5).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating-point values suffer from this same problem, only worse. After all, integers are a subset of real numbers. Therefore, the floating-point values must represent the same infinite set of integers. However, an infinite number of real values exist between any two integer values. In addition to having to limit your values between a maximum and minimum range, you cannot represent all the values between any pair of integers either.

To demonstrate the impact of limited-precision arithmetic, this chapter adopts a simplified decimal floating-point format for our examples. This format provides a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values, as shown in Figure 6-1.

Figure 6-1: A floating-point format

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. Multiplication and division don't require the exponents to be the same; instead, the exponent after a multiplication is the sum of the two operand exponents, and the exponent after a division is the difference of the dividend and divisor's exponents.

For example, when adding 1.2e1 and 4.5e0, you must adjust the values so that they have the same exponent. One way to do this is to convert 4.5e0 to 0.45e1 and then add, producing 1.65e1. Because the computation and result require only three significant digits, you can compute the correct result via the representation shown in Figure 6-1.

However, suppose you want to add the two values 1.23e1 and 4.56e0. Although both values can be represented using the three-significant-digit format, the computation and result do not fit into three significant digits. That is, 1.23e1 + 0.456e1 requires four digits of precision in order to compute the correct result of 1.686, so you must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so round the result to obtain 1.69e1.

In fact, the rounding does not occur after adding the two values together (that is, producing the sum 1.686e1 and then rounding this to 1.69e1), but rather when converting 4.56e0 to 0.456e1, because four digits of precision are required to maintain the value 0.456e1. Therefore, during the conversion, you have to round 0.456e1 to 0.46e1 so that the result fits into three significant digits. The sum of 1.23e1 and 0.46e1 then produces the final rounded sum of 1.69e1.

As you can see, the lack of *precision* (the number of digits or bits maintained in a computation) affects the *accuracy* (the correctness of the computation). In the addition/subtraction example, you could round the result because you maintained four significant digits *during* the calculation (specifically, when converting 4.56e0 to 0.456e1). If your floating-point calculation had been limited to three significant digits during computation, you would have had to truncate the last digit of the smaller number, obtaining 0.45e1 and producing a sum of 1.68e1, a value that is even less accurate.

To improve the accuracy of floating-point calculations, it is useful to maintain one or more extra digits for use during the calculation, such as the extra digit used to convert 4.56e0 to 0.456e1. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

6.7.3 Errors in Floating-Point Calculations

In a sequence of floating-point operations, errors can accumulate and greatly affect the computation itself. For example, suppose you were to add 1.23e3 to 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable;

after all, you can maintain only three significant digits, so adding in a small value shouldn't affect the result at all.

However, suppose you were to add 1.00e0 to 1.23e3 *ten times* (though not in the same calculation, where guard digits could maintain the fourth digit during the calculation). The first time you add 1.00e0 to 1.23e3, you get 1.23e3. You get this same result the second, third, fourth . . . and tenth times you add 1.00e0 to 1.23e3. On the other hand, had you added 1.00e0 to itself 10 times, then added the result (1.00e1) to 1.23e3, you would have gotten a different result, 1.24e3. Keep in mind this important guideline for limited-precision arithmetic:

> When performing complex operations, watch the order of evaluation, as it can affect the accuracy of the result.

You'll get more accurate results if the relative magnitudes (the exponents) are close to one another when adding and subtracting floating-point values. If you're performing a chain calculation involving addition and subtraction, attempt to group the values appropriately.

When computing addition and subtraction, you can also wind up with *false precision*. Consider the computation 1.23e0 – 1.22e0, which produces 0.01e0. Although the result is mathematically equivalent to 1.00e – 2, this latter form suggests that the last two digits are exactly 0. Unfortunately, you have only a single significant digit at this time (remember, the original result was 0.01e0, and those two leading 0s were significant digits). Indeed, some floating-point unit (FPU) or software packages might actually insert random digits (or bits) into the LO positions. This highlights a second important rule concerning limited-precision arithmetic:

> When subtracting two numbers with the same signs (or adding two numbers with different signs), be aware that the result may contain high-order significant digits (bits) that are 0. This reduces the number of significant digits (bits) by a like amount in the final result. If possible, try to arrange your calculations to avoid this.

By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error that already exists in a value. For example, if you multiply 1.23e0 by 2 when you should be multiplying 1.24e0 by 2, the result is even less accurate. This leads to a third important rule for working with limited-precision arithmetic:

> When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute $x * (y + z)$. Normally, you would add y and z together and multiply their sum by x. However, your results will be a little more accurate if you transform $x * (y + z)$ to get $x * y + x *$ z and compute the result by performing the multiplications first. Of course, the drawback

is that you must now perform two multiplications rather than one, so the result may be slower.

Multiplication and division have their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number, or dividing a large number by a small (fractional) number. This brings us to a fourth rule to follow when multiplying or dividing values:

> When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

6.7.4 Floating-Point Value Comparisons

Given the inaccuracies present in any computation (including converting an input string to a floating-point value), you should *never* compare two floating-point values to see if they are equal. In a binary floating-point format, different computations that produce the same (mathematical) result may differ in their least significant bits. For example, 1.31e0 + 1.69e0 should produce 3.00e0. Likewise, 1.50e0 + 1.50e0 should also produce 3.00e0. However, if you were to compare (1.31e0 + 1.69e0) against (1.50e0 + 1.50e0), you might find out that these sums are *not* equal to each other.

The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Because this is not necessarily true after two different floating-point computations that should produce the same result, a straight test for equality may not work. Instead, use the following test:

```
if Value1 >= (Value2 - error) and Value1 <= (Value2 + error) then ...
```

Another common way to handle this same comparison is to use a statement of this form:

```
if abs(Value1 - Value2) <= error then ...
```

In these statements, *error* should be a value slightly greater than the largest amount of error that will creep into your computations. The exact value will depend on the particular floating-point format you use. In short, follow this final rule:

> When comparing two floating-point numbers, always compare one value to see whether it is in the range given by the second value plus or minus a small error value.

Many other little problems can occur when using floating-point values. This book points out only some of the major problems and will make you aware that you cannot treat floating-point arithmetic like real arithmetic because of the inaccuracies present in limited-precision arithmetic. A good text on numerical analysis or even scientific computing can help fill in the

details. If you plan to work with floating-point arithmetic in any language, take the time to study the effects of limited-precision arithmetic on your computations (see section 6.13, "For More Information," on page 352).

Now that you've seen the theory behind floating-point arithmetic, we'll review the ARM's implementation of floating-point.

6.8 Floating-Point Arithmetic on the ARM

When the ARM CPU was first designed, floating-point arithmetic was among the set of "complex" instructions that RISC CPUs avoided. Those who required floating-point arithmetic were forced to implement it in software. As time passed, it became clear that high-performance systems required fast floating-point arithmetic, so it was added to the ARM's instruction set.

The ARM64 supports the IEEE single- and double-precision floating-point formats (see section 2.13, "IEEE Floating-Point Formats," on page 93), as well as a 16-bit half-precision floating-point format that appeared in later revisions of the IEEE standard. To support floating-point arithmetic, the ARM provides an extra set of registers and augments the instruction set with suitable floating-point instructions. Originally, these types of instructions were handled by coprocessors—separate chips that handled floating-point instructions (while the main CPU handled integer operations). In the ARM64 architecture, the FPU is built into the main CPU's integrated circuit.

The following subsections introduce the floating-point register set, the floating-point status register, and the floating-point control register. These are the programmer-visible components of the floating-point hardware on the ARM CPU.

6.8.1 Neon Registers

To support floating-point arithmetic, the ARM64 provides a second set of 32 registers specifically tailored to hold floating-point and other values. These are known as the *Neon registers* because, in addition to supporting scalar floating-point (FP) arithmetic, they also support vector arithmetic using the Neon instruction set extensions, covered in Chapter 11.

The 32 main FP/Neon registers are 128 bits each. Just as the general-purpose registers are divided into two sets based on their size (W*n* and X*n*), the FP/Neon registers are broken into five groups based on their size:

V0 to V31 The 128-bit *vector* registers (for Neon instructions), also referenced as Q0 to Q31, the qword registers. The V*n* names support special syntax for vector operations.

D0 to D31 The 64-bit *double-precision* floating-point registers.

S0 to S31 The 32-bit *single-precision* floating-point registers.

H0 to H31 The 16-bit *half-precision* floating-point registers.

B0 to B31 The 8-bit *byte* registers.

In addition to the 32 main registers, this set includes two special-purpose floating-point registers: the floating-point status register (FPSR) and the floating-point control register (FPCR), shown in Figure 6-2. You'll learn more about these registers in the following subsections.

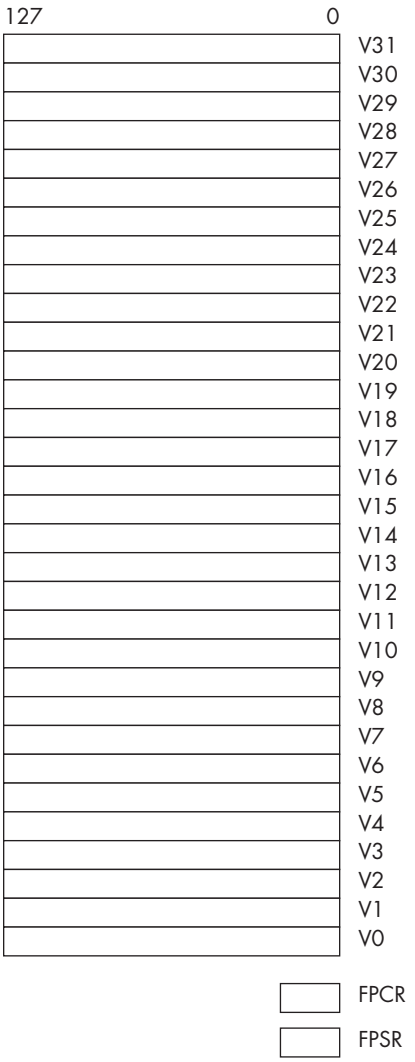


Figure 6-2: The FP/Neon registers

The B*n*, H*n*, S*n*, D*n*, and V*n* registers overlay one another, as shown in Figure 6-3.

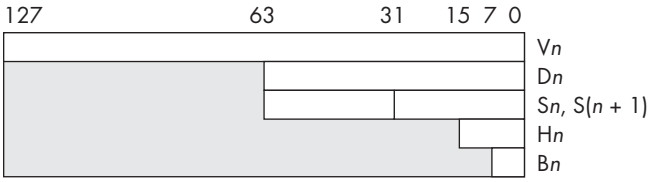


Figure 6-3: The FP/Neon register overlays

For historical reasons, the even-numbered single-precision registers (S0, S2, . . . , S30) are mapped to bits 0 through 31 in D0 through D15, and the odd-numbered single-precision registers are mapped to bits 32 through 64. No S*n* registers are mapped to D16 through D31 (see Figure 6-4).

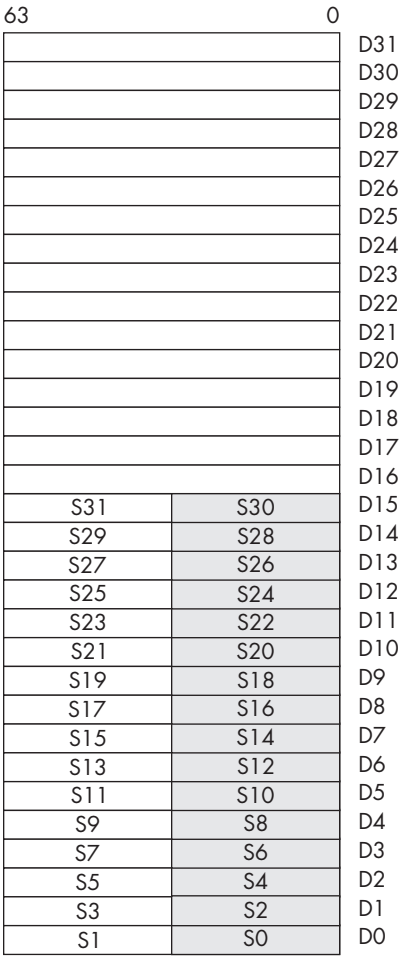


*Figure 6-4: How S*n *registers overlay D*n *registers*

The following sections concentrate mainly on the D*n* and S*n* register sets. This book doesn't discuss half-precision floating-point arithmetic in

depth, as it's used mainly by graphics processing units (GPUs) and certain graphics routines. The floating-point hardware doesn't actually work with half-precision values—it only allows you to convert between half- and single- or double-precision values.

Most of the ARM floating-point instructions operate on the D*n* or S*n* registers. This chapter collectively refers to these registers as F*n*, meaning you can substitute any double- or single-precision register for F*n*. I will also note exceptions as needed. Vector registers (V*n*) are the subject of Chapter 11.

6.8.2 Control Register

The *floating-point control register (FPCR)* specifies how certain floating-point operations take place. Although this register is 32 bits, only 6 bits are used, as you can see in Figure 6-5.

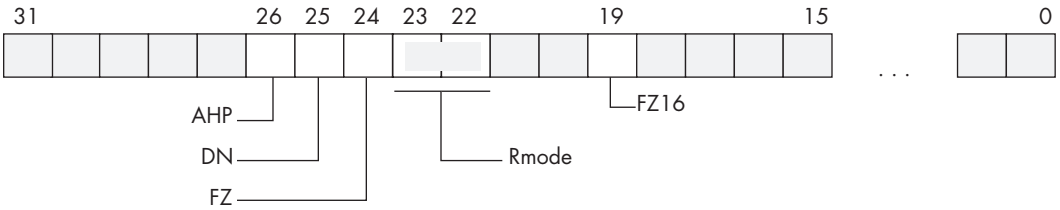


Figure 6-5: The FPCR layout

Table 6-9 describes the meaning of each of these bits.

Table 6-9: FPCR Bits

Bit(s)	Name	Description
19	FZ16	Flush-to-zero mode for half-precision arithmetic. 0 = disabled, 1 = enabled. This replaces denormalized values with 0. The result may not be as precise, but the instructions may execute faster.
22, 23	Rmode	Rounding mode: 00 = round to nearest, 01 = round to +infinity, 10 = round to –infinity, 11 = truncate (round toward 0).
24	FZ	Flush-to-zero mode for single- and double-precision arithmetic.
25	DN	Default NaN (not a number) mode. 0 = disable default NaN mode, 1 = enable. When disabled, NaNs propagate through arithmetic operations; when enabled, invalid operations return the default NaN.
26	AHP	Alternate half-precision bit. Enables (1) alternate half-precision mode or (0) IEEE half-precision mode.

For the most part, you'll leave all these bits set to 0. Setting Rmode to 0b11 is a reasonable change when you want to truncate rather than round a floating-point calculation.

To manipulate the FPCR register, use the `mrs` (move system to register) and `msr` (move register to system) instructions, specifying FPCR as the system register:

```
mrs  Xn, FPCR  // Copies FPCR to Xn
msr  FPCR, Xn  // Copies Xn to FPCR
```

For example, to clear all the (defined) bits in the FPCR, you'd use the following instructions:

```
mrs  x0,   fpcr
mov  x1,   #0xffff          // Load 0xf836ffff into X1, which is
movk x1,   #0xf836, lsl #16 // not a valid logical instr immediate value.
and  x0,   x0, x1           // Must put it in a register.
msr  fpcr, x0
```

Set the rounding mode to truncate with the following instructions:

```
mrs  x0, fpcr
orr  x0, x0, #0x00c00000 // Is valid logical instr immediate value
msr  fpcr, x0
```

The default FPCR settings are unknown on a warm reset, so you should always initialize this register before performing floating-point operations.

6.8.3 Status Register

The FPSR holds status information about ARM floating-point hardware. Reading this register provides the current floating-point status, while writing to it allows you to clear exception conditions. Although this is a 32-bit register, only 11 bits are defined and, in fact, only 7 of those are used in 64-bit mode (see Figure 6-6).

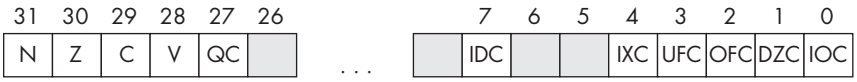


Figure 6-6: The FPSR layout

Table 6-10 describes the purpose of each of the bits in the FPSR.

Table 6-10: FPSR Bits

Bit(s)	Name	Definition
0	IOC	Invalid operation cumulative flag. This bit is set when the result of an operation has no mathematical value or cannot be represented.
1	DZC	Division by zero cumulative flag. This bit is set when a division by zero occurs.
2	OFC	Overflow cumulative flag. This bit is set when a floating-point operation causes an overflow situation.

(continued)

Table 6-10: FPSR Bits *(continued)*

Bit(s)	Name	Definition
3	UFC	Underflow cumulative flag. This bit is set when underflow occurs during an arithmetic operation.
4	IXC	Inexact cumulative flag. This bit is set (often!) when a floating-point operation produces an inexact result.
7	IDC	Input denormal cumulative flag. This bit is set when a denormalized input operand is replaced in the computation by a zero.
27	QC	Saturation cumulative flag. This flag is set when a saturation instruction clips a value. See Chapter 11 for a discussion of the saturating instructions.
28–31	N, C, Z, V	These flags are used only in 32-bit mode. In 64-bit mode, the floating-point comparisons and other instructions directly set the N, Z, C, and V flags in the PSTATE register.

You can read and write the FPSR with the `mrs` and `msr` instructions, using FPSR as the system register name. Read the FPSR to determine if any floating-point exceptions have occurred, and write the FPSR to clear the exception bits (by writing 0s to the affected bits in the register). For example the following code clears the Invalid Operation Cumulative flag in the FPSR:

```
mrs  x0, FPSR
and  x0, x0, #-2  // Clear IOC bit (-2 is 0xFFFF...FE).
msr  FPSR, x0
```

6.9 Floating-Point Instructions

The FPU adds many instructions to the ARM instruction set. I will classify these as data movement instructions, conversions, arithmetic instructions, comparisons, and miscellaneous instructions. This section describes each instruction in these categories.

6.9.1 FPU Data Movement Instructions

The *data movement instructions* transfer data between the internal FPU registers and memory. The instructions in this category are `ldr/ldur`, `str/stur`, `ldp/ldnp`, `stp/stnp`, and `fmov`.

6.9.1.1 ldr/ldur and str/stur

The `ldr` and `str` instructions load one of the FPU registers from a memory location, using the normal memory addressing modes. The `ldur/stur` instructions force an unscaled load or store operation, for cases where the assembler might choose a scaled indirect-plus-offset mode. Generally, rather than using `ldur/stur`, you'd let the assembler pick the appropriate underlying machine coding for you.

You can specify any of the FPU register names when using this instruction. For example, the following code loads the specified floating-point registers from memory:

```
ldr q0, [x0]  // Loads 128 bits from memory
ldr d0, [x0]  // Loads 64 bits from memory
ldr s0, [x0]  // Loads 32 bits from memory
ldr b0, [x0]  // Loads 8 bits from memory
```

6.9.1.2 ldp/ldnp and stp/stnp

The `ldp` and `stp` instructions work similarly to their integer counterparts with floating-point registers: they load or store a pair of registers at a time. These instructions do not support the H*n* or B*n* registers; you can load only word, dword, or qword FPU registers using these instructions.

The following examples demonstrate loading 256, 128, and 64 bits from memory:

```
ldp q0, q1, [x0]  // Loads 256 bits from memory
ldp d0, d1, [x0]  // Loads 128 bits from memory
ldp s0, s1, [x0]  // Loads 64 bits from memory
```

The `ldnp` and `stnp` instructions do nontemporal loads and stores. This informs the CPU that you don't intend to access the specified memory location again in the near future, so the CPU won't copy the data into its cache (a convenient example of what you can do in assembly and not in an HLL). This can improve performance by helping to prevent a situation known as *thrashing*, in which the CPU constantly moves data in and out of the cache memory.

6.9.1.3 fmov

The `fmov` instruction transfers data between two like-sized floating-point registers (where both registers are either 32 or 64 bits), or between a 32- or 64-bit general-purpose (GP) register and a like-sized floating-point register. Here is the allowable syntax for this instruction:

```
fmov Sd, Sn  // Move data between two 32-bit FP registers.
fmov Dd, Dn  // Move data between two 64-bit FP registers.
fmov Sd, Wn  // Move data from a 32-bit GP to an FP register.
fmov Wd, Sn  // Move data from a 32-bit FP to a GP register.
fmov Dd, Xn  // Move data from a 64-bit GP to an FP register.
fmov Xd, Dn  // Move data from a 64-bit FP to a GP register.
```

Moving a general-purpose register into a floating-point register does not convert an integer value in the GP register to a floating-point value; such an `fmov` operation assumes that the GP register contains the bit pattern for a floating-point number. Likewise, moving a floating-point register into a general-purpose register does not convert the floating-point value into an integer.

6.9.1.4 fmov with Immediate Operand

The ARM provides an `fmov` instruction that allows a very limited immediate operand. The syntax is as follows

```
fmov Sd, #fimm
fmov Dd, #fimm
```

where *fimm* is a floating-point constant from a very small set of possible values. The allowable values are $\pm n / 16 \times 2^m$, where $16 \le n \le 31$ and $-3 \le m \le 4$. This means you can represent values such as 1.0 or –2.0 but cannot represent 1.2345e5.

You cannot represent the value 0.0 with this immediate form. However, you can load 0.0 into a floating-point register by using one of the following two instructions:

```
fmov Sd, wzr
fmov Dd, xzr
```

If you want to load an arbitrary floating-point constant into a register, you will have to stick that constant into a memory location, using the `.single` or `.double` directive, and load the register from that location. Unfortunately, the `ldr` instruction doesn't accept floating-point immediate operands:

```
ldr d0, =10.0   // Generates an error
```

Fortunately, the PC-relative addressing mode does work, so you can access memory locations you've initialized in your `.text` section (preferably in the `.pool` area), as the following example demonstrates:

```
        .code
        .pool
fp10:   .double 10.0
         .
         .
         .
        ldr     d0, fp10
```

By adding the `.pool` directive, Gas can embed other assembler-generated constants in this area too.

6.9.2 FPU Arithmetic Instructions

The ARM CPU provides a large set of floating-point instructions that operate on single-precision and double-precision floating-point values. As for the integer operations, most of these instructions require three (floating-point) register operands: a destination, a left source, and a right source.

Table 6-11 lists the syntax for the arithmetic instructions. In this table, `Fd`, `Fn`, `Fm`, and `Fa` represent floating-point registers and can be S*n* or D*n*

(n = 0 to 31), depending on the precision of the instruction. For a given instruction, all registers must be the same size (32 or 64 bits).

Table 6-11: Floating-Point Arithmetic Instructions

Instruction	Operands	Description
`fadd`	`Fd, Fn, Fm`	Fd = Fn + Fm
`fsub`	`Fd, Fn, Fm`	Fd = Fn – Fm
`fmul`	`Fd, Fn, Fm`	Fd = Fn × Fm
`fnmul`	`Fd, Fn, Fm`	Fd = –(Fn × Fm)
`fmadd`	`Fd, Fn, Fm, Fa`	Fd = Fa + Fn × Fm
`fmsub`	`Fd, Fn, Fm, Fa`	Fd = Fa – Fn × Fm
`fnmadd`	`Fd, Fn, Fm, Fa`	Fd = –(Fa + Fn × Fm)
`fnmsub`	`Fd, Fn, Fm, Fa`	Fd = –(Fa – Fn × Fm)
`fdiv`	`Fd, Fn, Fm`	Fd = Fn / Fm
`fmax`	`Fd, Fn, Fm`	Fd = max(Fn, Fm), NaN if either operand is NaN
`fmaxnm`	`Fd, Fn, Fm`	Fd = max(Fn, Fm), number if other operand is (quiet) NaN
`fmin`	`Fd, Fn, Fm`	Fd = min(Fn, Fm), NaN if either operand is NaN
`fminnm`	`Fd, Fn, Fm`	Fd = min(Fn, Fm), number if other operand is (quiet) NaN
`fabs`	`Fd, Fn`	Fd = fabs(Fn), absolute value
`fneg`	`Fd, Fn`	Fd = –Fn
`fsqrt`	`Fd, Fn`	Fd = sqrt(Fn)

Many operations can raise an exception of one sort or another. For example, `fdiv` can set the DZC flag in the FPSR if a division by 0 occurs. Some operations, such as `fsqrt`, can produce an invalid result—for example, when trying to take the square root of a negative number. After a sequence of floating-point instructions, check the FPSR to see if the result obtained is valid. The FPSR bits are sticky and will remain set once an exception occurs; this allows you to check for an error at the end of a chain of calculations, rather than after each floating-point instruction.

SIGNALING VS. QUIET NANS

NaNs come in two varieties: signaling and quiet. When a *quiet NaN* occurs, the operations set the result to NaN (a special floating-point value; see section 2.14.1, "Nonnumeric Values," on page 97). Any further operations quietly propagate this value throughout the calculation so that the final result remains NaN.

Signaling NaNs, on the other hand, can raise an exception when the bad calculation occurs. This functionality can be enabled or disabled with the DN bit (bit 25) in the FPCR.

(continued)

When exceptions are enabled, the CPU invokes a special trap handler anytime an exception occurs. When disabled, the CPU will set only the status bits and pass NaNs through the calculation as an indication that an exception happened. The exception handler is generally provided by the OS and enabled via OS system calls; writing an exception handler to deal with this situation is beyond the scope of this book. Fortunately, exception processing is normally turned off by default, and you must explicitly test for exceptions by reading the FPSR.

6.9.3 Floating-Point Comparisons

The ARM provides a floating-point compare and a conditional compare instruction. Both have a couple of forms

```
fcmp   Fd, Fs
fcmpe  Fd, Fs
fcmp   Fd, #0.0
fcmpe  Fd, #0.0

fccmp  Fd, Fs, #nzcv, cond
fccmpe Fd, Fs, #nzcv, cond
```

where *nzcv* and *cond* have the same meanings they did with the `ccmp` instruction.

The instructions with the `e` suffix raise an exception if either operand is NaN during the comparison. Dealing with exceptions raised by these instructions is beyond the scope of this book, so subsequent example code uses just the forms without the `e` suffix.

The `fcmp` instruction will compare an FPU register against either another FPU register or the immediate constant `0.0`. If you need to compare against any other floating-point constant, you'll have to first load that into a register. Note that `fccmp` doesn't provide a form that allows a comparison against `0.0` (although you can copy XZR or WZR into another FPU register and compare against that).

6.9.3.1 Comparison Logic

The `fcmp` instruction sets the (PSR, not FPSR) condition code bits N, Z, C, and V in response to the comparison, allowing you to use the conditional branches and other conditional instructions to test the result of the comparison. However, the behavior of the settings is a bit different from integer comparisons. First of all, there aren't unsigned and signed comparisons (floating-point values are always signed); second, floating-point comparisons can be unordered.

Unordered comparisons occur when one or both of two values you're comparing are NaN, since two values are incomparable under those

circumstances. At best, you can say they are not equal to each other; it's safer simply to say the result is unordered and leave it at that. Generally, if the result of a comparison is unordered, something is seriously wrong and you'll want to take corrective action.

One way to avoid this issue is to use the `fcmpe` form, which can generate an exception, and leave it up to the exception handler to deal with unordered values. However, as noted earlier, dealing with those exceptions is beyond the scope of this book, so I recommend sticking with `fcmp`.

The `fcmp` instruction sets the N, Z, V, and C flags in such a way that you can test them for ordered and unordered results after a comparison. The good news is that you can handle unordered and ordered comparisons by using normal conditional branch and other instructions. The bad news is that the `fcmp` results slightly change the definition of those conditional branch instructions. Table 6-12 describes how `fcmp` sets the flags.

Table 6-12: Flags Set by `fcmp`

Condition	Meaning
EQ	Equal
NE	Not equal, or unordered
GE	Greater than or equal
LT	Less than, or unordered
GT	Greater than
LE	Less than or equal, or unordered
HI	Greater than, or unordered
HS/CS	Greater than or equal, or unordered
LO/CC	Less than
LS	Less than or equal
MI	Less than
PL	Greater than or equal, or unordered
VS	Unordered
VC	Ordered

Two points in Table 6-12 are easy to miss:

- The `fcmp` instruction sets the V flag if the comparison is unordered.
- Both signed and unsigned tests are used for floating-point comparisons, which are intrinsically signed values.

You'll notice that GE and GT are ordered comparisons, while LE and LT handle unordered comparisons. Likewise, LS and LO are ordered comparisons, while HI and HS also handle unordered comparisons. At first glance, this might seem weird; why not make one set (signed or unsigned) ordered and the other set unordered?

However, you want the two opposite tests (for example, LE and GT, or LT and GE) to handle all possible outcomes. One of the outcomes is unordered. Therefore, one of the opposite comparisons needs to handle unordered so that the two tests in each pair provide total coverage of the conditionals (the same logic applies to HI-LS and HS-LO). You can always test the overflow flag (V) to see whether a comparison is ordered or unordered.

6.9.3.2 Conditional Comparisons

The conditional floating-point comparison instruction, `fccmp`, is the floating-point analog to the integer conditional comparison instruction. You can use it to reduce complex Boolean expressions involving conjunction (AND) and disjunction (OR), as noted earlier (see section 6.5, "Conditional Comparisons and Boolean Expressions," on page 314).

6.9.3.3 Comparison for Equality

As discussed in section 6.7, "Floating-Point and Finite-Precision Arithmetic," on page 322, you should be very careful about comparing two floating-point values (especially for equality). Minor inaccuracies in two calculations that would produce the same result using infinite-precision real arithmetic may yield different results when using limited-precision floating-point arithmetic. If you want to compare two values for equality, compute their difference and determine whether the absolute value of their difference is within an acceptable error range.

The real question is how to determine an acceptable range for the error. Because the difference between these (presumably equal) floating-point values will manifest itself in the LO bits of the mantissa, the error value should be something corresponding to a 1 bit in one of those positions.

Listing 6-2 demonstrates how to calculate this error value.

```
// Listing6-2.S
//
// Demonstrate comparing two floating-point
// values for equality by using a difference
// and error range comparison.

        #include    "aoaa.inc"

// The following bit mask will keep the
// exponent bits in a 64-bit double-precision
// floating-point value. It zeros out the
// remaining sign and mantissa bits.

❶ maskFP  =       0x7FF0000000000000

// bits is the number of bits you want to
// mask out at the bottom of the mantissa.
// It must be greater than 0:
```

```
❷ bits    =       4
  bitMask =       (1 << bits)-1

  // expPosn is the position of the first
  // exponent bit in the double-precision
  // format:

  expPosn =       52

          .text
          .pool
  ttlStr: wastr   "Listing 6-2"
  fmtStr: wastr   "error for (%24.16e) = %e\n"
  difMsg: wastr   "Difference:%e\n"
  values: wastr   "Value1=%23.16e, Value2=%23.16e\n"
  eqMsg:  wastr   "Value1 == Value2\n"
  neMsg:  wastr   "Value1 != Value2\n"

  // When value2 is somewhere between
  // 8e-323 and 9e-323, the
  // comparison becomes not equal:

  value1: .double 1.0e-323
  value2: .double 9e-323

  // Generic values to compare:

  // value1: .double   1.2345678901234567
  // value2: .double   1.234567890123456

  // getTitle
  //
  // Return pointer to program title
  // to the C++ code:

          proc    getTitle, public
          lea     x0, ttlStr
          ret
          endp    getTitle

  // computeError
  //
  // Given a double-precision floating-point
  // value in D0, this function computes an
  // error range value for use in comparisons.
  // If the difference between two FP values
  // (one of which is the value passed in D0)
  // is less than the error range value, you
  // can consider the two values equal.

        ❸ proc    computeError

          // Preserve all registers this code
          // modifies:

          locals  ce
```

```
            qword   ce.saveXO1
            byte    stack, 64
            endl    ce

            enter   ce.size
            stp     xO, x1, [fp, #ce.saveXO1]

            // Move the FP number into XO so you can mask
            // bits:

            fmov    xO, dO

            // Generate mask to extract exponent:

❹           and     xO, xO, #maskFP     // Extract exponent bits.
            lsr     x1, xO, #expPosn    // Put exponent in bits 0-10.

            // We need to normalize the value,
            // if possible:

❺           cmp     x1, #(expPosn - bits - 1)
            blo     willBeDenormal

            // If the result won't be a subnormal
            // (denormalized value), then set
            // the mantissa bits to all Os
            // (plus the implied 1 bit) and
            // decrement the exponent to move
            // the "bits" position up to the
            // implied bit:

❻           sub     x1, x1, #expPosn-bits // Adjust exponent.
            lsl     xO, x1, #expPosn      // Put exponent back.
            b.al    allDone

        // If the result will be denormalized, handle that
        // situation down here:

❼       willBeDenormal:
            mov     xO, #bitMask
            lsl     xO, xO, x1  // Shift as much as you can.

        allDone:
            fmov    dO, xO      // Return in DO.
            ldp     xO, x1, [fp, #ce.saveXO1]
            leave
            endp    computeError

        ////////////////////////////////////////////////////
        //
        // Here's the asmMain procedure:

            proc    asmMain, public

            locals  am
            double  am.error
```

```
        double  am.diff
        byte    am.stackSpace, 64
        endl    am

        enter   am.size

// Display the values you're going to compare:

        ldr     d0, value1
        str     d0, [sp]
        ldr     d1, value2
        str     d1, [sp, #8]
        lea     x0, values
        bl      printf

// Compute the error value:

        ldr     d0, value1
        bl      computeError
        str     d0, [fp, #am.error]

// Print the error value:

        str     d0, [sp, #8]
        ldr     d1, value1
        str     d1, [sp]
        lea     x0, fmtStr
        bl      printf

// Compute the difference of the
// two values you're going to compare
// and print that difference:

        ldr     d0, value1
        ldr     d1, value2
        fsub    d0, d0, d1
        str     d0, [fp, #am.diff]
        str     d0, [sp]
        lea     x0, difMsg
        bl      printf

// Compare the difference of the two
// numbers against the error range.

        ldr     d1, [fp, #am.error]
        ldr     d0, [fp, #am.diff]
        fabs    d0, d0              // Must be abs(diff)!
        fcmp    d0, d1
        ble     isEqual

// Print whether you should
// treat these values as equal:

        lea     x0, neMsg
        b.al    printIt
```

```
isEqual:
        lea     x0, eqMsg
printIt:
        bl      printf

        leave                       // Return to caller.
        endp    asmMain
```

The mask `0x7FF0_0000_0000_0000` ❶, when ANDed with a double-precision floating-point value, will strip out the mantissa and sign bits, leaving the exponent in bit positions 52 to 62 (11-bit exponent).

The `bits` constant ❷ in this listing determines the number of LO bits in the mantissa that the code will eliminate when generating the error value (this is currently 4 bits, so the 4 LO bits of the mantissa become insignificant, but in most cases it should be 2 to 3 bits for single-precision and 3 to 4 bits for double-precision comparisons). Once the `computeError` function generates the error value, the main program uses that error to compare a couple of floating-point numbers and report whether they should be treated as equal (their difference is less than the error value) or not equal (their difference is greater). The `bitMask` value is just a string of 1 bits (`4` in Listing 6-2).

The procedure `computeError` ❸ is passed a floating-point value in D0. This function computes an error value for that floating-point number such that if it is compared with a second number, their difference will be less than the error value if they should be considered equal. This function returns the error value in the D0 register.

To compute the error value, `computeError` begins by shifting the exponent down to bits 0 to 10 so that it is easier to work with ❹. If the exponent is less than 52 – 5 bits, the error value will turn out to be a subnormal (denormalized) number. The code determines whether the error value will be normalized or subnormal ❺.

If the result will be a normalized number, the code generates the error value by 52 bits (47 if `bits` is 4) and then shifts the exponent back into its proper location ❻. The mantissa and sign bits will all be 0; however, the implied bit for double-precision numbers will be 1, because the exponent is not 0.

If the error value will turn out to be subnormal, the code sets the exponent to 0, denoting a denormalized value, and shifts the `bitMask` value to the left the number of bit positions specified by the exponent minus the `bits` value ❼.

Here's the `build` command and sample output for Listing 6-2:

```
$ ./build Listing6-2
$ ./Listing6-2
Calling Listing6-2:
Value1 = 9.8813129168249309e-324, Value2 = 8.8931816251424378e-323
error for ( 7.4109846876186982e-323) = 9.881313e-324
Difference:-7.9050500e-323
Value1 != Value2
Listing6-2 terminated
```

This demonstrates that the difference between `Value1` and `Value2` is definitely outside the error range allowed for this comparison.

6.9.3.4 Conditional Select Instruction

Although the ARM does not support all the conditional instructions present in the integer instruction set, it does support the most often used conditional instruction: conditional select, or `fcsel`. The `fcsel` instruction has the following syntax:

```
fcsel Fd, Ft, Ff, cond
```

This instruction will test the condition and copy `Ft` to `Fd` if the condition is true, or it will copy `Ff` to `Fd` if the condition is false.

6.9.4 Floating-Point Conversion Instructions

The ARM instruction set includes a wide variety of instructions that convert between various floating-point formats and between signed/unsigned integers and floating-point formats. Certain CPUs even support conversions between floating-point and fixed-point formats. This section describes these conversions.

6.9.4.1 fcvt

The `fcvt` instruction converts between the three supported floating-point formats (half-, single-, and double-precision). This is one of the few instructions that supports the H*n* registers (`ldr` and `str` are the others). The syntax for this instruction is the following:

```
fcvt Hd, Ss
fcvt Hd, Ds
fcvt Sd, Hs
fcvt Sd, Ds
fcvt Dd, Hs
fcvt Dd, Ss
```

These instructions convert their source operand to the type of the destination operand and copy the converted data into that operand. Of course, not all conversions can happen without error—be aware that converting a larger-size format to a smaller-size format can produce underflow and underflow exceptions. You might want to consider checking the FPSR after such an operation:

```
fcvt s0, d1
mrs  x0, FPSR
mov  w1, #0x8c
ands w0, w0, w1  // UFC, OFC, and IDC bits
bne  badCvt
```

This code demonstrates checking the UFC, OFC, and IDC bits to see if an error occurred after the conversion.

6.9.4.2 Conversion Between Floating-Point and Integer

The instructions in Table 6-13 convert between various floating-point (single- and double-precision) and integer formats. The syntax for these instructions is as follows

```
fcvt{m}{s|u} Rd, Fn
```

where *m* is a, m, n, p, or z that specifies a rounding mode (see Table 6-13, where FP = floating-point, SI = signed integer, and UI = unsigned integer). F*n* represents any single- or double-precision floating-point register, and R*d* represents any general-purpose register (W*d* or X*d*).

Table 6-13: The fcvt{*m*}{s|u} Conversion Instructions

Instruction	Description
fcvtas	Convert FP to SI; round away from 0.
fcvtau	Convert FP to UI; round away from 0.
fcvtms	Convert FP to SI; round toward –infinity (floor function).
fcvtmu	Convert FP to UI; round toward –infinity (floor function).
fcvtns	Convert FP to SI; round to even (standard IEEE rounding).
fcvtnu	Convert FP to UI; round to even (standard IEEE rounding).
fcvtps	Convert FP to SI; round toward +infinity (ceil function).
fcvtpu	Convert FP to UI; round toward +infinity (ceil function).
fcvtzs	Convert FP to SI; round toward 0 (truncation).
fcvtzu	Convert FP to SI; round toward 0 (truncation).

In addition to converting floating-point values to integers, the ARM provides two instructions that convert integers to floating-point values:

```
scvtf Fd, Rd  // Same register meanings as for fcv*
ucvtf Fd, Rd  // instructions
```

The scvtf instruction converts a signed integer to a floating-point value, and the ucvtf instruction converts an unsigned integer to floating-point. Note that some integer values cannot be exactly represented by a single- or double-precision value. For example, a double-precision floating-point value has a 56-bit mantissa, so it cannot precisely represent all 64-bit integers.

6.9.4.3 Fixed-Point Conversions

Some 64-bit ARM CPUs support conversion between a fixed-point binary value and a floating-point value. These instructions take the following forms:

```
fcvtzs Rd, Fs, #bits
fcvtzu Rd, Fs, #bits
scvtf  Fd, Rs, #bits
ucvtf  Fd, Fs, #bits
```

Here, *bits* is the number of bits to the right of the binary point in the general-purpose register. It is a constant from 0 to one less than the size of the general-purpose register. For example, in a 64-bit register, a value of 32 would provide you with 32 bits to the left and right of the binary point in the fixed-point number.

6.9.4.4 Rounding

The ARM provides several floating-point rounding instructions. They are similar in nature to the floating-point-to-integer conversion insofar as they round a real number to an integral value. However, these instructions produce not binary integer values but rather floating-point results that just happen to be integer numbers (or, rather, the floating-point representation of those integer numbers).

These instructions all take a pair of floating-point registers as operands. Both registers must be the same size (single- or double-precision). The generic syntax is as follows:

```
frint{m} Fd, Fs  // Both registers must be Sn or Dn.
```

The instruction descriptions appear in Table 6-14.

Table 6-14: The `frint{m}` Instructions

Instruction	Description
`frinta`	Round away from 0.
`frinti`	Round using the Rmode setting in the FPCR.
`frintm`	Round toward –infinity.
`frintn`	Normal rounding, exactly 0.5 rounds to nearest even value.
`frintp`	Round toward +infinity.
`frintx`	Round using FPCR mode; raise an exception if value was not originally an integer.
`frintz`	Round toward 0.

Now that you've reviewed the floating-point conversion instructions, I'll show you how to use floating-point instructions in code that interfaces with other programs.

6.10 The ARM ABI and Floating-Point Registers

The ARM ABI considers V0 through V7 and V16 through V31 to be volatile. The caller must preserve these registers across procedure calls if it requires that they retain their values across a call.

Registers V8 through V15 are nonvolatile. A callee must preserve these registers within a procedure if it modifies their values. Of course, the advantage of these registers is that once a procedure preserves them (for its caller), it does not have to worry about modification to these registers by any functions it calls.

Callers pass the first eight floating-point parameters in registers to a procedure. When passing a combination of integer and floating-point parameters, the caller passes the non-floating-point parameters in the general-purpose registers (X0 to X7) and the floating-point arguments in the floating-point registers. If the number of floating-point parameters exceeds eight, the caller passes the floating-point parameters on the stack.

Parameters are assigned the next available register, not a register number based on the parameter's position in the parameter list. Consider the following C function prototype:

```
void p
(
    int i,
    double d,
    int j,
    int k,
    double e,
    int l,
    double f,
    double g,
    double h
);
```

The ARM ABI would associate the registers in Table 6-15 with these formal parameters:

Table 6-15: Parameter Assignments to Registers

Register	Parameter
X0	`i`
D0	`d`
X1	`j`
X2	`k`
D1	`e`
X3	`l`
D2	`f`
D3	`g`
D4	`h`

If a function passes a floating-point parameter by reference, the address of that floating-point value is passed in the next available general-purpose register (no floating-point registers for pass-by-reference parameters).

If a function returns a floating-point result, it returns that value in D0 (or S0, if the language supports returning single-precision floats as function return results). See Chapter 11 for details on returning vectors (multiple floating-point values) as function results (hint: V0). If a function returns an array of floating-point values, the caller must allocate storage for that array and pass a pointer to that array in X8. The function will store the results into that storage before returning.

6.11 Using C Standard Library Math Functions

Although the ARM instruction set provides a set of machine instructions that compute basic arithmetic operations, it does not have instructions for computing complex mathematical functions such as sine, cosine, and tangent. You could (with the appropriate knowledge) write these functions yourself in assembly language, but a much simpler solution is available: call functions that are already written for you. In particular, the C stdlib contains many useful mathematical functions you can use. This section describes how to call several of them.

As a sample program that demonstrates passing floating-point values to functions, Listing 6-3 makes calls to various C stdlib `<math.h>` functions (specifically `sin()`, `cos()`, and `tan()`). Each of these functions accepts a double-precision parameter and returns a double-precision result.

```
// Listing6-3.S
//
// Demonstrates calling various C stdlib
// math functions

#include    "aoaa.inc"

        .text
        .extern sin  // C stdlib functions
        .extern cos  // this program calls
        .extern tan

        .pool
ttlStr: wastr   "Listing 6-3"

// Format strings for each of the outputs:

piStr:  wastr   "%s(pi) = %20.14e\n"
pi2Str: wastr   "%s(pi/2) = %20.14e\n"
pi4Str: wastr   "%s(pi/4) = %20.14e\n"
pi8Str: wastr   "%s(pi/8) = %20.14e\n\n"

// Function names (printed as %s argument
// in the format strings):

sinStr: wastr   "sin"
```

```
cosStr: wastr   "cos"
tanStr: wastr   "tan"

// Sample values to print for each
// of the functions:

pi:     .double 3.141592653588979
pi2:    .double 1.5707963267949
pi4:    .double 0.7853981639745
pi8:    .double 0.39269908169872

// getTitle
//
// Return pointer to program title
// to the C++ code.

        proc    getTitle, public
        lea     x0, ttlStr
        ret
        endp    getTitle


// Trampolines to the C stdlib math functions.
// These are necessary because lea can't take
// the address of a function that could be
// very far away (as the dynamic libraries
// probably are).
//
// Note: Must use real "b" instruction here
// rather than "b.al" because external
// functions are likely out of range.

      ❶ proc    sinVeneer
        b       sin
        endp    sinVeneer

        proc    cosVeneer
        b       cos
        endp    cosVeneer

        proc    tanVeneer
        b       tan
        endp    tanVeneer

// doPi( char *X0, func X1 )
//
// X0-  Contains the address of a function
//      that accepts a single double and
//      returns a double result.
// X1-  Contains the address of a string
//      specifying the function name.
//
// This function calls the specified function
```

```
// passing PI divided by 1, 2, 4, and 8 and
// then prints the result that comes back.

❷ proc    doPi

  locals  dp
  dword   dp.saveX1
  dword   dp.saveX0
  dword   dp.saveX19
  byte    dp.stackSpace, 64
  endl    dp

  // Set up activation record and save register values:

  enter   dp.size
  stp     x0, x1, [fp, #dp.saveX0]  // X1 -> saveX1, too
  str     x19, [fp, #dp.saveX19]    // Preserve nonvolatile.

  mov     x19, x0                   // Keep address in nonvolatile.

  // Call the function for various values
  // of pi/n:

❸ ldr     d0, pi
  blr     x19           // Call function.
  mstr    d0, [sp, #8]  // Save func result as parm.
  ldr     x1, [fp, #dp.saveX1]
  mstr    x1, [sp]
  lea     x0, piStr
  bl      printf

  ldr     d0, pi2
  blr     x19             // Call function.
  mstr    d0, [sp, #8]
  lea     x0, piStr
  ldr     x1, [fp, #dp.saveX1]
  mstr    x1, [sp]
  lea     x0, pi2Str
  bl      printf

  ldr     d0, pi4
  blr     x19             // Call function.
  mstr    d0, [sp, #8]
  lea     x0, piStr
  ldr     x1, [fp, #dp.saveX1]
  mstr    x1, [sp]
  lea     x0, pi4Str
  bl      printf

  ldr     d0, pi8
  blr     x19             // Call function.
  mstr    d0, [sp, #8]
  lea     x0, piStr
  ldr     x1, [fp, #dp.saveX1]
```

```
        mstr    x1, [sp]
        lea     x0, pi8Str
        bl      printf

        // Restore nonvolatile register
        // and return:

        ldr     x19, [fp, #dp.saveX19]
        leave
        endp    doPi

////////////////////////////////////////////////////
//
// Here's the asmMain procedure:

        proc    asmMain, public
        enter   64              // Generic entry

        // Load X0 with the address
        // of the veneer (trampoline) function
        // that calls the C stdlib math function,
        // load X1 with the function's name,
        // then call doPi to call the function
        // and print the results:

    ❹ lea     x0, sinVeneer   // SIN(x) output
      lea     x1, sinStr
      bl      doPi

      lea     x0, cosVeneer   // COS(x) output
      lea     x1, cosStr
      bl      doPi

      lea     x0, tanVeneer   // TAN(x) output
      lea     x1, tanStr
      bl      doPi

      leave                   // Return to C/C++ code.
      endp    asmMain
```

This program calls the `sin()`, `cos()`, and `tan()` functions indirectly—the address of the particular function is passed as a parameter to the `doPi` procedure. Unfortunately, macOS's PIE functionality prevents you from taking the address of such a function by using the `lea` macro, because there is no telling where the OS will load the dynamically linked (shared) library at runtime; it could be farther away than the ±4GB allowed by `lea`. Therefore, this code creates trampolines for these functions that the OS can patch to transfer control to wherever the functions are sitting in memory ❶. These trampolines are necessary only for macOS; though they will work with Linux code, Linux allows you to take the address of the C stdlib functions with `lea`.

The doPi function ❷ saves the values of X0, X1, and X19 in the activation record. Preserving X19 is necessary because this is a nonvolatile register. Saving X0 and X1 is necessary because the procedure needs their values across calls to printf(), and these registers are volatile.

The body of the doPi calls the appropriate function (sin(), cos(), or tan()) four times with the values π, π/2, π/4, and π/8, and it then displays the result these functions return ❸. Note how doPi calls the function indirectly by using the blr instruction—the address of the function was originally passed to doPi in the X0 register.

The main procedure loads the address of the trampoline (veneer) function into X0, along with a string pointer, and calls doPi to compute the values and print the results ❹. (Trampolines and veneers are explained further in Chapter 7.) Loading the address of the trampoline functions into X0 is necessary only under macOS; with Linux, you can load the address of the sin(), cos(), or tan() function directly and spare the minor inefficiency of having to jump through the trampoline function.

Here's the build command and sample output for Listing 6-3:

```
$ ./build -math Listing6-3
$ ./Listing6-3
Calling Listing6-3:
sin(pi) = 8.14137986335080e-13
sin(pi/2) = 1.00000000000000e+00
sin(pi/4) = 7.07106781594585e-01
sin(pi/8) = 3.82683432365086e-01

cos(pi) = -1.00000000000000e+00
cos(pi/2) = -3.49148133884313e-15
cos(pi/4) = 7.07106780778510e-01
cos(pi/8) = 9.23879532511288e-01

tan(pi) = -8.14137986335080e-13
tan(pi/2) = -2.86411383293069e+14
tan(pi/4) = 1.00000000115410e+00
tan(pi/8) = 4.14213562373090e-01

Listing6-3 terminated
```

You'll notice one difference between this build command and most of the others in the book: the -math argument. This tells Linux to link in the C stdlib math library functions (macOS automatically links this in). Without the -math option, you'll get a linker error when you try to build the program.

The C stdlib contains many double-precision functions you might find useful. Check them out online for more details. Many of these functions are unnecessary in assembly language, as they correspond to one or two machine instructions. Nevertheless, the library contains complex functions that you wouldn't want to write yourself.

You may find various functions online that purport to be faster than those in the C stdlib. Be careful about using them because they tend to be

notoriously inaccurate. Unless you're well grounded in numerical analysis, don't try to write these functions yourself.

6.12 Moving On

This chapter covered a lot of material: the remaining arithmetic instructions (including multiplication, division, and remainder, as well as `cmp` and the various conditional instructions), maintaining variables in registers rather than memory locations, and the proper use of volatile and nonvolatile registers. It also discussed creating structures to provide efficient access to global variables, converting arithmetic and logical expressions (integer and floating-point) to their machine instruction equivalents, and calling functions written in C/C++.

Armed with this information, you can now convert arithmetic expressions in an HLL such as C/C++ to ARM assembly language. The only basic skill missing from your programming repertoire is a good understanding of control structures in assembly language, which you'll learn in the next chapter.

6.13 For More Information

- My book *Write Great Code*, Volume 1 (No Starch Press, 2020), includes sections on the cache and thrashing.
- Reference Wikipedia for details on fixed-point arithmetic: *https://en.wikipedia.org/wiki/Fixed-point_arithmetic*.
- You can learn more about limited-precision arithmetic from the following resources:
  - A Central Connecticut State University tutorial in the form of an interactive questionnaire: *https://chortle.ccsu.edu/assemblytutorial/Chapter-29/ass29_10.html*.
  - Python documentation on the topic: *https://docs.python.org/3/tutorial/floatingpoint.html*.
- For more information on writing better code using floating-point arithmetic, see the following post on the Society of Actuaries website: *https://www.soa.org/news-and-publications/newsletters/compact/2014/may/com-2014-iss51/losing-my-precision-tips-for-handling-tricky-floating-point-arithmetic*.
- Wikipedia documents the C stdlib math functions at *https://en.wikipedia.org/wiki/C_mathematical_functions*.
- If you insist on writing your own transcendental functions, you might try to locate a copy of the following book (long out of print), the "bible" of transcendental functions: *Computer Approximations,* by John F. Hart, E.W. Cheney, and Charles L. Lawson (Krieger Publishing, 1978).

TEST YOURSELF

1. How does the `cmp` instruction affect the zero flag?
2. How does the `cmp` instruction affect the carry flag, with respect to an unsigned comparison?
3. How does the `cmp` instruction affect the negative and overflow flags, with respect to a signed comparison?
4. Convert the following expressions to assembly language (assume all variables are signed 32-bit integers):

```
x = x + y
x = y - z
x = y * z
x = y + z * t
x = (y + z) * t
x = -((x * y) / z)
x = (y == z) && (t != 0)
```

5. Compute the following expressions without using a `mul` instruction (assume all variables are signed 32-bit integers):

```
x = x * 2
x = y * 5
x = y * 8
```

6. Compute the following expressions without using a `udiv` or `sdiv` instruction (assume all variables are unsigned 64-bit integers):

```
x = x / 2
x = y / 8
x = z / 10
```

7. Convert the following expressions to assembly language (assume all variables are double-precision floating-point values):

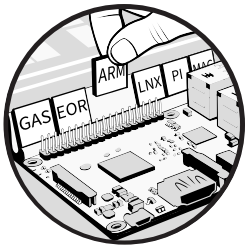
```
x = x + y
x = y - z
x = y * z
x = y + z * t
x = (y + z) * t
x = -((x * y) / z)
```

8. Convert the following expressions to assembly language by using floating-point instructions. Assume `bb` is a 1-byte Boolean variable and `x`, `y`, and `z` are `.double` floating-point variables:

```
bb = x < y
bb = x >= y && x < z
```

7

LOW-LEVEL CONTROL STRUCTURES



The examples in this book up to this point have created assembly control structures in an ad hoc manner. Now it's time to formalize how to control the operation of your assembly language programs. By the time you finish this chapter, you should be able to convert HLL control structures into assembly language control statements.

Control structures in assembly language consist of conditional branches and indirect jumps. This chapter discusses those instructions and how to emulate HLL control structures such as `if...else`, `switch`, and loop statements. This chapter also discusses labels, the targets of conditional branches and jump statements, as well as the scope of labels in an assembly language source file.

7.1 Statement Labels

Before discussing the jump instructions and how to use them to emulate control structures, an in-depth discussion of assembly language statement labels is necessary. *Labels* in an assembly language program stand in as symbolic names for addresses. Referring to a position in your code by using a name such as `LoopEntry` is far more convenient than using a numeric address such as 0xAF1C002345B7901E. For this reason, assembly language low-level control structures make extensive use of labels within source code (see section 2.10, "Control-Transfer Instructions," on page 74).

You can do three operations on code labels: transfer control to a label via a conditional or unconditional jump instruction, call a label via the `bl` instruction, and take the address of a label. The last of these is useful when you want to indirectly transfer control to that address at a later point in your program.

The following code sequence demonstrates how to take the address of a label in your program by using the `lea` macro:

```
stmtLbl:
   .
   .
   .
  lea x0, stmtLbl
   .
   .
   .
stmtLbl2:
```

Because addresses are 64-bit quantities, you'll typically load an address into a 64-bit general-purpose register by using the `lea` instruction. Also see section 7.5, "Taking the Address of Symbols in Your Code," on page 364 for more information about taking the address of a label in your programs.

7.2 Initializing Arrays with Statement Labels

Gas allows you to initialize double-word objects with the addresses of statement labels. The code fragment in Listing 7-1 demonstrates how to do this.

```
// Listing7-1.S
//
// Initializing qword values with the
// addresses of statement labels

#include "aoaa.inc"

            .data
            .align   3    // Align on dword boundary.
lblsInProc: .dword   globalLbl1, globalLbl2  // From procWLabels

            .code
```

```
// procWLabels
//
// Just a procedure containing private (lexically scoped)
// and global symbols. This really isn't an executable
// procedure.

            proc    procWLabels

globalLbl1:  b.al    globalLbl2
globalLbl2:
            ret
            endp    procWLabels

            .pool
            .align  3   // dword align
dataInCode:  .dword  globalLbl2, globalLbl1
```

You might recall that pointers in the `.text` section cannot refer to objects outside that section; however, it is perfectly legitimate for pointers in other sections (such as `.data`) to refer to symbols in the `.text` section.

As addresses on the ARM are 64-bit quantities, you will typically use the `.dword` directive, as in the previous examples, to initialize a data object with the address of a statement label.

7.3 Unconditional Transfer of Control

The `b.al` (branch) instruction unconditionally transfers control to another point in the program. This instruction has three forms: two PC-relative branches and an indirect jump. These instructions take the following forms:

```
b    label   // Range is ±128MB.
b.al label   // Range is ±1MB.
br   reg64
```

The first two instructions are *PC-relative branches*, which you've seen in various sample programs up to this point. For PC-relative branches, you normally specify the target address by using a statement label. The label appears either on the same line as an executable machine instruction or by itself on a line preceding it. The direct jump is completely equivalent to a `goto` statement in an HLL.

Here's an example of a direct jump that transfers control to a label elsewhere in the program:

```
          statements
          b laterInPgm   // Or b.al laterInPgm
              .
              .
              .
laterInPgm:
          statements
```

Unlike HLLs, for which your instructors usually forbid you to use `goto` statements, you will find that the use of the `b/b.al` instruction in assembly language is essential.

7.4 Register-Indirect Branches

The third form of the `br` reg_{64} branch instruction given earlier is a *register-indirect jump* instruction that transfers control to the instruction whose address appears in the specified 64-bit general-purpose register. To use the `br` instruction, you must load a 64-bit register with the address of a machine instruction prior to the execution of `br`. When several paths, each loading the register with a different address, converge on the same `br` instruction, control transfers to an appropriate location determined by the path up to that point.

Listing 7-2 reads a string of characters from the user that contain an integer value. It uses `strtol()` to convert that string to a binary integer value. This C stdlib function doesn't do the best job of reporting an error, so this program tests the return results to verify a correct input and uses register-indirect jumps to transfer control to different code paths based on the result.

The first part of Listing 7-2 contains constants, variables, external declarations, and the (usual) `getTitle()` function.

```
// Listing7-2.S
//
// Demonstrate indirect jumps to
// control flow through a program.

#include    "aoaa.inc"

maxLen      =       256
EINVAL      =       22      // "Magic" C stdlib constant, invalid argument
ERANGE      =       34      // Value out of range


            .data
buffer:     .fill   256, 0 // Input buffer

            .text
            .pool
ttlStr:     wastr   "Listing 7-2"

fmtStrA:    wastr   "value=%d, error=%d\n"

fmtStr1:    .ascii  "Enter an integer value between "
            wastr   "1 and 10 (0 to quit): "

badInpStr:  .ascii  "There was an error in readLine "
            wastr   "(ctrl-D pressed?)\n"

invalidStr: wastr   "The input string was not a proper number\n"
```

```
rangeStr:   .ascii  "The input value was outside the "
            wastr   "range 1-10\n"

unknownStr: .ascii  "The was a problem with strToInt "
            wastr   "(unknown error)\n"

goodStr:    wastr   "The input value was %d\n"

fmtStr:     wastr   "result:%d, errno:%d\n"


// getTitle
//
// Return pointer to program title
// to the C++ code.

        proc    getTitle, public
        lea     x0, ttlStr
        ret
        endp    getTitle
```

The next section of Listing 7-2 is the `strToInt` function, a wrapper around the C stdlib `strtol()` function that does a more thorough job of handling erroneous inputs from the user. See the comments for the function's return values:

```
// Listing7-2.S (cont.)
//
// strToInt
//
// Converts a string to an integer, checking for errors
//
// Argument:
//    X0-   Pointer to string containing (only) decimal
//              digits to convert to an integer
//
// Returns:
//    X0-   Integer value if conversion was successful.
//    X1-   Conversion state. One of the following:
//          0- Conversion successful
//          1- Illegal characters at the beginning of the
//                 string (or empty string)
//          2- Illegal characters at the end of the string
//          3- Value too large for 32-bit signed integer


            proc    strToInt

            locals  sti
            dword   sti.saveX19
            dword   sti.endPtr
            word    sti.value
            byte    sti.stackSpace, 64
```

```
            endl    sti

            enter   sti.size

            mov     x19, x0         // Save, so you can test later.

// X0 already contains string parameter for strtol,
// X1 needs the address of the string to convert, and
// X2 needs the base of the conversion (10).

          ❶ add     x1, fp, #sti.endPtr
            mov     x2, #10             // Decimal conversion
            bl      strtol

// On return:
//
//    X0-    Contains converted value, if successful
//    endPtr-Pointer to 1 position beyond last char in string
//
// If strtol returns with endPtr == strToConv, then there were no
// legal digits at the beginning of the string.

            mov     x1, #1                // Assume bad conversion.
            ldr     x2, [fp, #sti.endPtr] // Is startPtr = endPtr?
            cmp     x19, x2
            beq     returnValue

// If endPtr is not pointing at a 0 byte, you have
// junk at the end of the string.

            mov     x1, #2        // Assume junk at end.
            ldrb    w3, [x2]      // Byte at endPtr.
            cmp     x3, #0        // Is it zero?
            bne     returnValue   // Return error if not 0.

// If the return result is 0x7fff_ffff or 0x8000_0000
// (max long and min long, respectively), and the C
// global _errno variable contains ERANGE, you have
// a range error.

            str     w0, [fp, #sti.value] // Get C errno value.
          ❷ getErrno                     // Magic macro
            mov     x2, x0
            ldr     w0, [fp, #sti.value]

            mov     x1, 0         // Assume good input.
            cmp     w2, #ERANGE   // errno = out of range?
            bne     returnValue
            mov     x1, #3        // Assume out of range.

            mov     x2, 0xffff
            movk    x2, 0x7fff, lsl #16

            cmp     w0, w2
            beq     returnValue
```

```
            mvn     w2, w2          // W2 = 0x8000_0000
            cmp     w0, w2
            beq     returnValue

// If you get to this point, it's a good number.

            mov     x0, #0

returnValue:
            leave
            endp    strToInt
```

The `strtol()` ❶ function expects a pointer to an end-of-string pointer variable. The `strToInt` procedure reserved space for this pointer in the activation record. This code computes the address of that pointer variable to pass on to the `strtol()` function.

Retrieving the C `errno` variable ❷ is done differently in macOS and Linux (or, more likely, in Clang versus GCC). The `getErrno` macro in the *aoaa.inc* include file generates the appropriate code for the two systems. It returns `errno` in X0.

The final section of Listing 7-2 is the main program and the most interesting part of the code, because it demonstrates how to call the `strToInt` function:

```
// Listing7-2.S (cont.)
//
// Here's the asmMain procedure:

            proc    asmMain, public

            locals  am
            dword   am.saveX19              // Nonvolatile
            byte    am.stackSpace, 64
            endl    am

            enter   am.size
            str     x19, [fp, #am.saveX19]  // Must preserve X19.

// Prompt the user to enter a value
// from 1 to 10:

repeatPgm:  lea     x0, fmtStr1
            bl      printf

// Get user input:

            lea     x0, buffer
            mov     x1, #maxLen
            bl      readLine

            lea     x19, badInput // Initialize state machine.
          ❶ ands    w0, w0, w0    // X0 is -1 on bad input.
```

```
            bmi     hadError        // Only neg value readLine returns.

// Call strtoint to convert string to an integer and
// check for errors:

            lea     x0, buffer      // Ptr to string to convert
            bl      strToInt
            lea     x19, invalid
            cmp     w1, #1
            beq     hadError
            cmp     w1, #2
            beq     hadError

            lea     x19, range
            cmp     w1, #3
            beq     hadError

            lea     x19, unknown
            cmp     w1, #0
            bne     hadError


// At this point, input is valid and is sitting in X0.
//
// First, check to see if the user entered 0 (to quit
// the program):

          ❷ ands    x0, x0, x0      // Test for zero.
            beq     allDone

// However, we need to verify that the number is in the
// range 1-10:

            lea     x19, range
            cmp     x0, #1
            blt     hadError
            cmp     x0, #10
            bgt     hadError

// Pretend a bunch of work happens here dealing with the
// input number:

            lea     x19, goodInput

// The different code streams all merge together here to
// execute some common code (for brevity, we'll pretend that happens;
// no such code exists here):

hadError:

// At the end of the common code (which mustn't mess with
// X19), separate into five code streams based
// on the pointer value in X19:

          ❸ br      x19
```

```
// Transfer here if readLine returned an error:

badInput:   lea     x0, badInpStr
            bl      printf
            b.al    allDone

// Transfer here if there was a nondigit character
// in the string:

invalid:    lea     x0, invalidStr
            bl      printf
            b.al    repeatPgm

// Transfer here if the input value was out of range:

range:      lea     x0, rangeStr
            bl      printf
            b.al    repeatPgm

// Shouldn't ever get here. Happens if strToInt returns
// a value outside the range 0-3:

unknown:    lea     x0, unknownStr
            bl      printf
            b.al    repeatPgm

// Transfer down here on a good user input:

goodInput:  mov     w1, w0
            lea     x0, goodStr
            mstr    w1, [sp]
            bl      printf
            b.al    repeatPgm

// Branch here when the user selects "quit program" by
// entering the value 0:

allDone:    ldr     x19, [fp, #am.saveX19] // Must restore before returning.
            leave

            endp    asmMain
```

The main function loads the X19 register with the address of code to execute based on the `strToInt` return results. The `strToInt` function returns one of the following states (see the comments in the previous code for an explanation):

- Valid input
- Illegal characters at the beginning of the string
- Illegal characters at the end of the string
- Range error

The program then transfers control to different sections of `asmMain` based on the value held in X19, which specifies the type of result `strToInt` returns.

The `readline` function returns –1 ❶ if there was an error reading the line of text from the user, which typically occurs when the end of the file is detected. This is the *only* negative value that `readline` returns, so rather than test for –1, this code just checks to see if `readline` returned a negative value. The test is a little sneaky, but it's a standard trick; anytime you AND a value with itself, you get the original value back. In this case, the code uses the `ands` instruction, which also sets the Z flag if the value was 0 and sets the N flag if the number was negative ❷. Therefore, testing the N flag afterward checks for an error condition. Note that a `cmp x0, #0` instruction would serve this same purpose.

Once again, this code uses the `ands` instruction ❷ to compare the result against 0. This time, it's actually checking for the value 0 (via the Z flag) by using the `beq` instruction immediately afterward. This is where the program in Listing 7-2 demonstrates using the `br` (branch indirect through register) instruction to implement the logic ❸.

Here's the `build` command and a sample run of Listing 7-2:

```
$ ./build Listing7-2
$ ./Listing7-2
Calling Listing7-2:
Enter an integer value between 1 and 10 (0 to quit): a123
The input string was not a proper number
Enter an integer value between 1 and 10 (0 to quit): 123a
The input string was not a proper number
Enter an integer value between 1 and 10 (0 to quit): 1234567890123
The input value was outside the range 1-10
Enter an integer value between 1 and 10 (0 to quit): -1
The input value was outside the range 1-10
Enter an integer value between 1 and 10 (0 to quit): 11
The input value was outside the range 1-10
Enter an integer value between 1 and 10 (0 to quit): 5
The input value was 5
Enter an integer value between 1 and 10 (0 to quit): 0
Listing7-2 terminated
```

This sample run demonstrates several bad inputs, including non-numeric inputs, out-of-range values, a legitimate value, and entering `0` to exit the program.

7.5 Taking the Address of Symbols in Your Code

Listing 7-2 computed the address of various symbols throughout the `.text` section in order to load those addresses into a register for later use. Obtaining the runtime address of a symbol in the program is a common operation in assembly language programs, because this is how you access data (and code) indirectly via a register.

This chapter covers control structures, and this section discusses obtaining addresses of statement labels in the program. Much of the information in this section is review material from earlier chapters in this book, but I've pulled it together here for reference purposes and extended the discussion.

7.5.1 Revisiting the lea Macro

Listing 7-2 used the `lea` macro to initialize 64-bit registers with the address of a location to jump to via the `br` instruction. This has been the go-to macro for obtaining the address of a symbol throughout this book. However, remember that `lea` is a macro and that

```
lea x0, symbol
```

translates into this:

```
// Under macOS:

    adrp x0, symbol@PAGE
    add  x0, x0, symbol@PAGEOFF

// Under Linux:

    adrp x0, symbol
    add  x0, x0, :lo12:symbol
```

The two-instruction sequence allows the `lea` macro to compute the address of a PC-relative symbol anywhere in a ±4GB range. The `adr` instruction can also compute the address of a symbol but it supports only a ±1MB range (see section 1.8.1, "`ldr`, `str`, `adr`, and `adrp`," on page 23).

When taking the address of nearby statement labels in the `.text` section, it is going to be more efficient to use the `adr` instruction:

```
adr x0, symbol
```

The only time this will fail is if your `.text` section is very large and the symbol is more than 1MB away from the `adr` instruction. The main reason for using the `lea` macro is to obtain the address of a symbol that is in a different section (especially on macOS, whose PIE/ASLR policy will likely locate that section farther than ±1MB away).

If the symbol/memory location whose address you wish to compute is farther than ±4GB away from the current instruction, you'll have to use one of the approaches in the following sections to obtain its address.

7.5.2 Statically Computing the Address of a Symbol

Since memory addresses are 64 bits, and the `.dword` directive allows you to initialize a dword object with a 64-bit value, shouldn't it be possible to initialize such an object with the 64-bit address of another symbol in the program? The answer depends on the OS you're running under.

Under Linux, it is perfectly legal (even when running PIE code) to do the following

```
varPtr:  .dword variable
```

where *variable* is the name of a symbol appearing in a .data, .bss, .rodata, or .text section. When Linux loads the executable program into memory, it will automatically patch this dword memory location with the address of that symbol in memory (wherever Linux has loaded it). Depending on the section, you might be able to directly load the contents of this location in the X0 register by using the following instruction, assuming that the symbol is within the PC-relative range of the ldr instruction:

```
ldr x0, varPtr
```

Sadly, this scheme may not work under macOS, where you're not allowed to use absolute addresses in your .text section. If you move varPtr to the .data section, macOS will accept the pointer initialization but will reject the ldr instruction with the same complaint about an illegal absolute address. Of course, you could use the lea macro to load the address of varPtr into X0 and then fetch *variable*'s address by using the [X0] addressing mode; however, at that point, you may as well use the lea instruction to load the address of *variable* directly into X0. In any case, you're back to the ±4GB limitation of the lea macro.

You can get around the absolute address limitation of macOS by using a relative address rather than an absolute address. A *relative address* is just an offset from a fixed point in memory (for example, a PC-relative address is an offset from the address held in the PC register). You can create a self-relative 64-bit pointer by using the following statement:

```
varPtr:  .dword variable-.  // "." is same as "varPtr" here.
```

This initializes this 64-bit memory location with the distance (in bytes) from the varPtr object to the desired memory location (*variable*). This is known as a *self-relative pointer* because the offset is from the pointer variable itself. As it turns out, macOS's assembler is perfectly happy with this address expression (even in the .text section) because it is not an absolute address.

MACOS AND INITIALIZED POINTERS

MacOS does not allow 64-bit absolute addresses within the .text section. They can't point into the .text section or at other sections. No absolute addresses, absolutely.

This restriction does not exist in other sections. You can have initialized 64-bit pointers in a .data section or in an .rodata section. Those pointers can

even point at addresses within the .text section. I don't know why absolute pointers are allowed in these other sections but not the .text section, but I suspect that an exploit took advantage of a pointer in the .text section that doesn't work if the pointer is in other sections.

Of course, you cannot simply load these 64 bits into a register and address the memory location at which they point. The value is an offset, not an address. However, if you add the address of varPtr to its contents, this will give you the address of *variable*, as demonstrated in the following code:

```
adr x0, varPtr   // Assume varPtr is in .text and nearby.
ldr x1, varPtr   // Get varPtr address and contents, then
add x0, x0, x1   // add them together for variable's address.
```

This sequence solves the problem with addresses under macOS and happens to work just fine under Linux as well. Because this sequence will work under both OSes, this book adopts this scheme when fetching addresses from variables in memory.

Under macOS, this sequence requires varPtr to be in the same .text section as the instructions. Otherwise, macOS will complain that varPtr is an absolute address and will reject this code. Because I've written this book assuming the code will generally work under Linux and macOS, I will keep such labels in the .text section.

A single ldr instruction will also work fine under Linux, so if you're writing Linux-only code, the single ldr is more efficient.

7.5.3 Dynamically Computing the Address of a Memory Object

Computing the address of a nonstatic memory object is a bit more involved than doing the same for static (.data, .bss, .text, .rodata, and so on) memory objects.

Because every ARM machine instruction is exactly 32 bits in length, you can view a .text section containing nothing but machine instructions as an array of words, where the value in each word just happens to be the encoding of a machine instruction. (This view isn't 100 percent accurate; if the .text section contains data as well as instructions, there are limitations to how far you can go with treating the .text section as an array of instructions. However, if you limit yourself to those areas that contain only instructions, everything will be fine.)

With this in mind, it is possible to manipulate the values in the .text section by using the techniques for arrays from section 4.7, "Arrays," on page 194. This includes techniques such as indexing into arrays and computing the effective address of array elements.

NOTE *Section 5.6.2, "Passing by Reference," on page 256 describes a procedure for computing the effective address of an object you reference via the ARM's various addressing modes. For data objects, see that discussion.*

Consider the following instruction sequence of arbitrary instructions cut from Listing 7-2:

```
goodInput:  mov     w1, w0
            lea     x0, goodStr
            mstr    w1, [sp]
            bl      printf
            b.al    repeatPgm
```

The label `goodInput` is the base address of an array of five words containing the five instructions in this short sequence. You can, of course, take the base address of this array by using the `adr` instruction (or `lea` if this sequence is too far away). Once you have this base address in a register (such as X0), you can use the array-indexing calculation to compute the address of a particular entry in the array:

```
element_address = base_address + index × element_size
```

The *`element_size`* value is 4, as each instruction is 32 bits. Index 0 specifies the `mov` instruction, index 1 specifies the `adr` instruction, and so on.

Given an index value in X1, you can transfer control directly to one of these five instructions by using the following code:

```
adr x0, goodInput
add x0, x0, x1, lsl #2
br  x0
```

The `add` instruction multiplies the index (X1) by 4 before adding it to the base address. This computes the byte address of the specified instruction in the sequence; then `br` transfers control to the instruction.

In many respects, this is similar to a `switch` or `case` statement, where a unique case is associated with each instruction in the sequence. This chapter considers such control structures in section 7.6.7, "`switch...case` Statements," on page 389. In the meantime, just know that you can dynamically compute the address of one of the instructions in this sequence by using normal effective address calculations.

7.5.4 Working with Veneers

In the rare case you need to branch to a location beyond the range of the conditional branch instructions, you can use an instruction sequence such as the following

```
        bcc_opposite  skipJmp
        lea           x16, destLbl
```

```
        br      x16
skipJmp:
```

where `bcc`$_{opposite}$ is the opposite of the branch you want to take. This opposite branch skips over the code that transfers control to the target location. This provides you with the 4GB range of the `lea` macro, which should be sufficient if you're branching to code in your program. The opposite conditional branch transfers control to the normal *fall-though point* in the code (the code you'd normally fall through to if the condition is false). If the condition is true, control transfers to a memory-indirect jump that jumps to the original target location via a 64-bit pointer.

This sequence is known as a *veneer* (or a *trampoline*), because a program jumps to this point to move even further in the program—much like jumping on a trampoline lets you jump higher and higher. Veneers are useful for call and unconditional jump instructions that use the PC-relative addressing mode (and thus are limited to a ±1MB range around the current instruction). You'll rarely use veneers to transfer to another location within your program, since it's unlikely you'll write assembly language programs that large.

Note the use of the X16 register in this example. The ARM ABI reserves registers X16 and X17 for dynamic linking and veneer use. You're free to use these two as volatile registers with the expectation that their contents may be changed upon executing a branch instruction (of any kind, though generally a `bl` instruction). Compilers and linkers will typically modify an out-of-range branch instruction to transfer code to a nearby veneer, which then transfers control the full distance to the actual destination. When creating your own veneers, it makes sense to use these registers as temporaries for that purpose.

Branching to code outside this range generally means you're transferring control to a function in a shared (or dynamically linked) library. See the appropriate documentation for your OS for details on such libraries.

Table 7-1 lists the opposite conditions; refer to Table 2-11 on page 82 for the available opposite branch macros in *aoaa.inc*.

Table 7-1: Opposite Conditions

Branch condition	Opposite
eq	ne
ne	eq
hi	ls
hs	lo
lo	hs
ls	hi
gt	le
ge	lt

(continued)

Table 7-1: Opposite Conditions *(continued)*

Branch condition	Opposite
lt	ge
le	gt
cs	cc
cc	cs
vs	vc
vc	vs
mi	pl
pl	mi

If the destination location is beyond the ±4GB range of the `lea` macro, you'll need to create a 4-byte pointer (offset) to the actual location and use code such as the following:

```
    adr     x16, destPtr
    ldr     x17, destPtr
    add     x16, x16, x17
    br      x16
destPtr:
    .dword  destination-.  // Same as "destination-destPtr"
```

This particular sequence is sufficiently useful that the *aoaa.inc* include file provides a macro that expands to it:

```
goto destination
```

If you need to call a procedure that's more than ±4GB away, you could emit similar code:

```
    adr     x16, destPtr
    ldr     x17, destPtr
    add     x16, x16, x17
    blr     x16
    b.al    skipAdrs
destPtr:
    .dword  destination-.
skipAdrs:
```

However, it's easier to do this:

```
    bl      veneer
     .
     .
     .
veneer:
    goto destination
```

With the discussion of veneers out of the way, the next section can discuss how to implement HLL-like control structures in assembly language.

7.6 Implementing Common Control Structures in Assembly Language

This section shows you how to implement HLL-like control structures such as decisions, loops, and other control constructs by using pure assembly language. It concludes by showing some ARM instructions designed for creating common loops.

Throughout many of the following examples, this chapter assumes that various variables are local variables in the activation record (indexed off of the FP register) or static/global variables indexed off the SB (X28) register. We presume that appropriate structure declarations have been made for all the variable's identifiers and that the FP/SB registers have been properly initialized to point at these structures.

7.6.1 Decisions

In its most basic form, a *decision* is a branch within the code that switches between two possible execution paths based on a certain condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the following `if...then...endif` statement in an HLL:

```
if(expression) then
    statements
endif;
```

To convert this to assembly language, you must write statements that evaluate the *expression* and then branch around the *statements* if the result is false. For example, if you had the C statements

```
if(aa == bb)
{
    printf("aa is equal to bb\n");
}
```

you could translate this to assembly as follows:

```
      ldr  w0, [fp, #aa]    // Assume aa and bb are 32-bit integers.
      ldr  w1, [fp, #bb]
      cmp  w0, w1
      bne  aNEb             // Use opposite branch to skip then
      lea  x0, aIsEqlBstr   // " aa is equal to bb\n".
      bl   printf
aNEb:
```

In general, conditional statements may be broken into three basic categories: `if` statements, `switch...case` statements, and indirect jumps. Next, you'll learn about these program structures, how to use them, and how to write them in assembly language.

7.6.2 if...then...else Sequences

The most common conditional statements are the `if...then...endif` and `if...then...else...endif` statements. These two statements take the form shown in Figure 7-1.

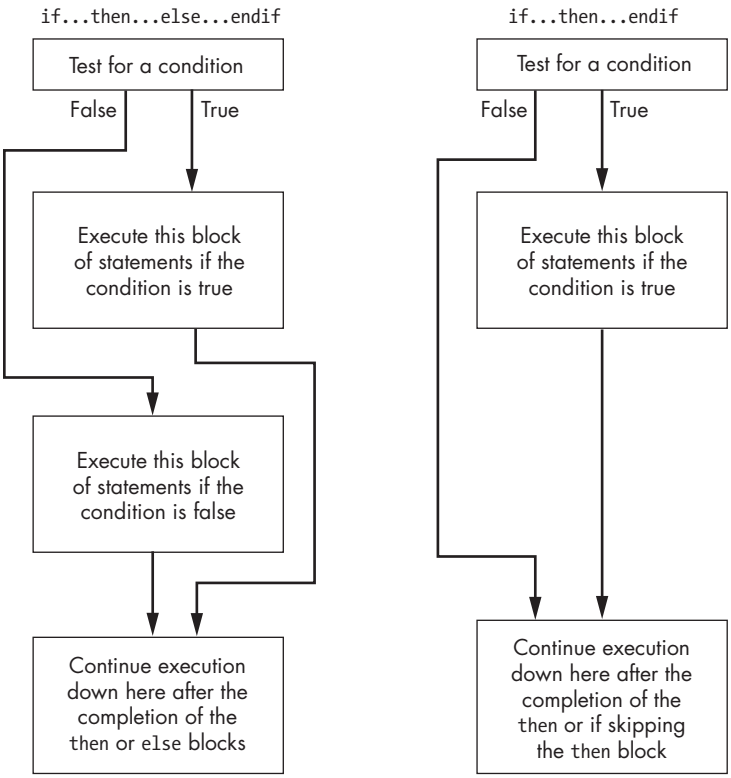


Figure 7-1: The `if...then...else...endif` and `if...then...endif` statements

The `if...then...endif` statement is just a special case of the `if...then...else...endif` statement (with an empty `else` block). The basic implementation of an `if...then...else...endif` statement in ARM assembly language looks something like this

```
    Sequence of statements to test a condition
          bcc ElseCode;

    Sequence of statements corresponding to the THEN block

          b.al EndOfIf
```

```
ElseCode:
    Sequence of statements corresponding to the ELSE block

EndOfIf:
```

where bcc represents a conditional branch instruction (typically the opposite branch of the condition being tested).

For example, suppose you want to convert the C/C++ statement into assembly language:

```
if(aa == bb)
    c = d;
else
    bb = bb + 1;
```

To do so, you could use the following ARM code:

```
          ldr  w0, [fp, #aa]   // aa and bb are 32-bit integers
          ldr  w1, [fp, #bb]   // in the current activation record.
          cmp  w0, w1
          bne  ElseBlk         // Use opposite branch to goto else.
          ldr  w0, [sb, #d]    // Assume c and d are 32-bit static
          str  w0, [sb, #c]    // variables in the static base
          b.al EndOfIf         // structure (pointed at by SB).

ElseBlk:
          ldr w0, [fp, #bb]
          add w0, w0, #1
          str w0, [fp, #bb]

EndOfIf:
```

For simple expressions like (aa == bb), generating the proper code for an if...then...else...endif statement is easy. Should the expression become more complex, the code complexity increases as well. Consider the following C/C++ if statement:

```
if(((x > y) && (z < t )) || (aa != bb))
    c = d;
```

To convert a complex if statement such as this one, break it into a sequence of three if statements as follows (assuming the use of short-circuit evaluation; see section 7.6.5, "Short-Circuit vs. Complete Boolean Evaluation," on page 382 for details):

```
if(aa != bb)
    c = d;
else if(x > y)
    if(z < t)
         c = d;
```

This conversion comes from the following C/C++ equivalences:

```
if(expr1 && expr2) stmt;
```

is equivalent to

```
if(expr1) if(expr2) stmt;
```

and

```
if(expr1 || expr2) stmt;
```

is equivalent to

```
if(expr1) stmt;
else if(expr2) stmt;
```

In assembly language, the former `if` statement becomes the following:

```
// if(((x > y) && (z < t)) || (aa != bb))
//      c = d;
//
// Assume x = W0, y = W1, z = W2, t = W3, aa = W4, bb = W5, c = W6, and d = W7
// and all variables are signed integers.

          cmp  w4, w5     // (aa != bb)?
          bne  DoIf
          cmp  w0, w1     // (x > y)?
          bngt EndOfIf    // Not greater than
          cmp  w2, w3     // (z < t)?
          bnlt EndOfIf    // Not less than
DoIf:
          mov  w6, w7     // c = d
EndOfIf:
```

Note the use of opposite branches to suggest that falling through is the main condition to consider.

The biggest problem with complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. HLL expressions are much easier to read and comprehend, so well-written comments are essential for clear assembly language implementations of `if...then...else...endif` statements. The following code shows an elegant implementation of the preceding example:

```
// if(((x > y) && (z < t)) || (aa != bb))
//      c = d;
//
// Assume x = W0, y = W1, z = W2, t = W3, aa = W4, bb = W5, c = W6,
// and d = W7.
//
// Implemented as:
```

```
//
// if (aa != bb) then goto DoIf

          cmp  w4, w5   // (aa != bb)?
          bne  DoIf

// if not (x > y) then goto EndOfIf

          cmp  w0, w1   // (x > y)?
          bngt EndOfIf  // Not greater than

// if not (z < t ) then goto EndOfIf

          cmp  w2, w3   // (z < t)?
          bnlt EndOfIf  // Not less than

// true block:

DoIf:
          mov w6, w7   // c = d
EndOfIf:
```

Whenever you're working in assembly language, don't forget to step back for a moment and see if you can rethink the solution in assembly language rather than playing "human C/C++ compiler." When working with a complex Boolean expression, your first thought should be, "Can I use the conditional compare instruction to resolve this?" The following example does just that:

```
// if(((x > y) && (z < t)) || (aa != bb))
//      c = d;
//
// Assume x = W1, y = W2, z = W3, t = W4, aa = W5, bb = W6, c = W0, and d = W7.

            cmp    w1, w2              // x > y   ? gt : ngt (C ternary ?: op)
            ccmp   w3, w4, #ccnlt, gt  // x > y   ? gt : ngt
            ccmp   w5, w6, #ccne, nlt  // nlt   ? (a != bb ? ne : nne) : ne
            csel   w0, w7, w0, ne      // if(ne) c = d
```

The `cmp` instruction sets the flags for `(x > y)`. The first `ccmp` instruction sets the flags to simulate a signed `ge` (not less than) if `(x <= y)` or based on the comparison of `(z < t)` if `(x > y)`. After executing the first `ccmp` instruction, N ≠ V if `(x > y) && (z < t)`.

Upon executing the second `ccmp` instruction, if N ≠ V (meaning signed less than), the code just sets NZCV to simulate `ne` and doesn't bother comparing `aa` and `bb` (because the left-hand side of the disjunction operator is already true, there is no need to evaluate the third parenthetical expression). Setting Z = 0 means the `csel` instruction will copy `d` to `c` (based on the `ne` condition).

If N = V when executing the second `ccmp` instruction, the `ge` condition is true, which means that the conjunction operation yielded false and you

must test to see if aa does not equal bb. That will set the flags appropriately for the csel instruction. Listing 7-3 demonstrates the execution of this conditional comparison code.

```
// Listing7-3.S
//
// Demonstrate the ccmp instruction
// handling complex Boolean expressions.

#include    "aoaa.inc"

            .data

xArray:     .word   -1, 0, 1,-1, 0, 1,-1, 0, 1, 1
yArray:     .word   -1,-1,-1, 0, 0, 0, 1, 1, 1, 0
zArray:     .word   -1, 0, 1,-1, 0, 1,-1, 0, 1, 0
tArray:     .word    0, 0, 0, 1, 1, 1,-1,-1,-1, 1
aArray:     .word    0, 0, 0,-1,-1,-1, 1, 1, 1, 1
bArray:     .word   -1, 0, 1,-1, 0, 1,-1, 0, 1, 1
size        =       10

            .text
            .pool
ttlStr:     wastr   "Listing 7-3"
fmtStr1:    .ascii  "((x > y) && (z < t)) || (aa != bb)\n"
            .ascii  " x  y  z  t aa bb Result\n"
            wastr   "-- -- -- -- -- -- ------\n"
fmtStr2:    wastr   "%2d %2d %2d %2d %2d %2d   %2d\n"

// getTitle
//
// Return pointer to program title
// to the C++ code:

            proc    getTitle, public
            adr     x0, ttlStr
            ret
            endp    getTitle

//////////////////////////////////////////////////////
//
// Here's the asmMain procedure:

            proc    asmMain, public

            locals  am
            qword   saveX1920
            qword   saveX2122
            qword   saveX2324
            dword   saveX25
            byte    stackSpace, 64
            endl    am
```

```
            enter   am.size

// Save nonvolatile registers and initialize
// them to point at xArray, yArray, zArray,
// tArray, aArray, and bArray:

            stp     x19, x20, [fp, #saveX1920]
            stp     x21, x22, [fp, #saveX2122]
            stp     x23, x24, [fp, #saveX2324]
            str     x25, [fp, #saveX25]

#define x   x19
#define y   x20
#define z   x21
#define t   x22
#define aa  x23
#define bb  x24

            lea     x, xArray
            lea     y, yArray
            lea     z, zArray
            lea     t, tArray
            lea     aa, aArray
            lea     bb, bArray

            lea     x0, fmtStr1
            bl      printf

// Loop through the array elements
// and print their values along
// with the result of
// ((x > y) && (z < t)) || (aa != bb)

            mov     x25, #0
rptLp:      ldr     w1, [x, x25, lsl #2]    // W1 = x[X25]
            ldr     w2, [y, x25, lsl #2]    // W2 = y[X25]
            ldr     w3, [z, x25, lsl #2]    // W3 = z[X25]
            ldr     w4, [t, x25, lsl #2]    // W4 = t[X25]
            ldr     w5, [aa, x25, lsl #2]   // W5 = aa[X25]
            ldr     w6, [bb, x25, lsl #2]   // W6 = bb[X25]

            cmp     w1, w2
            ccmp    w3, w4, #ccnlt, gt
            ccmp    w5, w6, #ccne, nlt
            cset    w7, ne

            lea     x0, fmtStr2
            mstr    w1, [sp]
            mstr    w2, [sp, #8]
            mstr    w3, [sp, #16]
            mstr    w4, [sp, #24]
            mstr    w5, [sp, #32]
            mstr    w6, [sp, #40]
            mstr    w7, [sp, #48]
```

```
            bl      printf
            add     x25, x25, #1
            cmp     x25, #size
            blo     rptLp

// Restore nonvolatile register values
// and return:

            ldp     x19, x20, [fp, #saveX1920]
            ldp     x21, x22, [fp, #saveX2122]
            ldp     x23, x24, [fp, #saveX2324]
            ldr     x25, [fp, #saveX25]

            leave
            endp    asmMain
```

Here's the `build` command and sample output for Listing 7-3:

```
$ ./build Listing7-3
$ ./Listing7-3
Calling Listing7-3:
((x > y) && (z < t)) || (aa != bb)
 x  y  z  t aa bb Result
-- -- -- -- -- -- ------
-1 -1 -1  0  0 -1      1
 0 -1  0  0  0  0      0
 1 -1  1  0  0  1      1
-1  0 -1  1 -1 -1      0
 0  0  0  1 -1  0      1
 1  0  1  1 -1  1      1
-1  1 -1 -1  1 -1      1
 0  1  0 -1  1  0      1
 1  1  1 -1  1  1      0
 1  0  0  1  1  1      1
Listing7-3 terminated
```

The output shows the truth table for the given expression.

7.6.3 Complex if Statements Using Complete Boolean Evaluation

Many Boolean expressions involve conjunction (AND) or disjunction (OR) operations. You can convert such Boolean expressions into assembly language in two ways: using *complete Boolean evaluation* or using *short-circuit Boolean evaluation*. This section discusses complete Boolean evaluation, and the next discusses short-circuit Boolean evaluation.

Conversion via complete Boolean evaluation is almost identical to converting arithmetic expressions into assembly language, as covered in section 6.4, "Logical Expressions," on page 312. However, for Boolean evaluation, you do not need to store the result in a variable; once the evaluation of the expression is complete, you check whether you have a false (0) or true (1, or nonzero) result to take whatever action the Boolean expression dictates. Remember that only the `ands` instruction sets the zero flag; there is

no orrs instruction. Consider the following if statement and its conversion to assembly language using complete Boolean evaluation:

```
//     if(((x < y) && (z > t)) || (aa != bb))
//          Stmt1 ;
//
// Assume all variables are 32-bit integers and are local
// variables in the activation record.

          ldr  w0, [fp, #x]
          ldr  w1, [fp, #y]
          cmp  w0, w1
          cset w7, lt        // Store x < y in W7.
          ldr  w0, [fp, #z]
          ldr  w1, [fp, #t]
          cmp  w0, w1
          cset w6, gt        // Store z > t in W6.
          and  w6, w6, w7    // Put (x < y) && (z > t) into W6.
          ldr  w0, [fp, #aa]
          ldr  w1, [fp, #bb]
          cmp  w0, w1
          cset w0, ne        // Store aa != bb into W0.
          orr  w0, w0, w6    // Put (x < y) && (z > t) ||
          cmp  w0, #0        //    (aa != bb) into W0.
          beq  SkipStmt1     // Branch if result is false.

     Code for Stmt1

SkipStmt1:
```

This code computes a Boolean result in the W0 register and then, at the end of the computation, tests this value to see whether it contains true or false. If the result is false, this sequence skips over the code associated with Stmt1. The important thing is that the program will execute every instruction that computes this Boolean result (up to the beq instruction).

By now you should recognize that we can improve this code by using the ccmp instruction:

```
          ldr  w0, [fp, #x]
          ldr  w1, [fp, #y]
          cmp  w0, w1
          ldr  w0, [fp, #z]
          ldr  w1, [fp, #t]
          ccmp w0, w1, #ccngt, lt
          ldr  w0, [fp, #aa]
          ldr  w1, [fp, #bb]
          ccmp w0, w1, #cceq, gt
          beq  SkipStmt1     // Branch if result is false.

     Code for Stmt1

SkipStmt1:
```

The code is still a bit longer than usual, but this is the result of using memory variables rather than registers for everything in this example. Even though this example uses the `ccmp` instruction, the code still executes each and every instruction in the sequence, even if the condition becomes false early on and could never become true.

7.6.4 Short-Circuit Boolean Evaluation

If you are willing to expend a little more effort (and your Boolean expression doesn't depend on side effects), you can often convert a Boolean expression to a faster sequence of assembly language instructions by using *short-circuit Boolean evaluation*. This approach attempts to determine whether an expression is true or false by executing only some of the instructions that would compute the complete expression.

Consider the expression `aa && bb`. Once you determine that `aa` is false, there is no need to evaluate `bb` because there is no way the expression can be true. If `bb` represents a complex subexpression rather than a single Boolean variable, it should be clear that evaluating only `aa` is more efficient.

As a concrete example, consider the subexpression `((x < y) && (z > t))`. Once you determine that `x` is not less than `y`, there is no need to check whether `z` is greater than `t` because the expression will be false regardless of the values of `z` and `t`. The following code fragment shows how to implement short-circuit Boolean evaluation for this expression:

```
// if((x < y) && (z > t)) then ...

          ldr  w0, [fp, #x]
          ldr  w1, [fp, #y]
          cmp  w0, w1
          bnlt TestFails
          ldr  w0, [fp, #z]
          ldr  w1, [fp, #t]
          cmp  w0, w1
          bngt TestFails

      Code for THEN clause of IF statement

TestFails:
```

The code skips any further testing once it determines that `x` is not less than `y`. Of course, if `x` is less than `y`, the program has to test `z` to see if it is greater than `t`; if not, the program skips over the `then` clause. Only if the program satisfies both conditions does the code fall through to the `then` clause.

For the logical OR operation, the technique is similar. If the first subexpression evaluates to true, there is no need to test the second operand. Whatever the second operand's value is at that point, the full expression still evaluates to true. The following example demonstrates the use of short-circuit evaluation with disjunction (`||`):

```
// if(W0 < 'A' || W0 > 'Z')
//     then printf("Not an uppercase char");
// endif;

          cmp  w0, #'A'
          blo  ItsNotUC
          cmp  w0, #'Z'
          bnhi ItWasUC

ItsNotUC:
  Code to process W0 if it's not an uppercase character

ItWasUC:
```

Because the conjunction and disjunction operators are commutative, you can evaluate the left or right operand first if it is more convenient to do so.

Be aware that some expressions depend on the leftmost subexpression evaluating one way in order for the rightmost subexpression to be valid; `if(x != NULL && x -> y)` is a common test in C/C++, for example.

As one last example in this section, consider the full Boolean expression from the previous section:

```
// if(((x < y) && (z > t)) || (aa != bb))  Stmt1 ;

          ldr  w0, [sb, #aa]   // Assume aa and bb are globals.
          ldr  w1, [sb, #bb]
          cmp  w0, w1
          bne  DoStmt1
          ldr  w0, [fp, #x]   // Assume x, y, z, and t
          ldr  w1, [fp, #y]   // are all locals.
          cmp  w0, w1
          bnlt SkipStmt1
          ldr  w0, [fp, #z]
          ldr  w1, [fp, #t]
          cmp  w0, w1
          bngt SkipStmt1

DoStmt1:
      Code for Stmt1

SkipStmt1:
```

The code in this example evaluates `aa != bb` first, because it is shorter and faster, and the remaining subexpression last. This is a common technique assembly language programmers use to write better code.

This assumes, of course, that all comparisons are equally likely to be true or false. If you can predict that the subexpression `aa != bb` will be false the vast majority of the time, it would be best to test that condition last.

7.6.5 Short-Circuit vs. Complete Boolean Evaluation

When using complete Boolean evaluation, every statement in the sequence for that expression will execute; short-circuit Boolean evaluation, on the other hand, may not require the execution of every statement associated with the Boolean expression. As you've seen in the previous two sections, code based on short-circuit evaluation is often shorter and possibly faster.

However, short-circuit Boolean evaluation may not produce the correct result in some cases. Given an expression with *side effects* (changes to variables within the expression), short-circuit Boolean evaluation will produce a different result than complete Boolean evaluation. Consider the following C/C++ example:

```
if((x == y) && (++z != 0))  Stmt;
```

Using complete Boolean evaluation, you might generate the following:

```
        ldr  w0, [fp, #x]      // See if x == y.
        ldr  w1, [fp, #y]
        cmp  w0, w1
        ldr  w2, [fp, #z]
        add  w2, w1, 1         // ++z
        str  w2, [fp, #z]
        ccmp w2, #0, #cceq, eq
        beq  SkipStmt

    Code for Stmt

SkipStmt:
```

The `ccmp` instruction compares the incremented value of `z` against 0, but only if `x` is equal to `y`. If `x` is not equal to `y`, the `ccmp` instruction sets the Z flag to 1 so that control transfers to `SkipStmt` with the following `beq` instruction.

Using short-circuit Boolean evaluation, you might generate the following code:

```
        ldr  w0, [fp, #x]      // See if x == y.
        ldr  w1, [fp, #y]
        cmp  w0, w1
        bne  SkipStmt
        ldr  w2, [fp, #z]
        adds w2, w1, 1         // ++z -- sets Z flag if z
        str  w2, [fp, #z]      // becomes 0.
        beq  SkipStmt          // See if incremented z is 0.

    Code for Stmt

SkipStmt:
```

A subtle but important difference exists between these two conversions: if x is equal to y, the first version still *increments z and compares it to 0* before it executes the code associated with Stmt. The short-circuit version, on the other hand, skips the code that increments z if it turns out that x is equal to y. Therefore, the behavior of these two code fragments is different if x is equal to y.

Neither implementation is wrong; depending on the circumstances, you may or may not want the code to increment z if x is equal to y. However, it is important to realize that these two schemes produce different results, so you can choose an appropriate implementation if the effect of this code on z matters to your program.

Many programs take advantage of short-circuit Boolean evaluation and rely on the program not evaluating certain components of the expression. The following C/C++ code fragment demonstrates perhaps the most common example that requires short-circuit Boolean evaluation:

```
if( pntr != NULL && *pntr == 'a' )  Stmt;
```

If it turns out that pntr is NULL, the expression is false, and there is no need to evaluate the remainder of the expression. This statement relies on short-circuit Boolean evaluation for correct operation. Were C/C++ to use complete Boolean evaluation, the second half of the expression would attempt to dereference a NULL pointer, when pntr is NULL.

Consider the translation of this statement using complete Boolean evaluation:

```
// Complete Boolean evaluation:

        ldr  x0, [fp, #pntr]
        cmp  x0, #0    // Check to see if X0 is 0 (NULL is 0).
        cset w1, ne    // w1 = pntr != NULL
        ldrb w0, [x0]  // Get *pntr into W0.
        cmp  w0, #'a'
        cset w2, eq
        ands w1, w1, w2
        beq  SkipStmt

    Code for Stmt

SkipStmt:
```

If pntr contains NULL (0), this program will attempt to access the data at location 0 in memory via the ldrb w0, [x0] instruction. Under most OSes, this will cause a memory access fault (segmentation fault).

Now consider the short-circuit Boolean conversion:

```
    ldr  x0, [fp, #pntr] // See if pntr contains NULL (0)
    cmp  x0, #0          // and immediately skip past Stmt
    beq  SkipStmt        // if this is the case.
```

```
        ldrb w0, [x0]       // If we get to this point, pntr
        cmp  w0, #'a'       // contains a non-NULL value, so see
        bne  SkipStmt       // if it points at the character 'a'.

         Code for Stmt

SkipStmt:
```

In this example, the problem with dereferencing the `NULL` pointer doesn't exist. If `pntr` contains `NULL`, this code skips over the statements that attempt to access the memory address that `pntr` contains.

7.6.6 Efficient Implementation of if Statements in Assembly Language

Encoding `if` statements efficiently in assembly language takes a little more thought than simply choosing short-circuit evaluation over complete Boolean evaluation. To write code that executes as quickly as possible in assembly language, you must carefully analyze the situation and generate the code appropriately. The following paragraphs provide suggestions you can apply to your programs to improve their performance.

7.6.6.1 Knowing Your Data

Programmers often mistakenly assume that data is random. In reality, data is rarely random, and if you know the types of values that your program commonly uses, you can write better code. To see how, consider the following C/C++ statement:

```
if((aa == bb) && (c < d)) ++i;
```

Because C/C++ uses short-circuit evaluation, this code will test whether `aa` is equal to `bb`. If so, it will test whether `c` is less than `d`. If you expect `aa` to be equal to `bb` most of the time but don't expect `c` to be less than `d` most of the time, this statement will execute slower than it should. Consider the following Gas implementation of this code:

```
        ldr  w0, [fp, #aa]
        ldr  w1, [fp, #bb]
        cmp  w0, w1
        bne  DontIncI

        ldr  w0, [fp, #c]
        ldr  w1, [fp, #d]
        cmp  w0, w1
        bnlt DontIncI

        ldr  w0, [sb, #i]
        add  w0, w0, #1
        str  w0, [sb, #i]

DontIncI:
```

As you can see, if aa is equal to bb most of the time and c is not less than d most of the time, you will have to execute the first eight instructions nearly every time in order to determine that the expression is false. Now consider the following implementation that takes advantage of this knowledge and the fact that the && operator is commutative:

```
        ldr  w0, [fp, #c]
        ldr  w1, [fp, #d]
        cmp  w0, w1
        bnlt DontIncI

        ldr  w0, [fp, #aa]
        ldr  w1, [fp, #bb]
        cmp  w0, w1
        bne  DontIncI

        ldr  w0, [sb, #i]
        add  w0, w0, #1
        str  w0, [sb, #i]

DontIncI:
```

The code first checks whether c is less than d. If most of the time c is not less than d, this code determines that it has to skip to the label DontIncI after executing only three instructions in the typical case, compared with seven instructions in the previous example.

Optimizations like this are much more obvious in assembly language than in an HLL, one of the main reasons assembly programs are often faster than their HLL counterparts. The key here is to understand the behavior of your data so you can make intelligent decisions.

7.6.6.2 Rearranging Expressions

Even if your data is random, or you can't determine how the input values will affect your decisions, rearranging the terms in your expressions may still be beneficial. Some calculations take far longer to compute than others. For example, computing the remainder is slower than a simple cmp instruction. Therefore, if you have a statement like the following, you may want to rearrange the expression so that the cmp comes first:

```
if((x % 10 = 0) && (x != y)) ++x;
```

Converted directly to assembly code, this if statement becomes the following:

```
        ldr   w1, [fp, #x]      // Compute x % 10.
        mov   w2, #10
        udiv  w0, w1, w2
        msub  w0, w0, w2, w1
        cmp   w0, #0
        bne   SkipIf
```

```
        ldr   w0, [fp, #x]
        ldr   w1, [fp, #y]
        cmp   w0, w1
        beq   SkipIf

        add   w0, w0, #1        // ++x
        str   w0, [fp, #x]

SkipIf:
```

The remainder computation is expensive (about one-third the speed of most of the other instructions in this example). Unless it is three times more likely that the remainder is 0 rather than x is equal to y, it would be better to do the comparison first and the remainder calculation afterward:

```
        ldr   w1, [fp, #x]      // Compute x % 10.
        ldr   w1, [fp, #y]
        cmp   w0, w1
        beq   SkipIf

        ldr   w1, [fp, #x]
        mov   w2, #10
        udiv  w0, w1, w2
        msub  w0, w0, w2, w1
        cmp   w0, #0
        bne   SkipIf

        add   w1, w1, #1        // ++x
        str   w1, [fp, #x]

SkipIf:
```

The && and || operators are commutative in the mathematical sense that if you evaluate that left or right side first, the logical result is the same. In terms of execution, they are not commutative because the order of evaluation may cause the code to skip the evaluation of the second subexpression; in particular, these operators may not be commutative if side effects occur within the expression. This example works fine because there are no side effects or possible exceptions being shielded by the reordered evaluation of the && operator.

7.6.6.3 Destructuring Code

Structured code is sometimes less efficient than unstructured code because it introduces code duplication or extra branches that might not be present in unstructured code. Most of the time, this is tolerable because unstructured code is difficult to read and maintain; sacrificing some performance in exchange for maintainable code is often acceptable. In certain instances, however, you may need all the performance you can get and might choose to compromise the readability of your code.

In HLLs, you can often get away with writing structured code because the compiler will optimize it, producing unstructured machine code. Unfortunately, when writing in assembly language, the machine code you get is exactly equivalent to the assembly code you write.

Taking previously written structured code and rewriting it in an unstructured fashion to improve performance is known is *destructuring code*. The difference between unstructured code and destructured code is that unstructured code was written that way in the first place; destructured code started out as structured code and was purposefully written in an unstructured fashion to make it more efficient. Pure unstructured code is usually hard to read and maintain. Destructured code isn't quite as bad because you limit the damage (unstructuring the code) to only those sections where it is absolutely necessary.

One classic way to destructure code is to use *code movement*, physically moving sections of code elsewhere in the program. You move code that your program rarely uses out of the way of code that executes most of the time.

Code movement can improve the efficiency of a program two ways. First, a branch that is taken is more expensive (time-consuming) than a branch that is not taken. If you move the rarely used code to another spot in the program and branch to it on the rare occasion the branch is taken, most of the time you will fall straight through to the code that executes most frequently. Second, sequential machine instructions consume cache storage. If you move rarely executed statements out of the normal code stream to another section of the program that is rarely loaded into cache, this will improve the cache performance of the system.

For example, consider the following pseudo C/C++ statement:

```
if(See_If_an_Error_Has_Occurred)
{
     Statements to execute if no error
}
else
{
     Error-handling statements
}
```

In normal code, you don't expect errors to be frequent. Therefore, you would typically expect the `then` section of the preceding `if` to execute far more often than the `else` clause. The preceding code could translate into the following assembly code:

```
     cmp See_If_an_Error_Has_Occurred, #true
     beq HandleTheError

          Statements to execute if no error

     b.al EndOfIf
```

```
HandleTheError:
          Error-handling statements
EndOfIf:
```

If the expression is false, this code falls through to the normal statements and then jumps over the error-handling statements. Instructions that transfer control from one point in your program to another (for example, `b.al` instructions) tend to be slow. It is much faster to execute a sequential set of instructions than to jump all over the place in your program. Unfortunately, the preceding code doesn't allow this.

One way to rectify this problem is to move the `else` clause of the code somewhere else in your program. You could rewrite the code as follows:

```
     cmp See_If_an_Error_Has_Occurred, #true
     beq HandleTheError

          Statements to execute if no error

EndOfIf:

  // At some other point in your program (typically after a b.al
  // or ret instruction), you would insert the following code:

HandleTheError:
          Error-handling statements
     b.al EndOfIf
```

The program isn't any shorter. The `b.al` you removed from the original sequence winds up at the end of the `else` clause. However, because the `else` clause rarely executes, moving the `b.al` instruction from the `then` clause (which executes frequently) to the `else` clause is a big performance win, because the `then` clause executes using only straight-line code. This technique is surprisingly effective in many time-critical code segments.

7.6.6.4 Calculating Rather than Branching

On the ARM processor, branches are expensive compared to many other instructions. For this reason, it is sometimes better to execute more instructions in a sequence than fewer instructions that involve branching.

For example, consider the simple assignment `w0 = abs(w0)`. Unfortunately, no ARM instruction computes the absolute value of an integer. The obvious way to handle this is with an instruction sequence that uses a conditional jump to skip over the `neg` instruction (which creates a positive value in W0 if W0 was negative):

```
          cmp w0, #0
          bpl ItsPositive

          neg w0, w0

ItsPositive:
```

Now consider the following sequence that will also do the job:

```
cmp  w0, #0
cneg w0, w0, mi
```

Not only is the instruction set shorter, it also doesn't involve any branches, so it runs faster. This demonstrates why it's good to know the instruction set!

Another example of calculation versus branching that you've seen is using the `ccmp` instruction to handle conjunction and disjunction in a Boolean expression (see section 7.6.5, "Short-Circuit vs. Complete Boolean Evaluation," on page 382). Though they tend to execute more instructions than short-circuit evaluation, no branching is involved, and this often equates to faster-running code.

Sometimes calculation without branching isn't possible. For certain types of branches (in particular, multiway branches), you can combine calculations with a single branch to handle complex operations, as discussed in the next section.

7.6.7 switch...case Statements

The C/C++ `switch` statement takes the following form:

```
switch(expression)
 {
    case const1:
      Code to execute if
          expression equals const1

    case const2:
      Code to execute if
          expression equals const2
      .
      .
      .
    case constn:
      Code to execute if
          expression equals constn

    default:  // Note that the default section is optional.
      Code to execute if expression
          does not equal any of the case values

 }
```

When this statement executes, it checks the value of the *expression* against the constants *const1* to *constn*. If it finds a match, the corresponding statements execute.

C/C++ places a few restrictions on the `switch` statement. First, it allows only an integer expression (or something whose underlying type can be an integer). Second, all the constants in the `case` clauses must be unique. The

next few subsections describe the semantics of the switch statement and various implementations and clarify the reasons for the restrictions.

7.6.7.1 switch Statement Semantics

Most introductory programming texts introduce the switch...case statement by explaining it as a sequence of if...then...elseif...else...endif statements. They might claim that the following two pieces of C/C++ code are equivalent:

```
switch( w0 )
{
    case 0: printf("i=0"); break;
    case 1: printf("i=1"); break;
    case 2: printf("i=2"); break;
}

if( w0 == 0 )
    printf("i=0");
else if( w0 == 1 )
    printf("i=1");
else if( w0 == 2 )
    printf("i=2");
```

While semantically these two code segments may be the same, their implementation is usually different. Whereas the if...then...elseif...else...endif chain does a comparison for each conditional statement in the sequence, the switch statement normally uses an indirect jump to transfer control to any one of several statements with a single computation.

7.6.7.2 if...else Implementation of switch

The switch (and if...else...elseif) statements could be written in assembly language with the following code:

```
// if...then...else...endif form:

          ldr w0, [fp, #i]
          cmp w0, #0          // Check for 0.
          bne Not0

  Code to print "i = 0"

          b.al EndCase

Not0:
          cmp w0, #1
          bne Not1

  Code to print "i = 1"

          b.al EndCase
```

```
Not1:
          cmp w0, #2
          bne EndCase

  Code to print "i = 2"

EndCase:
```

This code takes longer to determine that the last case should execute than it does to determine whether the first case executes. This is because the `if...else...elseif` version implements a linear search through the case values, checking them one at a time from first to last until it finds a match.

7.6.7.3 Indirect Jump switch Implementation

A faster implementation of the `switch` statement is possible by using an *indirect jump table* (a table containing target addresses to jump to). This implementation uses the switch expression as an index into a table of addresses; each address points at the target case's code to execute. The following example demonstrates the use of a jump table:

```
// Indirect jump version

        ldr  w0, [fp, #i]   // Zero-extends into X0!
     ❶ adr  x1, JmpTbl
     ❷ ldr  x0, [x1, x0, lsl #3]
     ❸ add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt0-JmpTbl, Stmt1-JmpTbl, Stmt2-JmpTbl

Stmt0:

  Code to print "i = 0"

        b.al EndCase

Stmt1:

  Code to print "i = 1"

         b.al EndCase

Stmt2:

  Code to print "i = 2"

EndCase:
```

To use the scaled-indexed addressing mode, this code begins by loading the address of the jump table (`JmpTbl`) into X1 ❶. Because `JmpTbl`

is in the .text section (and nearby), the code uses the PC-relative addressing mode.

The code fetches the *i*th entry from JmpTbl ❷. Because each entry in the jump table is 8 bytes long, the code must multiply the index (i, which is in X0) by 8, which the lsl #3 argument handles. The base address (in X1) plus index times 8 gives the address of the appropriate entry in JmpTbl.

Because the entries in JmpTbl are offsets rather than absolute addresses (remember, macOS doesn't allow absolute addresses in the .text section), you must convert the offset to an absolute address by adding in the base address of the jump table ❸ (as each entry in the table is an offset from the base address). The following br instruction transfers control to the appropriate case in the switch statement.

To begin, a switch statement requires that you create an array of pointers, with each element containing the address of a statement label in your code; those labels must be attached to the sequence of instructions to execute for each case in the switch statement. As noted in the code annotations, macOS does not allow absolute addresses here, so the code uses offsets from the base address of the jump table, which also works for Linux. In this example, the JmpTbl array, initialized with the offsets of the statement labels Stmt0, Stmt1, and Stmt2, serves this purpose. You must place the jump-table array in a location that will never be executed as code (such as immediately after a br instruction, as in this example).

The program loads the W0 register with the value of i (assuming i is a 32-bit unsigned integer, the ldr instruction zero-extends W0 into X0). It then uses this value as an index into the JmpTbl array (W1 holds the base address of the JmpTbl array) and transfers control to the 8-byte address found at the specified location. For example, if W0 contains 0, the br x0 instruction will fetch the double word at address JmpTbl+0 ($W0 \times 8 = 0$). Because the first double word in the table contains the offset of Stmt0, the br instruction transfers control to the first instruction following the Stmt0 label. Likewise, if i (and therefore, W0) contains 1, then the indirect br instruction fetches the double word at offset 8 from the table and transfers control to the first instruction following the Stmt1 label (because the offset of Stmt1 appears at offset 8 in the table). Finally, if i (W0) contains 2, then this code fragment transfers control to the statements following the Stmt2 label because it appears at offset 16 in the JmpTbl table.

As you add more (consecutive) cases, the jump-table implementation becomes more efficient (in terms of both space and speed) than the if... elseif form. Except for simple cases, the switch statement is almost always faster, and usually by a large margin. As long as the case values are consecutive, the switch statement version is often smaller as well.

7.6.7.4 Noncontiguous Jump-Table Entries and Range Limiting

What happens if you need to include nonconsecutive case labels or can't be sure the switch value doesn't go out of range? With the C/C++ switch statement, such an occurrence will transfer control to the first statement after the switch statement (or to a default case, if one is present in the switch).

However, this doesn't happen in the example in the previous section. If variable i does not contain 0, 1, or 2, executing the previous code produces undefined results. For example, if i contains 5 when you execute the code, the indirect br instruction will fetch the dword at offset 40 (5×8) in JmpTbl and transfer control to that offset. Unfortunately, JmpTbl doesn't have six entries, so the program will fetch the value of the sixth double word following JmpTbl and use that as the target offset, which will often crash your program or transfer control to an unexpected location.

The solution is to place a few instructions before the indirect br to verify that the switch selection value is within a reasonable range. In the previous example, you'd want to verify that the value of i is in the range 0 to 2 before executing the br instruction. If the value of i is outside this range, the program should simply jump to the endcase label, which corresponds to dropping down to the first statement after the entire switch statement. The following code provides this modification:

```
        ldr  w0, [fp, #i]  // Zero-extends into X0!
        cmp  w0, #2        // Default case if i > 2
        bhi  EndCase
        adr  x1, JmpTbl
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt0-JmpTbl, Stmt1-JmpTbl, Stmt2-JmpTbl

Stmt0:

  Code to print "i = 0"

        b.al EndCase

Stmt1:

  Code to print "i = 1"

        b.al EndCase

Stmt2:

  Code to print "i = 2"

EndCase:
```

Although this code handles the problem of selection values being outside the range 0 to 2, it still suffers from two severe restrictions:

- The cases must start with the value 0. That is, the minimum case constant has to be 0.
- The case values must be contiguous.

Solving the first problem is easy, and you deal with it in two steps. First, you compare the case selection value against a lower and upper bound before determining whether the case value is legal, as shown in the following example:

```
// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: This code does *NOT* work.
// Keep reading to find out why.

        ldr  w0, [fp, #i]  // Zero-extends into X0!
        cmp  w0, #5        // Verify i is in the range
        blo  EndCase       // 5 to 7 before indirect
        cmp  w0, #7        // branch executes.
        bhi  EndCase
        adr  x1, JmpTbl
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt5-JmpTbl, Stmt6-JmpTbl, Stmt7-JmpTbl

Stmt5:
  Code to print "i = 5"

        b.al EndCase

Stmt6:
  Code to print "i = 6"

        b.al EndCase

Stmt7:
  Code to print "i = 7"

EndCase:
```

This code adds a pair of extra instructions, `cmp` and `blo`, to test the selection value to ensure it is in the range 5 to 7. If not, control drops down to the `EndCase` label; otherwise, control transfers via the indirect `br` instruction. Unfortunately, as the comments point out, this code is broken.

Consider what happens if variable `i` contains the value 5: the code will verify that 5 is in the range 5 to 7 and then will fetch the dword at offset 40 (5×8) and jump to that address. As before, however, this loads 8 bytes outside the bounds of the table and does not transfer control to a defined location. One solution is to subtract the smallest case selection value from W0 before executing the `br` instruction, as shown in the following example:

```
// SWITCH statement specifying cases 5, 6, and 7:

        ldr  w0, [fp, #i]  // Zero-extends into X0!
        subs w0, w0, #5    // Subtract smallest range.
        blo  EndCase       // Subtract sets flags same as cmp!
```

```
        cmp  w0, #7-5      // Verify in range 5 to 7.
        bhi  EndCase
        adr  x1, JmpTbl
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt5-JmpTbl, Stmt6-JmpTbl, Stmt7-JmpTbl

Stmt5:
  Code to print "i = 5"

        b.al EndCase

Stmt6:
  Code to print "i = 6"

        b.al EndCase

Stmt7:
  Code to print "i = 7"

EndCase:
```

By subtracting 5 from the value in W0, the code forces W0 to take on the value 0, 1, or 2 prior to the `br` instruction. Therefore, case-selection value 5 jumps to `Stmt5`, case-selection value 6 transfers control to `Stmt6`, and case-selection value 7 jumps to `Stmt7`.

This code has one piece of trickery: the `subs` instruction serves double duty. It not only adjusts the lower bound of the `switch` expression down to 0 but also serves as the comparison against 5 for the lower bound. Remember, the `cmp` instruction sets the flags the same way as the `subs` instruction. Therefore, subtracting 5 is the same as comparing against 5 as far as the flag settings are concerned. When comparing the value in W0 against 7, the code must actually compare against 2 because we've subtracted 5 from the original index value.

You can handle cases that don't start with 0 in another way:

```
// SWITCH statement specifying cases 5, 6, and 7:

        ldr  w0, [fp, #i]  // Zero-extends into X0!
        cmp  w0, #5        // Verify the index is in
        blo  EndCase       // the range 5 to 7.
        cmp  w0, #7
        bhi  EndCase
        adr  x1, JmpTbl - 5*8 // Base address - 40
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0
```

```
JmpTbl: .dword Stmt5-JmpTbl, Stmt6-JmpTbl, Stmt7-JmpTbl

Stmt5:

  Code to print "i = 5"

        b.al EndCase

Stmt6:

  Code to print "i = 6"

        b.al EndCase

Stmt7:

  Code to print "i = 7"

EndCase:
```

This example subtracts 40 (5×8) from the base address of the jump table when loading that base address into X1. The index is still in the range 5 to 7, yielding an offset of 40 to 56 into the table; however, because the base address is now specified 40 bytes before the actual table, the array-indexing calculation properly indexes into the jump-table entries.

The C/C++ `switch` statement provides a `default` clause that executes if the case-selection value doesn't match any of the case values. The following `switch` statement includes a `default` clause:

```
switch(expression)
{

    case 5:  printf("expression = 5"); break;
    case 6:  printf("expression = 6"); break;
    case 7:  printf("expression = 7"); break;
    default:
        printf("expression does not equal 5, 6, or 7");
}
```

Implementing the equivalent of the `default` clause in pure assembly language is easy: just use a different target label in the `blo` and `bhi` instructions at the beginning of the code. The following example implements a `switch` statement similar to the preceding one:

```
// SWITCH statement specifying cases 5, 6, and 7:

        ldr  w0, [fp, #i]     // Zero-extends into X0!
        cmp  w0, #5           // Verify the index is in
        blo  DefaultCase      // the range 5 to 7.
        cmp  w0, #7
```

```
        bhi  DefaultCase
        adr  x1, JmpTbl - 5 * 8 // Base address - 40
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt5-JmpTbl, Stmt6-JmpTbl, Stmt7-JmpTbl

Stmt5:

  Code to print "i = 5"

        b.al EndCase

Stmt6:

  Code to print "i = 6"

        b.al EndCase

Stmt7:

  Code to print "i = 7"

        b.al EndCase

DefaultCase:

  Code to print "expression does not equal 5, 6, or 7"

EndCase:
```

The second restriction noted earlier, that the case values need to be contiguous, is easy to handle by inserting extra entries into the jump table. Consider the following C/C++ `switch` statement:

```
switch(i)
{
    case 1:  printf("i = 1"); break;
    case 2:  printf("i = 2"); break;
    case 4:  printf("i = 4"); break;
    case 8:  printf("i = 8"); break;
    default:
        printf("i is not 1, 2, 4, or 8");
}
```

The minimum switch value is 1, and the maximum value is 8. Therefore, the code before the indirect `br` instruction needs to compare the value in `i` against 1 and 8. If the value is from 1 to 8, it's still possible that `i` might not contain a legal case-selection value. However, because the `br` instruction indexes into a table of double words, the table must have eight double-word entries.

To handle the values from 1 to 8 that are not case-selection values, simply put the statement label of the `default` clause (or the label specifying the first instruction after the end of the `switch` if there is no `default` clause) in each of the jump-table entries that don't have a corresponding `case` clause. The following code demonstrates this technique:

```
// SWITCH statement specifying cases 1, 2, 4, and 8:

        ldr  w0, [fp, #i]     // Zero-extends into X0!
        cmp  w0, #1           // Verify the index is in
        blo  DefaultCase      // the range 1 to 8.
        cmp  w0, #8
        bhi  DefaultCase
        adr  x1, JmpTbl - 1 * 8 // Base address - 8
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0

JmpTbl: .dword Stmt1-JmpTbl
        .dword Stmt2-JmpTbl
        .dword DefaultCase-JmpTbl // Case 3
        .dword Stmt4-JmpTbl
        .dword DefaultCase-JmpTbl // Case 5
        .dword DefaultCase-JmpTbl // Case 6
        .dword DefaultCase-JmpTbl // Case 7
        .dword Stmt8-JmpTbl

Stmt1:

  Code to print "i = 1"

        b.al EndCase

Stmt2:

  Code to print "i = 2"

        b.al EndCase

Stmt4:

  Code to print "i = 4"

        b.al EndCase

Stmt8:

  Code to print "i = 8"

        b.al EndCase

DefaultCase:
```

```
    Code to print "expression does not equal 1, 2, 4, or 8"

EndCase:
```

This code uses `cmp` instructions to ensure that the `switch` value is in the range 1 to 8 and transfers control to the `DefaultCase` label if this is the case.

7.6.7.5 Sparse Jump Tables

The current implementation of the `switch` statement has a problem. If the `case` values contain nonconsecutive entries that are widely spaced, the jump table could become exceedingly large. The following `switch` statement would generate an extremely large code file:

```
switch(i)
{
    case 1:         Stmt1 ;
    case 100:       Stmt2 ;
    case 1000:      Stmt3 ;
    case 10000:     Stmt4 ;
    default:        Stmt5 ;

}
```

In this situation, your program will be much smaller if you implement the `switch` statement with a sequence of `if` statements rather than using an indirect jump statement. However, the size of the jump table does not normally affect the execution speed of the program. If the jump table contains 2 entries or 2,000, the `switch` statement will execute the multiway branch in a constant amount of time. The `if` statement implementation requires a linearly increasing amount of time for each `case` label appearing in the `case` statement.

One of the biggest advantages to using assembly language over an HLL like Swift or C/C++ is that you get to choose the actual implementation of statements like `switch`. In some instances, you can implement a `switch` statement as a sequence of `if...then...elseif` statements, you can implement it as a jump table, or you can use a hybrid of the two. The following code examples demonstrate combining `if...then...elseif` and jump-table implementations for the same control structure:

```
switch(i)
{
    case 0:     Stmt0 ;
    case 1:     Stmt1 ;
    case 2:     Stmt2 ;
    case 100:   Stmt3 ;
    default:    Stmt4 ;

}
```

That code could become the following:

```
        ldr  w0, [fp, #i]
        cmp  w0, #100         // Special case 100
        beq  DoStmt3
        cmp  w0, #2
        bhi  DefaultCase
        adr  x1, JmpTbl
        ldr  x0, [x1, x0, lsl #3]
        add  x0, x0, x1
        br   x0
         .
         .
         .
```

Some `switch` statements have sparse cases, but the cases are often grouped into contiguous clusters. Consider the following C/C++ `switch` statement:

```
switch(expression)
{
    case 0:

  Code for case 0

        break;

    case 1:

  Code for case 1

        break;

    case 2:

  Code for case 2

        break;

    case 10:

  Code for case 10

        break;

    case 11:

  Code for case 11

        break;

    case 100:
```

```
  Code for case 100

        break;

    case 101:

  Code for case 101

        break;

    case 103:

  Code for case 103

        break;

    case 1000:

  Code for case 1000

        break;

    case 1001:

  Code for case 1001

        break;

    case 1003:

  Code for case 1003

        break;

    default:

  Code for default case

        break;
} // End switch.
```

You can convert a `switch` statement that consists of widely separated groups of (nearly) contiguous cases to assembly language code using one jump-table implementation for each contiguous group, then use comparison instructions to determine which jump-table instruction sequence to execute. Here's one possible implementation of the previous C/C++ code:

```
// Assume expression has been computed and is sitting in X0
// at this point ...

        cmp   x0, #100
        blo   try0_11
        cmp   x0, #103
```

```
        bhi   try1000_1003
        adr   x1, jt100 - 100*8
        ldr   x0, [x1, x0, lsl #3]
        add   x0, x0, x1
        br    x0

jt100:  .dword case100-jt100, case101-jt100
        .dword default-jt100, case103-jt100

try0_11: cmp   x0, #11 // Handle cases 0-11 here.
        bhi   default
        adr   x1, jt0_11
        ldr   x0, [x1, x0, lsl #3]
        add   x0, x0, x1
        br    x0

jt0_11:  .dword case0-jt0_11, case1-jt0_11, case2-jt0_11
        .dword default-jt0_11, default-jt0_11
        .dword default-jt0_11, default-jt0_11
        .dword default-jt0_11, default-jt0_11
        .dword default-jt0_11, case10-jt0_11, case11-jt0_11

try1000_1003:
        cmp   x0, #1000
        blo   default
        cmp   x0, #1003
        bhi   default
        adr   x1, jt1000 - 1000*8
        ldr   x0, [x1, x0, lsl #3]
        add   x0, x0, x1
        br    x0
jt1000:  .dword case1000-jt1000, case1001-jt1000
        .dword default-jt1000, case1003-jt1000
          .
          .
          .
  Code for the actual cases here
```

This code sequence combines groups 0 to 2 and 10 to 11 into a single group (requiring seven additional jump-table entries) in order to save having to write an additional jump-table sequence. For a set of cases this simple, it's easier to just use compare-and-branch sequences, but I've simplified this example to demonstrate multiple jump tables.

7.6.7.6 Other switch Statement Alternatives

What happens if the cases are too sparse to do anything but compare the expression's value case by case? In this situation, the code is not necessarily doomed to being translated into the equivalent of an `if...elseif...else...endif` sequence. However, before considering other alternatives, remember that not all `if...elseif...else...endif` sequences are created equal. Look back at the last example in the previous section (the sparse `switch`

statement). A straightforward implementation might have been something like this:

```
if(unsignedExpression <= 11)
{
  Switch for 0 to 11.
}
else if(unsignedExpression >= 100 && unsignedExpression <= 103)
{
  Switch for 100 to 103.
}
else if(unsignedExpression >= 1000 && unsignedExpression <= 1003)
{
  Switch for 1000 to 1003.
}
else
{
  Code for default case
}
```

Instead, the former implementation first tests against the value 100 and branches based on the comparison being less than (cases 0 to 11) or greater than (cases 1000 to 1001), effectively creating a small *binary search* that reduces the number of comparisons. It's hard to see the savings in the HLL code, but in assembly code you can count the number of instructions that would be executed in the best and worst cases and see an improvement over the standard *linear search* approach of simply comparing the values in the cases in the order they appear in the `switch` statement. (Of course, if you have many groups in a sparse `switch` statement, a binary search will be much faster, on average, than a linear search.)

If your cases are too sparse (no meaningful groups at all), such as the 1, 10, 100, 1,000, 10,000 example given in section 7.6.7.5, "Sparse Jump Tables," on page 399, you can't reasonably implement the `switch` statement by using a jump table. Rather than devolving into a straight linear search, which can be slow, a better solution is to sort your cases and test them by using a binary search.

With a *binary search*, you first compare the expression value against the middle case value. If it's less than the middle value, you repeat the search on the first half of the list of values; if it's greater than the middle value, you repeat the test on the second half of the values; if it's equal, obviously you drop into the code to handle that test. The following code shows the binary search version of the 1, 10, 100, . . . example:

```
// Assume expression has been calculated into X0.

        cmp x0, #100
        blo try1_10
        bhi try1000_10000

    Code to handle case 100

        b.al AllDone
```

```
try1_10:
        cmp x0, #1
        beq case1
        cmp x0, #10
        bne defaultCase

    Code to handle case 10

        b.al AllDone
case1:
    Code to handle case 1

        b.al AllDone

try1000_10000:
        cmp x0, #1000
        beq case1000
        mov x1, #10000   // cmp can't handle 10000.
        cmp x0, x1
        bne defaultCase

    Code to handle case 10,000

        b.al AllDone

case1000:

    Code to handle case 1,000

        b.al AllDone

defaultCase:

    Code to handle defaultCase

AllDone:
```

The techniques presented in this section have many possible alternatives. For example, one common solution is to create a table containing a set of records, with each record entry a two-tuple containing a case value and a jump address. Rather than having a long sequence of compare instructions, a short loop can sequence through all the table elements, searching for the case value and transferring control to the corresponding jump address if a match occurs. This scheme is slower than the other techniques in this section, but it can be much shorter than the traditional `if...elseif...else...endif` implementation. With a little effort, you could use a binary search if the table is sorted.

7.6.7.7 Jump-Table Size Reductions

All the `switch` statement examples up to this point have used double-word arrays for the jump table. With a 64-bit offset, these jump tables can

transfer control to any location in the ARM's address space. In reality, this range is almost never necessary. Most offsets will be relatively small numbers (often less than ±128, or ±32,767). This means that the HO bits of the jump-table entries will likely be all 0s or all 1s (if the offset is negative). With a slight modification to the instructions that transfer control through the jump table, cutting the size of the table in half is easy:

```
        adr  x1, JmpTbl
        ldr  w0, [x1, x0, lsl #2]   // X4 for 32-bit entries
        add  x0, x1, w0, sxtw       // Sign-extend W0 to 64 bits.
        br   x0

JmpTbl: .word Stmt1-JmpTbl, ...
```

This example has three modifications to the other examples in this chapter:

- The scaled-indexed addressing mode (`ldr` instruction) scales the index (in X0) by 4 instead of 8 (because we're accessing elements of a word array rather than a dword array).
- The `add` instruction sign-extends W0 to 64 bits before adding the value with X1.
- The jump table contains word entries instead of dword entries.

This modification limits the range of the case labels to ±2GB around the jump table, rather than the full 64-bit address space—hardly a limitation for most programs. In exchange for this limit, the jump table is now half its original size.

Before you get the sneaky idea of reducing the size of the table entries to 16 bits (giving you a ±32K range), be aware that neither macOS's nor Linux's object code format—Mach-O and the Executable Linkable Format (ELF), respectively—supports 16-bit relocatable offsets; 32-bit offsets are the best you can do.

7.7 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the state machine. In basic terms, a *state machine* is a piece of code that keeps track of its execution history by entering and leaving certain states. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned `goto` statement. Certain variants of C, such as GNU's GCC from the Free Software Foundation, provide similar features. In assembly language, the indirect jump can implement state machines.

In one sense, all programs are state machines. The CPU registers and values in memory constitute the state of that machine. However, this chapter uses a much more constrained definition. For most purposes, only a single variable (or the value in the PC register) will denote the current state.

For a concrete example of a state machine, suppose you have a procedure and want to perform one operation the first time you call it, a different operation the second time you call it, something else the third time you call it, and then something new again on the fourth call. After the fourth call, the code repeats these four operations in order. For example, say you want the procedure to add W0 and W1 the first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as shown in Listing 7-4.

```
// Listing7-4.S
//
// A simple state machine example


#include    "aoaa.inc"

❶ #define     state   x19

            .code
            .extern printf

ttlStr:     wastr   "Listing 7-4"
fmtStr0:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=5, W21=6\n"

fmtStr0b:   .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=1, W21=2\n"

fmtStrx:    .ascii  "Back from StateMachine, "
            wastr   "state=%d, W20=%d\n"

fmtStr1:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=50, W21=60\n"

fmtStr2:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=10, W21=20\n"

fmtStr3:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=50, W21=5\n"

// getTitle
//
// Return pointer to program title
// to the C++ code.

            proc    getTitle, public
            adr     x0, ttlStr
            ret
            endp    getTitle

// State machine is a leaf procedure. Don't bother
// to save LR on stack.
//
```

```
// Although "state" is technically a nonvolatile
// register, the whole point of this procedure
// is to modify it, so we don't preserve it.
// Likewise, X20 gets modified by this code,
// so it doesn't preserve its value either.

            proc    StateMachine
            cmp     state, #0
            bne     TryState1

// State 0: Add W21 to W20 and switch to state 1:

            add     w20, w20, w21
            add     state, state, #1  // State 0 becomes state 1.
            b.al    exit

TryState1:
            cmp     state, #1
            bne     TryState2

// State 1: Subtract W21 from W20 and switch to state 2:

            sub     w20, w20, w21
            add     state, state, 1   // State 1 becomes state 2.
            b.al    exit

TryState2:  cmp     state, #2
            bne     MustBeState3

// If this is state 2, multiply W21 by W20 and switch to state 3:

            mul     w20, w20, w21
            add     state, state, #1  // State 2 becomes state 3.
            b.al    exit

// If it isn't one of the preceding states, we must be in
// state 3, so divide W20 by W21 and switch back to state 0.

MustBeState3:
            sdiv    w20, w20, w21
            mov     state, #0        // Reset the state back to 0.

exit:       ret
            endp    StateMachine

//////////////////////////////////////////////////////
//
// Here's the asmMain procedure:

            proc    asmMain, public

            locals  am
            dword   saveX19
            dword   saveX2021
```

```
            byte    stackSpace, 64
            endl    am

            enter   am.size

// Save nonvolatile registers and initialize
// them to point at xArray, yArray, zArray,
// tArray, aArray, and bArray:

          ❷ str     state, [fp, #saveX19]
            stp     x20, x21, [fp, #saveX2021]
          ❸ mov     state, #0

// Demonstrate state 0:

            lea     x0, fmtStr0
            mov     x1, state
            mstr    x1, [sp]
            bl      printf

            mov     x20, #5
            mov     x21, #6
            bl      StateMachine

          ❹ lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 1:

            lea     x0, fmtStr1
            mov     x1, state
            bl      printf

            mov     x20, #50
            mov     x21, #60
            bl      StateMachine

          ❺ lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 2:

            lea     x0, fmtStr2
            mov     x1, state
            mstr    x1, [sp]
            bl      printf
```

```
            mov     x20, #10
            mov     x21, #20
            bl      StateMachine

          ❻ lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 3:

            lea     x0, fmtStr3
            mov     x1, state
            mstr    x1, [sp]
            bl      printf

            mov     x20, #50
            mov     x21, #5
            bl      StateMachine

          ❼ lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate back in state 0:

            lea     x0, fmtStr0b
            mov     x1, state
            mstr    x1, [sp]
            bl      printf

            mov     x20, #1
            mov     x21, #2
            bl      StateMachine

          ❽ lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Restore nonvolatile register values
// and return.

            ldr     state, [fp, #saveX19]
            ldp     x20, x21, [fp, #saveX2021]
            leave   // Return to C/C++ code.
            endp    asmMain
```

This code uses X19 to maintain the `state` variable ❶. The main program preserves X19 (and X20) ❷ and then initializes the state machine to state 0 ❸. The code then makes successive calls to the state machine functions and prints the results from state 0 ❹, 1 ❺, 2 ❻, and 3 ❼. After executing in state 3, the code returns to state 0 and prints the result ❽.

Here's the `build` command and program output:

```
$ ./build Listing7-4
$ ./Listing7-4
Calling Listing7-4:
Calling StateMachine, state=0, W20=5, W21=6
Back from StateMachine, state=1, W20=11
Calling StateMachine, state=1, W20=50, W21=60
Back from StateMachine, state=2, W20=-10
Calling StateMachine, state=2, W20=10, W21=20
Back from StateMachine, state=3, W20=200
Calling StateMachine, state=3, W20=50, W21=5
Back from StateMachine, state=0, W20=10
Calling StateMachine, state=0, W20=1, W21=2
Back from StateMachine, state=1, W20=3
Listing7-4 terminated
```

Technically, the `StateMachine` procedure is not the state machine. Instead, the variable `state` and the `cmp/bne` instructions constitute the state machine. The procedure is little more than a `switch` statement implemented via the `if...then...elseif` construct. The only unique thing is that it remembers how many times it has been called (or rather, how many times, modulo 4, it has been called) and behaves differently depending on the number of calls.

While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The astute reader may recognize that this code could be made a little faster by using an actual `switch` statement rather than the `if...then...elseif...endif` implementation. However, an even better solution exists.

It's common to use an indirect jump to implement a state machine in assembly language. Rather than having a `state` variable that contains a value like 0, 1, 2, or 3, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests needed to select the proper code fragment. Consider the implementation in Listing 7-5 using the indirect jump.

```
// Listing7-5.S
//
// An indirect jump state machine example

#include    "aoaa.inc"

#define     state   x19
```

```
            .code
            .extern printf

ttlStr:     wastr   "Listing 7-5"
fmtStr0:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=5, W21=6\n"

fmtStr0b:   .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=1, W21=2\n"

fmtStrx:    .ascii  "Back from StateMachine, "
            wastr   "state=%d, W20=%d\n"

fmtStr1:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=50, W21=60\n"

fmtStr2:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=10, W21=20\n"

fmtStr3:    .ascii  "Calling StateMachine, "
            wastr   "state=%d, W20=50, W21=5\n"

// getTitle
//
// Return pointer to program title
// to the C++ code.

            proc    getTitle, public
            adr     x0, ttlStr
            ret
            endp    getTitle

// State machine is a leaf procedure. Don't bother
// to save LR on stack.
//
// Although "state" is technically a nonvolatile
// register, the whole point of this procedure
// is to modify it, so we don't preserve it.
// Likewise, x20 gets modified by this code,
// so it doesn't preserve its value either.

            proc    StateMachine
          ❶ br      state   // Transfer control to current state.

// State 0: Add W21 to W20 and switch to state 1:

state0:
            add   w20, w20, w21
          ❷ adr   state, state1   // Set next state.
            ret

// State 1: Subtract W21 from W20 and switch to state 2:

state1:
            sub   w20, w20, w21
```

```
            adr   state, state2   // Switch to state 2.
            ret


// If this is state 2, multiply W21 by W20 and switch to state 3:

state2:
            mul   w20, w20, w21
            adr   state, state3   // Switch to state 3.
            ret

// If it isn't one of the preceding states, we must be in
// state 3, so divide W20 by W21 and switch back to state 0.

state3:
            sdiv    w20, w20, w21
            adr     state, state0
            ret
            endp    StateMachine

////////////////////////////////////////////////////////
//
// Here's the asmMain procedure:

            proc    asmMain, public

            locals  am
            dword   saveX19
            dword   saveX2021
            byte    stackSpace, 64
            endl    am

            enter   am.size

// Save nonvolatile registers and initialize
// them to point at xArray, yArray, zArray,
// tArray, aArray, and bArray:

            str     state, [fp, #saveX19]
            stp     x20, x21, [fp, #saveX2021]

// Initialize state machine:

          ❸ adr     state, state0

// Demonstrate state 0:

            lea     x0, fmtStr0
            mov     x1, state
            mstr    x1, [sp]
            bl      printf
```

```
            mov     x20, #5
            mov     x21, #6
            bl      StateMachine

            lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 1:

            lea     x0, fmtStr1
            mov     x1, state
            bl      printf

            mov     x20, #50
            mov     x21, #60
            bl      StateMachine

            lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 2:

            lea     x0, fmtStr2
            mov     x1, state
            mstr    x1, [sp]
            bl      printf

            mov     x20, #10
            mov     x21, #20
            bl      StateMachine

            lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate state 3:

            lea     x0, fmtStr3
            mov     x1, state
            mstr    x1, [sp]
            bl      printf
```

```
            mov     x20, #50
            mov     x21, #5
            bl      StateMachine

            lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Demonstrate back in state 0:

            lea     x0, fmtStr0b
            mov     x1, state
            mstr    x1, [sp]
            bl      printf

            mov     x20, #1
            mov     x21, #2
            bl      StateMachine

            lea     x0, fmtStrx
            mov     x1, state
            mov     x2, x20
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

// Restore nonvolatile register values
// and return:

            ldr     state, [fp, #saveX19]
            ldp     x20, x21, [fp, #saveX2021]
            leave   // Return to C/C++ code.
            endp    asmMain
```

This code has the same structure as Listing 7-4. The main difference is that this code assumes that the target address of the state machine is in X19 rather than a state number.

The `br` instruction at the beginning of the `StateMachine` procedure ❶ transfers control to the location pointed at by the `state` variable (X19). The first time you call `StateMachine`, it points at the `state0` label. Thereafter, each subsection of code sets the `state` variable to point at the appropriate successor code. Within each state of the state machine ❷, the code sets X19 to the address of the next entry point of the state machine (rather than setting a state number). The main program initializes the state machine with the address of the `State0` label ❸ rather than the value 0. Otherwise, this main program is largely the same as in Listing 7-4.

Here's the `build` command and program output:

```
$ ./build Listing7-5
$ ./Listing7-5
Calling Listing7-5:
Calling StateMachine, state=4196420, W20=5, W21=6
Back from StateMachine, state=4196432, W20=11
Calling StateMachine, state=4196432, W20=50, W21=60
Back from StateMachine, state=4196444, W20=-10
Calling StateMachine, state=4196444, W20=10, W21=20
Back from StateMachine, state=4196456, W20=200
Calling StateMachine, state=4196456, W20=50, W21=5
Back from StateMachine, state=4196420, W20=10
Calling StateMachine, state=4196420, W20=1, W21=2
Back from StateMachine, state=4196432, W20=3
Listing7-5 terminated
```

This output demonstrates that Listing 7-5 behaves in a manner similar to Listing 7-4.

7.8 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) that make up a typical program. Most HLLs have implied loop structures hidden away. For example, consider the BASIC statement `if A$ = B$ then 100`. This `if` statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in `A$` to the corresponding character in `B$` and then jump to statement 100 if and only if all the characters matched. (The C stdlib provides the `strcmp` routine that compares the strings for you, effectively hiding the loop. However, if you were to write this function yourself, the looping nature of the operation would be obvious.)

Program loops consist of three components: an optional initialization component, an optional loop-termination test, and the body of the loop. The order in which you assemble these components can dramatically affect the loop's operation. Three permutations of these components appear frequently in programs: `while` loops, `repeat...until` loops (`do...while` in C/C++), and infinite loops (for example, `for(;;)` in C/C++). This section covers those three loop types along with C-style `for` loops (definite loops), register usage in loops, and breaking out of loops.

7.8.1 while

The most generic loop is the `while` loop. In C/C++, it takes the following form:

```
while(expression) statement(s);
```

In the `while` loop, the termination test appears at the beginning of the loop. As a direct consequence of the position of the termination test, the

body of the loop may never execute if the Boolean expression is always false.

Consider the following C/C++ `while` loop:

```
i = 0;
while(i < 100)
{
    ++i;
}
```

The `i = 0;` statement is the initialization code for this loop. `i` is a loop-control variable, because it controls the execution of the body of the loop. `i < 100` is the loop-termination condition: the loop will not terminate as long as `i` is less than 100. The single statement `++i;` (increment `i`) is the loop body that executes on each loop iteration.

A C/C++ `while` loop can be easily synthesized using `if` and `goto` statements. For example, you may replace the previous C `while` loop with the following C code:

```
i = 0;
WhileLp:
if(i < 100)
{
    ++i;
      goto WhileLp;
}
```

More generally, you can construct any `while` loop as follows:

```
Optional initialization code

UniqueLabel:
if(not_termination_condition)
{
    Loop body
    goto UniqueLabel;
}
```

Therefore, you can use the techniques from earlier in this chapter to convert `if` statements to assembly language and add a single `b.al` instruction to produce a `while` loop. The example in this section translates to the following pure ARM assembly code:

```
          mov  w0, #0
WhileLp:
          cmp  w0, #100
          bnlt WhileDone
```

```
    Loop body

        add  w0, w0, #1   // ++i
        b.al WhileLp

WhileDone:
```

GCC will actually convert most `while` statements to different ARM code than this section presents. The reason for the difference appears in section 7.9.1, "Moving the Termination Condition to the End of a Loop," on page 428, which explores how to write more efficient loop code.

7.8.2 repeat...until

The `repeat...until` loop, also called the `do...while` loop in C, tests for the termination condition at the end of the loop rather than at the beginning. In Pascal, the `repeat...until` loop takes the following form:

```
Optional initialization code
repeat

    Loop body

until(termination_condition);
```

This is comparable to the following C/C++ `do...while` loop:

```
Optional initialization code
do
{
    Loop body

}while(not_termination_condition);
```

This sequence executes the initialization code, then executes the loop body, and finally tests a condition to see whether the loop should repeat. If the Boolean expression evaluates to false, the loop repeats; otherwise, the loop terminates. In the `repeat...until` loop, the termination test appears at the end of the loop and, as a direct consequence, the loop body always executes at least once.

Like the `while` loop, the `repeat...until` loop can be synthesized with an `if` statement and a `b.al` (branch). The following is an example of just such an implementation:

```
Initialization code
SomeUniqueLabel:

    Loop body

if(not_the_termination_condition) goto SomeUniqueLabel;
```

Based on the material presented in the previous sections, you can easily synthesize `repeat...until` loops in assembly language, as shown in the following simple example:

```
     repeat (* Pascal code *)

          write('Enter a number greater than 100:');
          readln(i);

     until(i > 100);

// This translates to the following if/jmp code:

     RepeatLabel:

          write('Enter a number greater than 100:');
          readln(i);

     if(i <= 100) then goto RepeatLabel;

// It also translates into the following assembly code:

RepeatLabel:

          bl    print
          wastr "Enter a number greater than 100: "
          bl    readInt // Function to read integer from user

          cmp   w0, #100 // Assume readInt returns integer in W0.
          bngt  RepeatLabel
```

The `repeat...until` loop has a slightly more efficient implementation because it combines the loop termination test and the branch back to the beginning of the loop.

7.8.3 forever/endfor

If `while` loops test for termination at the beginning of the loop and `repeat...until` and `do...while` loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. The C/C++ high-level `for(;;)` loop, combined with the `break` statement, provides this capability. The C/C++ infinite loop takes the following form:

```
for(;;)
{
    Loop body

}
```

There is no explicit termination condition. The `for(;;)` construct forms an infinite loop. A `break` statement usually handles loop termination. Consider the following C++ code that employs a `for(;;)` construct:

```
for(;;)
{
     cin >> character;
     if(character == '.') break;
     cout << character;

}
```

Converting a `for(;;)` loop to pure assembly language is easy: all you need is a label and a `b.al` instruction. The `break` statement in this example is also nothing more than a `b.al` instruction (or conditional jump). The pure assembly language version of the preceding code looks something like the following:

```
foreverLabel:

          bl   getchar    // Assume it returns char in W0.
          cmp  w0, #'.'
          beq  ForIsDone

          bl   putcchar   // Assume this prints the char in W0.
          b.al foreverLabel

ForIsDone:
```

As you can see, the forever loop has a very simple implementation.

7.8.4 for

The standard `for` loop is a special form of the `while` loop that repeats the loop body a specific number of times, which is known as a *definite* loop. In C/C++, the `for` loop takes the following form:

```
for(Initialization_Stmt; Termination_Expression; inc_Stmt)
{
    statements

}
```

This is equivalent to the following:

```
Initialization_Stmt;
while(Termination_Expression)
{
    statements

    inc_Stmt;

}
```

Traditionally, programs use the `for` loop to process arrays and other objects accessed in sequential order. You normally initialize a loop-control

variable with the initialization statement, then use the loop-control variable as an index into the array (or other data type), as shown in the following example:

```
for(i = 0; i < 7; ++i)
{
    printf("Array Element = %d /n", SomeArray[i]);

}
```

To convert this to pure assembly language, begin by translating the `for` loop into an equivalent `while` loop:

```
i = 0;
while(i < 7)
{
    printf("Array Element = %d \n", SomeArray[i]);
    ++i;
}
```

Now, using the techniques from section 7.8.1, "`while`," on page 415, translate the code into pure assembly language:

```
          mov  x19, #0               // Use X19 to hold loop index.
WhileLp:  cmp  x19, #7
          bnlt EndWhileLp

          lea  x0, fmtStr            // fmtStr = "Array Element = %d\n"
          lea  x1, SomeArray
          ldr  w1, [x1, x19, lsl #2] // SomeArray is word array.
          mstr x1, [sp]
          bl   printf

          add  x19, x19, #1   // ++i
          b.al WhileLp;

EndWhileLp:
```

This is a fairly efficient implementation of a `while` loop in assembly language, though for `for` loops that execute a fixed number of times, you might consider using the `cbnz` instruction (see section 7.8.6, "ARM Looping Instructions," on page 425).

7.8.5 break and continue

The C/C++ `break` and `continue` statements both translate into a single `b.al` instruction. The `break` statement exits the loop that immediately contains the `break` statement; the `continue` statement restarts the loop that contains the `continue` statement.

To convert a `break` statement to pure assembly language, just emit a `goto`/`b.al` instruction that transfers control to the first statement following

the end of the loop to exit. You can do this by placing a label after the loop body and jumping to that label. The following code fragments demonstrate this technique for the various loops:

```
// Breaking out of a FOR(;;) loop:

for(;;)
{
      stmts

      // break;
      goto BreakFromForever;

      stmts
}
BreakFromForever:

// Breaking out of a FOR loop:

for(initStmt; expr; incStmt)
{
      stmts

      // break;
      goto BrkFromFor;

      stmts
}
BrkFromFor:

// Breaking out of a WHILE loop:

while(expr)
{
      stmts

      // break;
      goto BrkFromWhile;

      stmts
}
BrkFromWhile:

// Breaking out of a REPEAT...UNTIL loop (do...while is similar):

repeat
      stmts

      // break;
      goto BrkFromRpt;

      stmts
until(expr);
BrkFromRpt:
```

In pure assembly language, convert the appropriate control structures to assembly and replace the `goto` with a `b.al` instruction.

The `continue` statement is slightly more complex than the `break` statement. The implementation is still a single `b.al` instruction; however, the target label doesn't wind up going in the same spot for each of the loops. Figures 7-2 through 7-5 show where the `continue` statement transfers control for each of the loops.

Figure 7-2 shows the `for(;;)` loop with a `continue` statement.

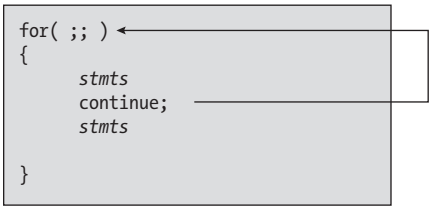


Figure 7-2: The `continue` destination and the `for(;;)` loop

Figure 7-3 shows the `while` loop with a `continue` statement.

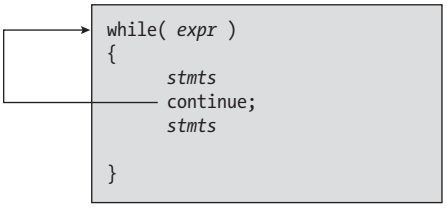


Figure 7-3: The `continue` destination and the `while` loop

Figure 7-4 shows a C/C++ `for` loop with a `continue` statement.

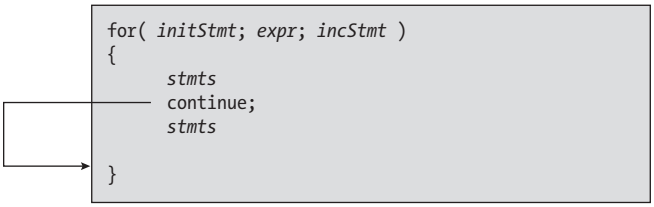


Figure 7-4: The `continue` destination and the `for` loop

Note in Figure 7-4 that the `continue` statement forces the execution of `incStmt` and then transfers control to the test for loop termination.

Figure 7-5 shows a `repeat...until` loop with a `continue` statement.

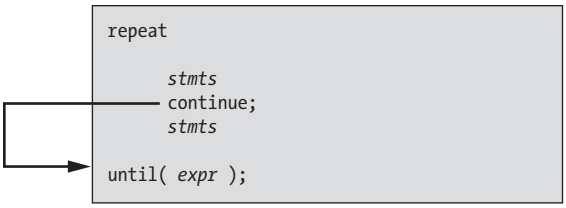


Figure 7-5: The `continue` destination and the `repeat...until` loop

The following code fragments demonstrate how to convert the `continue` statement into an appropriate `b.al` instruction for each of these loop types:

```
// for(;;)/continue/endfor
// Conversion of forever loop with continue
// to pure assembly:
// for(;;)
// {
//      stmts
//      continue;
//      stmts
// }
//
// Converted code:

foreverLbl:

      stmts

      // continue;
      b.al foreverLbl

      stmts

      b.al foreverLbl

// while/continue/endwhile
// Conversion of while loop with continue
// into pure assembly:
//
// while(expr)
// {
//      stmts
//      continue;
//      stmts
// }
//
// Converted code:

whlLabel:

  Code to evaluate expr
```

```
      bcc EndOfWhile      // Skip loop on expr failure.

      stmts

      // continue;
      b.al whlLabel      // Jump to start of loop on continue.

      stmts

      b.al whlLabel      // Repeat the code.
EndOfWhile:


// for/continue/endfor
// Conversion for a for loop with continue
// into pure assembly:
//
// for(initStmt; expr; incStmt)
// {
//      stmts
//      continue;
//      stmts
// }
//
// Converted code:

       initStmt
ForLpLbl:

  Code to evaluate expr

       bcc EndOfFor     // Branch if expression fails.

       stmts

       // continue;
       b.al ContFor     // Branch to incStmt on continue.

       stmts

ContFor:

       incStmt

       b.al ForLpLbl

EndOfFor:


// repeat...continue...until
// repeat
//      stmts
//      continue;
//      stmts
```

```
// until(expr);
//
// do
// {
//      stmts
//      continue;
//      stmts
//
// }while(!expr);
//
// Converted code:

RptLpLbl:

      stmts

      // continue;
      b.al ContRpt     // Continue branches to termination test.

      stmts

ContRpt:

  Code to test expr

      bcc RptLpLbl     // Jumps if expression evaluates false.
```

In each case, the `b.al` instruction transfers control to the point in the loop where it will test the loop condition and increment the loop control variable (for `for` loops), or to the beginning of the loop's body.

7.8.6 ARM Looping Instructions

The ARM CPU provides four machine instructions that are useful for creating loops. These instructions violate the RISC principle of "an instruction does only one thing," but they are quite handy even if they are a little "CISCy."

The first two instructions test a register's value and branch if that register is equal to, or not equal to, 0. The two instructions are `cbz` (compare and branch if zero) and `cbnz` (compare and branch if not zero). Their syntax is

```
cbz  wn, label
cbz  xn, label
cbnz wn, label
cbnz xn, label
```

where X*n* and W*n* are the register to compare against 0 and *label* is a statement label within ±1MB of the current instruction.

These instructions are equivalent to the following:

```
cmp wn, wzr   // cbz wn, label
beq label
```

```
cmp xn, xzr   // cbz xn, label
beq label

cmp wn, wzr   // cbnz wn, label
bne label

cmp xn, xzr   // cbnz xn, label
bne label
```

Another useful pair of instructions are `tbz` (test bit for 0 and branch) and `tbnz` (test bit for not 0 and branch). These instructions test a bit in a register and branch based on that bit's value (0 or nonzero). The syntax for these instructions is

```
tbz  wn, #imm6, label
tbz  xn, #imm6, label
tbnz wn, #imm6, label
tbnz xn, #imm6, label
```

where X*n* and W*n* are the register to test, imm_6 is a bit number in the range 0–63 for 64-bit registers and 0–31 for 32-bit registers, and *label* is a statement label within ±32KB of the current instruction. The `tbz` instruction branches to the *label* if the specified bit in the register is 0, while the `tbnz` instruction branches if the bit is not 0.

7.8.7 Register Usage and Loops

Given that the ARM accesses registers more efficiently than memory locations, registers are the ideal spot to place loop-control variables (especially for small loops). However, registers are a limited resource, despite the many registers available on the ARM. Unlike with memory, you cannot place much data in the registers.

Loops present a special challenge for registers. Registers are perfect for loop-control variables because they're efficient to manipulate and can serve as indexes into arrays and other data structures (a common use for loop-control variables). However, the limited availability of registers often creates problems when using registers in this fashion. This is especially true if you call other functions/procedures within the loops, which limits you to using nonvolatile registers for loop control variables. Consider the following code with nested loops that will not work properly because it attempts to reuse a register (X19) that is already in use, leading to the corruption of the outer loop's loop-control variable:

```
        mov  w19, #8
loop1:
        mov  w19, #4
loop2:
        stmts
```

```
          subs w19, w19, #1
          bne  loop2

          subs w19, w19, #1
          bne  loop1
```

The intent here was to create a set of nested loops, one loop inside another. The inner loop (`loop2`) should repeat four times for each of the eight executions of the outer loop (`loop1`). Unfortunately, both loops use the same register as a loop-control variable. Therefore, this will form an infinite loop. Because W19 is always 0 upon encountering the second `subs` instruction, control will always transfer to the `loop1` label (because decrementing 0 produces a nonzero result).

The solution is to save and restore the W19 register or to use a different register in place of W19 for the outer loop; the following code demonstrates preserving W19 across the execution of the loop:

```
          mov w19, #8
loop1:
          str  w19, [sp, #-16]!   // Push onto stack.
          mov  w19, #4
loop2:
          stmts

          subs w19, w19, #1
          bne  loop2

          ldr  w19, [sp], #16     // Pop off the stack.
          subs w19, w19, #1       // Decrement W19.
          bne  loop1

or

          mov  w19,#8
loop1:
          mov  w20, #4
loop2:
          stmts

          subs w20, w20, #1
          bne  loop2

          subs w19, w19, #1
          bne  loop1
```

Register corruption is one of the primary sources of bugs in loops in assembly language programs, so always keep an eye out for this problem.

Until this point, this chapter has mainly focused on the correct implementation of various types of loops in assembly language. The next section begins discussing how to write loops efficiently in assembly language.

7.9 Loop Performance Improvements

Because loops are the primary source of performance problems within a program, they are also the place to look when attempting to speed up your software. A treatise on how to write efficient programs is beyond the scope of this chapter, but you should be aware of the following concepts when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute a single iteration of the loop.

7.9.1 Moving the Termination Condition to the End of a Loop

As you may have noticed, the `repeat...until` loop is slightly more efficient than a `while` loop. This is because `repeat...until` manages to combine the loop's Boolean test along with the branch back to the beginning of the loop. You can improve the other loops to be slightly more efficient. Consider the following flow graphs for the three types of loops presented earlier:

```
repeat...until loop:
     Initialization code
          Loop body
     Test for termination and branch back if necessary.
     Code following the loop

while loop:
     Initialization code
     Loop-termination test
          Loop body
          Jump back to test.
     Code following the loop

forever/endfor loop:
     Initialization code
          Loop body part 1
          Loop-termination test
          Loop body part 2
          Jump back to loop body part 1
     Code following the loop
```

The `repeat...until` loop is the simplest of the bunch. This is reflected in the assembly language implementation of these loops. Consider the following semantically identical `repeat...until` and `while` loops:

```
// Example involving a while loop:

        mov  w0, w1
        sub  w0, w0, #20

// while(W0 <= W1)
```

```
whileLp: cmp  w0, w1
         bnle endwhile

         stmts

         add  w0, w0, #1
         b.al whileLp
endwhile:

// Example involving a repeat...until loop:

         mov  w0, w1
         sub  w0, w0, #20
repeatLp:

         stmts

         add  w0, w0, #1
         cmp  w0, w1
         bngt repeatLp
```

Testing for the termination condition at the end of the loop allows you to remove a `b.al` instruction from the loop, which can be significant if the loop is nested inside other loops. Given the definition of the loop, you can easily see that the loop will execute exactly 20 times, which suggests that the conversion to a `repeat...until` loop is trivial and always possible. Unfortunately, it's not always quite this easy.

Consider the following C code:

```
while( w0 <= w1 )
{
    stmts

    ++w0;
}
```

In this example, you don't know what W0 contains upon entry into the loop. Therefore, you cannot assume that the loop body will execute at least once. This means you must test for loop termination before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single `b.al` instruction:

```
          b.al WhlTest
TopOfLoop:

          stmts

          add  w0, w0, #1
WhlTest:  cmp  w0, w1
          ble TopOfLoop
```

Although the code is as long as the original `while` loop, the `b.al` instruction executes only once rather than on each repetition of the loop. However, the slight gain in efficiency is obtained via a slight loss in readability, so be sure to comment it. The second code sequence is also closer to spaghetti code than the original implementation. Such is often the price of a small performance gain. Carefully analyze your code to ensure that such a performance boost is worth the loss of clarity.

7.9.2 Executing the Loop Backward

Because of the nature of the flags on the ARM, loops that repeat from a number down to (or up to) 0 are more efficient than loops that execute from 0 to another value. Compare the following C/C++ `for` loop and the comparable assembly language code:

```
for(j = 1; j <= 8; ++j)
{
      stmts
}

// Conversion to pure assembly (as well as using a
// REPEAT...UNTIL form):

mov w0, #1     // Assume j = W0.
ForLp:

      stmts

      add w0, w0, #1
      cmp w0, #8
      ble ForLp
```

Now consider another loop that also has eight iterations but runs its loop-control variable from 8 down to 1 rather than from 1 up to 8, thereby saving a comparison on each repetition of the loop:

```
     mov  w0, #8  // Assume j = W0.
LoopLbl:

     stmts

     subs w0, w0, #1
     bne  LoopLbl
```

Saving the execution time of the `cmp` instruction on each iteration of the loop may result in faster code. Unfortunately, you cannot force all loops to run backward. However, with a little effort and some coercion, you should be able to write many `for` loops so that they operate backward.

The previous example worked out well because the loop ran from 8 down to 1. The loop terminated when the loop-control variable became 0. What happens if you need to execute the loop when the loop-control variable goes

to 0? For example, suppose that the preceding loop needed to range from 7 down to 0. As long as the lower bound is nonnegative, you can substitute the `bpl` instruction in place of the `bne` instruction in the earlier code:

```
    mov  w0, #7   // Assume j = W0.
LoopLbl:

    stmts

    subs w0, w0, #1
    bpl  LoopLbl
```

This loop will repeat eight times, with W0 (`j`) taking on the values 7 to 0. When W0 decrements 0 to –1, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but are actually negative. If the loop-control variable is a word, values in the range 2,147,483,648 to 4,294,967,295 are negative in the two's complement system. Therefore, initializing the loop-control variable with any 32-bit value in this range (or, of course, 0) terminates the loop after a single execution. This can get you into trouble if you're not careful.

7.9.3 Eliminating Loop-Invariant Calculations

A *loop-invariant computation* is a calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop but can instead compute them outside the loop and reference the value of the computations inside the loop. The following C code demonstrates an invariant computation:

```
for(i = 0; i < n; ++i)
{
    k = (j - 2) + i
    Other code
}
```

Because `j` never changes throughout the execution of this loop, the subexpression `j - 2` can be computed outside the loop:

```
jm2 = j - 2;
for(i = 0; i < n; ++i)
{

    k = jm2 + i;
    Other code
}
```

This translates to the following assembly code, which moves the invariant calculation outside the loop:

```
     ldr  w19, [fp, #j]
     sub  w19, w19, #2
```

```
      mov  w20, #0         // Assume W20 = i.
lp:   cmp  w20, #n
      bnlt loopDone
      add  w21, w19, w20   // k = jm2 + i
      add  w20, w20, #1

      Other code

      b.al lp
loopDone:
```

7.9.4 Unraveling Loops

For small loops—those whose bodies are composed of just a few statements—the overhead required to process the loop may constitute a significant percentage of the total processing time. For example, consider the following Pascal code and its associated ARM assembly language code:

```
     for i := 3 downto 0 do A[i] := 0;

          mov  w19, #3       // Assume i = W19.
          add  x20, fp, #A   // LEA X20,A, assuming A is local.
LoopLbl:
          str  wzr, [x20, x19, lsl #2]
          subs w19, w19, #1
          bpl  LoopLbl
```

Three instructions execute on each repetition of the loop. Only one instruction is doing the desired operation (moving a 0 into an element of `A`). The remaining two instructions control the loop. Therefore, it takes 12 instructions to do the operation logically required by 4.

While we could make many improvements to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing: it's storing four 0s into `A[0]` through `A[3]`. A more efficient approach is to use four `str` instructions to accomplish the same task. For example, if `A` is an array of words, the following code initializes `A` much faster than the preceding code:

```
str  wzr, [fp, #A + 0]
str  wzr, [fp, #A + 4]
str  wzr, [fp, #A + 8]
str  wzr, [fp, #A + 12]
```

Although this is a simple example, it shows the benefit of *loop unraveling* (also known as *loop unrolling*), which consists of repeating the loop's body as source code for each iteration of the loop. If this simple loop appeared buried inside a set of nested loops, the 3:1 instruction reduction could possibly double the performance of that section of your program. (It would be criminal not to mention at this point that you could cut this down to two

instructions by storing XZR, a double word, into A + 0 and A + 8, though that is a different optimization.)

Of course, you cannot unravel all loops. Loops that execute a variable number of times are difficult to unravel because there is rarely a way to determine at assembly time the number of loop iterations. Therefore, unraveling a loop is a process best applied to loops that execute a known number of times, with the number of times known at assembly time.

Even if you repeat a loop a fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions controlling the loop (and handling other overhead operations) represents a significant percentage of the total number of instructions in the loop. Had the previous loop contained 36 instructions in the body (exclusive of the 3 overhead instructions), the performance improvement would be, at best, only 10 percent, compared with the 300 to 400 percent it now enjoys.

Therefore, the costs of unraveling a loop—all the extra code that must be inserted into your program—quickly reach a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

7.9.5 Using Induction Variables

This section introduces optimizations based on induction variables. An *induction variable* is one whose value depends entirely on the value of another variable. Consider the following Pascal loop:

```
for i := 0 to 255 do csetVar[i] := [];
```

Here the program is initializing each element of an array of character sets to the empty set. The straightforward code to achieve this is the following:

```
     str  wzr, [fp, #i]
     lea  x20, csetVar
FLp:

     // Assume that each element of a csetVar
     // array contains 16 bytes (256 bits).

     ldr  w19, [fp, #i]
     lsl  w19, w19, #4     // i * 16 (element size)

     // Set this element to the empty set (all 0 bits).

     str  xzr, [x20, x19]  // Fill in first 8 bytes.
     add  x20, x20, #8
     str  xzr, [x20, x19]  // Initialize second 8 bytes.
     sub  x20, x20, #8
```

```
    ldr  w19, [fp, #i]
    add  w19, w19, #1
    str  w19, [fp, #i]
    cmp  w19, #256     // Quit if at end of array.
    blo  FLp
```

Although unraveling this code will still improve performance, it will take 2,304 instructions to accomplish this task—too many for all but the most time-critical applications. However, you can reduce the execution time of the loop's body by using induction variables.

In the preceding example, the index into the array `csetVar` tracks the loop-control variable; it's always equal to the value of the loop-control variable times 16. Because `i` doesn't appear anywhere else in the loop, there is no sense in performing the computations on `i`. Why not operate directly on the array index value? Furthermore, because the scaled-indexed addressing mode doesn't support an integer offset component, the code is constantly adding 8 to or subtracting 8 from X20 to initialize the second half of each character set element. This computation can also be worked into the induction of the loop control variable. The following code demonstrates this technique:

```
    lea  x20, csetVar
    add  x19, x20, #255 * 16  // Compute array ending address.
FLp:

    // Set current element to the empty set (all 0 bits).

    str  xzr, [x20]       // Fill in first 8 bytes.
    str  xzr, [x20, #8]   // Fill in second 8 bytes.

    add  w20, x20, #16    // Move on to next element.
    cmp  x20, x19
    blo  FLp
```

The induction that takes place in this example occurs when the code initializes the loop control variable with the address of the array (moved into X20 for efficiency) and then increments it by 16 on each iteration of the loop rather than by 1. This allows the code to use the indirect-plus-offset addressing mode (rather than the scaled-indexed addressing mode), as no shift is required. Once the code can use the indirect-plus-offset mode, it can drop the addition and subtraction of the loop control variable in order to access the second half of each character set array element.

7.10 Moving On

After mastering the material in this chapter and the chapters up to this point, you should be capable of translating many HLL programs into assembly code.

This chapter covered several concepts concerning the implementation of loops in assembly language. It discussed statement labels, including working with their addresses, efficiently representing pointers to labels in your programs, using unconditional and indirect branches, working with veneers, and transferring control to statements beyond the range of the ARM branches. It then covered decisions: how to implement `if...then...else...elseif`, `switch` statements, state machines in assembly language, Boolean expressions, and complete/short-circuit evaluation. It also described how to utilize 32-bit PC-relative addresses to reduce jump-table (and pointer) sizes. Finally, this chapter described various kinds of loops, improving loop performance, and the special ARM machine instructions that support loop construction.

You're now prepared to start writing some serious assembly language code. Starting with the next chapter, you'll learn some intermediate assembly language programming that enables you to write code that is difficult or impossible to write in HLLs.

7.11 For More Information

- My book *Write Great Code,* Volume 2, 2nd edition (No Starch Press, 2020) provides a good discussion of the implementation of various HLL control structures in low-level assembly language. It also discusses optimizations such as induction, unrolling, strength reduction, and so on that apply to optimizing loops.

TEST YOURSELF

1. What are the typical mechanisms for obtaining the address of a label appearing in a program?
2. What is the form of the indirect branch instruction?
3. What is a state machine?
4. What is a trampoline?
5. Explain the difference between short-circuit and complete Boolean evaluation.
6. Convert the following `if` statements to assembly language sequences by using complete Boolean evaluation (assume all variables are unsigned 32-bit integer values):

   a.

```
if(x == y || z > t)
{
 Do something.
}
```

(continued)

b.

```
if(x != y && z < t)
{
 then statements
}
else
{
 else statements
}
```

7. Convert the preceding statements (a) and (b) to assembly language by using short-circuit Boolean evaluation, assuming all variables are signed 16-bit integer values.
8. Convert the following `switch` statements to assembly language (assume all variables are unsigned 32-bit integers):

   a.

```
switch(s)
{
 case 0: case 0 code break;
 case 1: case 1 code break;
 case 2: case 2 code break;
 case 3: case 3 code break;
}
```

   b.

```
switch(t)
{
 case 2: case 2 code break;
 case 4: case 4 code break;
 case 5: case 5 code break;
 case 6: case 6 code break;
 default: default code
}
```

9. Convert the following `while` loops to assembly code (assume all variables are signed 32-bit integers):

   a.

```
while(i < j)
{
 Code for loop body
}

do
{
```

```
 Code for loop body
} while(i != j);
```

b.

```
do
{
 Code for loop body, part a
 if(m != 5) continue;
 Code for loop body, part b
 if(n == 6) break;
 Code for loop body, part c
} while(i < j && k > j);
```

c.

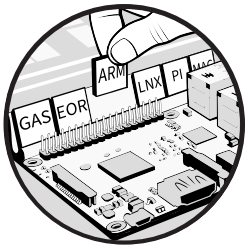
```
for(int i = 0; i < 10; ++i)
{
 Code for loop body
}
```

PART III

ADVANCED ASSEMBLY LANGUAGE

8

ADVANCED ARITHMETIC



This chapter covers extended-precision arithmetic and arithmetic on operands of different sizes. By the end of this chapter, you should know how to apply arithmetic and logical operations to integer operands of any size, including those larger than 64 bits, and how to convert operands of different sizes into a compatible format.

8.1 Extended-Precision Operations

Assembly language does not limit the size of integer operations, a major advantage over HLLs (which typically rely on functions, written in assembly language, to handle extended-precision arithmetic). For example, the standard C programming language defines four integer sizes: `short int`, `int`, `long int`, and `long long int`. On the PC, these are often 16-, 32-, and 64-bit integers.

Although the ARM machine instructions limit you to processing 32- or 64-bit integers with a single instruction, you can use multiple instructions

to process integers of any size. This section describes how to extend various arithmetic and logical operations from 32 or 64 bits to as many bits as you please.

8.1.1 Addition

The ARM `add/adds` instruction adds two 32- or 64-bit numbers. After the execution of `adds`, the ARM carry flag is set if you have an overflow out of the HO bit of the sum. You can use this information to do extended-precision addition operations. (This book uses *multidigit* and *multibyte* as synonyms for *extended precision*.)

Consider the way you manually perform a multidigit addition operation, as shown in Figure 8-1.

Step 1: Add the least significant digits together

```
  289                          289
+ 456          produces      + 456
                               5 with carry 1
```

Step 2: Add the next significant digits plus carry

```
  1  (carry)                   1  (carry)
  289                          289
+ 456          produces      + 456
   5                            45 with carry 1
```

Step 3: Add the most significant digits together

```
  1  (carry)                   1  (carry)
  289                          289
+ 456          produces      + 456
   45                           745
```

Figure 8-1: Multidigit addition

The ARM handles extended-precision arithmetic the same way, except instead of adding the numbers a digit at a time, it adds them together a word or double word at a time, breaking a larger operation into a sequence of smaller ones. For example, consider the three-double-word (192-bit) addition operation in Figure 8-2.

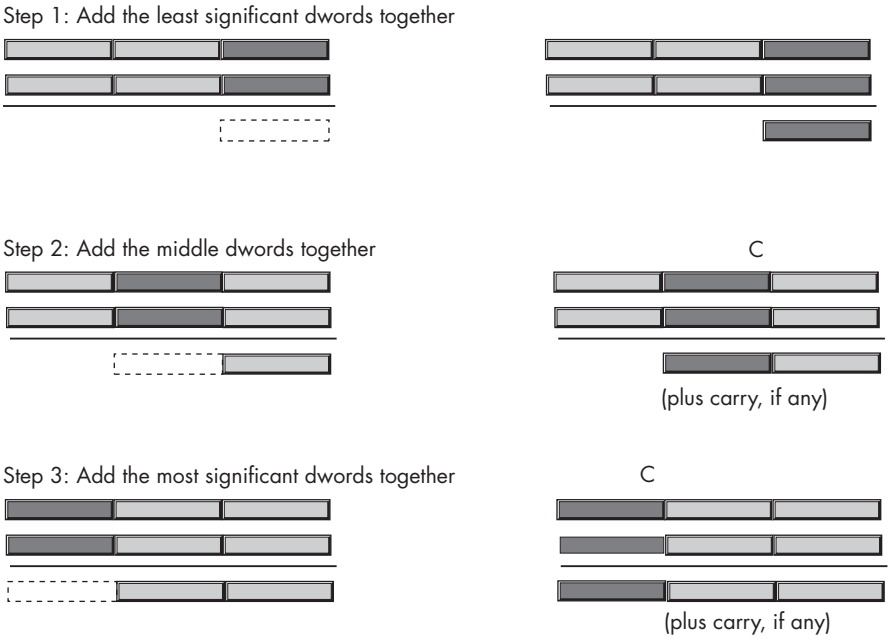


Figure 8-2: Adding two 192-bit objects together

Since the ARM processor family is capable of adding together at most 64 bits at a time (using general-purpose registers), the operation must proceed in blocks of 64 bits or fewer, according to the following steps:

1. Add the two LO double words together just as you would add the two LO digits of a decimal number together in the manual algorithm, using the `adds` instruction. If there is a carry out of the LO addition, `adds` sets the carry flag to 1. Otherwise, it clears the carry flag.
2. Add together the second pair of double words in the two 192-bit values, plus the carry out of the previous addition (if any), using the `adcs` (add with carry) instruction. The `adcs` instruction uses the same syntax as `adds` and performs almost the same operation:

   ```
   adcs dest, source1, source2 // dest := source1 + source2 + C
   ```

   The only difference is that `adcs` adds in the value of the carry flag along with the source operands. It sets the flags the same way `adds` does (including setting the carry flag if there is an unsigned overflow). This is exactly what we need in order to add together the middle two double words of our 192-bit sum.
3. Add the HO double words of the 192-bit value with the carry out of the sum of the middle two quad words by using `adcs`. (You could also use a plain `adc` instruction if you don't need the flag settings after the instruction.)

To summarize, the adds instruction adds the LO double words together, and adcs adds all other double-word pairs together. At the end of the extended-precision addition sequence, the carry flag indicates unsigned overflow (if set), a set overflow flag indicates signed overflow, and the sign flag indicates the sign of the result. The zero flag doesn't have any real meaning at the end of the extended-precision addition; it simply means that the sum of the two HO double words is 0 and does not indicate that the whole result is 0.

For example, suppose you have two 128-bit values you wish to add together, defined as follows:

```
        .data
X:      .qword   0
Y:      .qword   0
```

Say you want to store the sum in a third variable Z, which is also a qword. The following ARM code will accomplish this task:

```
lea   x0, X
ldr   x3, [x0]    // Add together the LO 64 bits
lea   x1, Y       // of the numbers and store the
ldr   x4, [x1]    // result into the LO dword of Z.
adds  x5, x3, x4
lea   x2, Z
str   x5, [x2]

ldr   x3, [x0, #8]    // Add together the HO 64 bits (with
ldr   x4, [x1, #8]    // carry) and store the result into
adcs  x5, x3, x4      // the HO dword of Z.
str   x5, [x2, #8]
```

The first seven instructions add the LO double words of X and Y together and store the result into the LO double word of Z. The last four instructions add the HO double words of X and Y together, along with the carry from the LO word, and store the result in the HO double word of Z.

You can extend this algorithm to any number of bits by using adcs to add in the higher-order values. For example, to add together two 256-bit values declared as arrays of four double words, you could use code like the following:

```
        .data
BigVal1: .space  4*8        // Array of four double words
BigVal2: .space  4*8
BigVal3: .space  4*8        // Holds the sum
     .
     .
     .
    lea  x0, BigVal1
    lea  x1, BigVal2
    lea  x2, BigVal3
```

```
    ldr  x4, [x0]          // BigVal1[0]
    ldr  x5, [x1]          // BigVal2[0]
    adds x6, x4, x5
    str  x6, [x2]          // BigVal3[0]

    ldr  x4, [x0, #8]      // BigVal1[1]
    ldr  x5, [x1, #8]      // BigVal2[1]
    adcs x6, x4, x5
    str  x6, [x2, #8]      // BigVal3[1]

    ldr  x4, [x0, #16]     // BigVal1[2]
    ldr  x5, [x1, #16]     // BigVal2[2]
    adcs x6, x4, x5
    str  x6, [x2, #16]     // BigVal3[2]

    ldr  x4, [x0, #24]     // BigVal1[3]
    ldr  x5, [x1, #24]     // BigVal2[3]
    adcs x6, x4, x5
    str  x6, [x2, #24]     // BigVal3[3]
```

This produces a 256-bit sum and stores it in the memory location `BigVal3`.

8.1.2 Subtraction

The ARM also performs multibyte subtraction the same way you would manually, except that it subtracts whole words or double words at a time rather than decimal digits. Use the `subs` instruction on the LO word or double word and the `sbc/sbcs` (subtract with carry) instruction on the HO values.

The following example demonstrates a 128-bit subtraction using the 64-bit registers on the ARM:

```
        .data
Left:   .qword   .-.
Right:  .qword   .-.
Diff:   .qword   .-.
     .
     .
     .
    lea  x0, Left
    ldr  x3, [x0]
    lea  x1, Right
    ldr  x4, [x1]
    subs x5, x3, x4
    lea  x2, Diff
    str  x5, [x2]

    ldr  x3, [x0, #8]
    ldr  x4, [x1, #8]
    sbcs x5, x3, x4
    str  x5, [x2, #8]
```

The following example demonstrates a 256-bit subtraction:

```
        .data
BigVal1: .space  4*8      // Array of four dwords
BigVal2: .space  4*8
BigVal3: .space  4*8
     .
     .
     .

// Compute BigVal3 := BigVal1 - BigVal2.

    lea  x0, BigVal1
    lea  x1, BigVal2
    lea  x2, BigVal3

    ldr  x4, [x0]           // BigVal1[0]
    ldr  x5, [x1]           // BigVal2[0]
    subs x6, x4, x5
    str  x6, [x2]           // BigVal3[0]

    ldr  x4, [x0, #8]       // BigVal1[1]
    ldr  x5, [x1, #8]       // BigVal2[1]
    sbcs x6, x4, x5
    str  x6, [x2, #8]       // BigVal3[1]

    ldr  x4, [x0, #16]      // BigVal1[2]
    ldr  x5, [x1, #16]      // BigVal2[2]
    sbcs x6, x4, x5
    str  x6, [x2, #16]      // BigVal3[2]

    ldr  x4, [x0, #24]      // BigVal1[3]
    ldr  x5, [x1, #24]      // BigVal2[3]
    sbcs x6, x4, x5
    str  x6, [x2, #24]      // BigVal3[3]
```

This produces a 256-bit difference and stores it in the memory location `BigVal3`.

8.1.3 Comparisons

Unfortunately, there's no "compare with carry" instruction that you can use to perform extended-precision comparisons. However, you can compare extended-precision values by using just a `cmp` instruction.

Consider the two unsigned values 0x2157 and 0x1293. The LO bytes of these two values do not affect the outcome of the comparison. Simply comparing the HO bytes, 0x21 with 0x12, tells you that the first value is greater than the second.

You must look at both bytes of a pair of values if the HO bytes are equal. In all other cases, comparing the HO bytes tells you everything you need to know about the values. This is true for any number of bytes, not just two. The following code compares two signed 128-bit integers by comparing

their HO double words first and comparing their LO double words only if the HO quad words are equal:

```
// This sequence transfers control to location "IsGreater" if
// DwordValue > DwordValue2. It transfers control to "IsLess" if
// DwordValue < DwordValue2. It falls through to the instruction
// following this sequence if DwordValue = DwordValue2.
// To test for inequality, change the "IsGreater" and "IsLess"
// operands to "NotEqual" in this code.

        ldr x0, [fp, #DwordValue+8]   // Get HO dword.
        ldr x1, [fp, #DwordValue2+8]
        cmp x0, x1
        bgt IsGreater
        blt IsLess

        ldr x0, [fp, #DwordValue+0]   // If HO qwords equal,
        ldr x1, [fp, #DwordValue2+0] // then we must compare
        cmp x0, x1                    // the LO dwords.
        bgt IsGreater
        blt IsLess

// Fall through to this point if the two values are equal.
```

To compare unsigned values, use the `bhi` and `blo` instructions in place of `bgt` and `blt`.

You can synthesize any comparison from the preceding sequence, as shown in the following examples that demonstrate signed comparisons; just substitute `bhi`, `bhs`, `blo`, and `bls` for `bgt`, `bge`, `blt`, and `ble` (respectively) if you want unsigned comparisons. Each of the following examples assumes these declarations:

```
locals  lcl
oword   OW1
oword   OW2
byte    stkSpace, 64
endl    lcl
```

The following code implements a 128-bit test to see if `OW1 < OW2` (signed). Control transfers to the `IsLess` label if `OW1 < OW2`. Control falls through to the next statement (label `NotLess`) if this is not true:

```
    ldr x0, [fp, #OW1+8]   // Gets HO dword
    ldr x1, [fp, #OW2+8]
    cmp x0, x1
    bgt NotLess
    blt IsLess

    ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
    ldr x1, [fp, #OW2+0]   // dwords are equal.
    cmp x0, x1
    blt IsLess
NotLess:
```

Here is a 128-bit test to see if `OW1 <= OW2` (signed). This code jumps to `IsLessEQ` if the condition is true:

```
    ldr x0, [fp, #OW1+8]   // Gets HO dword
    ldr x1, [fp, #OW2+8]
    cmp x0, x1
    bgt NotLessEQ
    blt IsLessEQ

    ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
    ldr x1, [fp, #OW2+0]   // dwords are equal.
    cmp x0, x1
    ble IsLessEQ
NotLessEQ:
```

This is a 128-bit test to see if `OW1 > OW2` (signed). It jumps to `IsGtr` if this condition is true:

```
    ldr x0, [fp, #OW1+8]   // Gets HO dword
    ldr x1, [fp, #OW2+8]
    cmp x0, x1
    bgt IsGtr
    blt NotGtr

    ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
    ldr x1, [fp, #OW2+0]   // dwords are equal.
    cmp x0, x1
    bgt IsGtr
NotGtr:
```

The following is a 128-bit test to see if `OW1 >= OW2` (signed). This code jumps to label `IsGtrEQ` if this is the case:

```
    ldr x0, [fp, #OW1+8]   // Gets HO dword
    ldr x1, [fp, #OW2+8]
    cmp x0, x1
    bgt IsGtrEQ
    blt NotGtrEQ

    ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
    ldr x1, [fp, #OW2+0]   // dwords are equal.
    cmp x0, x1
    bge IsGtrEQ
NotGtrEQ:
```

Here is a 128-bit test to see if `OW1 == OW2` (signed or unsigned). This code branches to the label `IsEqual` if `OW1 == OW2`. It falls through to the next instruction if they are not equal:

```
    ldr x0, [fp, #OW1+8]   // Gets HO dword
    ldr x1, [fp, #OW2+8]
```

```
    cmp x0, x1
    bne NotEqual

    ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
    ldr x1, [fp, #OW2+0]   // dwords are equal.
    cmp x0, x1
    beq IsEqual
NotEqual:                  // Fall through to here if not equal.
```

The following is a 128-bit test to see if `OW1 != OW2` (signed or unsigned). This code branches to the label `IsNotEqual` if `OW1 != OW2`. It falls through to the next instruction if they are equal:

```
ldr x0, [fp, #OW1+8]   // Gets HO dword
ldr x1, [fp, #OW2+8]
cmp x0, x1
bne NotEqual

ldr x0, [fp, #OW1+0]   // Fall through to here if the HO
ldr x1, [fp, #OW2+0]   // dwords are equal.
cmp x0, x1
bne NotEqual

// Fall through to here if they are equal.
```

To generalize the preceding code for objects larger than 128 bits, start the comparison with the objects' HO double words and work your way down to their LO double words, as long as the corresponding double words are equal. The following example compares two 256-bit values to see if the first is less than or equal (unsigned) to the second:

```
        locals cmp256
        dword  Big1, 4
        dword  Big2, 4
        endl   cmp256
         .
         .
         .
        ldr  x0, [fp, #Big1+24]
        ldr  x1, [fp, #Big2+24]
        cmp  x0, x1
        blo  isLE
        bhi  notLE

        ldr  x0, [fp, #Big1+16]
        ldr  x1, [fp, #Big2+16]
        cmp  x0, x1
        blo  isLE
        bhi  notLE
```

```
        ldr  x0, [fp, #Big1+8]
        ldr  x1, [fp, #Big2+8]
        cmp  x0, x1
        blo  isLE
        bhi  notLE

        ldr  x0, [fp, #Big1+0]
        ldr  x1, [fp, #Big2+0]
        cmp  x0, x1
        bnls notLE
isLE:

        Code to execute if Big1 <= Big2
         .
         .
         .
notLE:

        Code to execute if Big1 > Big2
```

Presumably, there is a branch immediately before the `notLE` label to skip over the code to execute if `Big1 > Big2`.

8.1.4 Multiplication

Although 64 × 64-bit multiplication (or one of the smaller variants) is usually sufficient, sometimes you may want to multiply larger values. Use the ARM single-operand `umul` and `smul` instructions for extended-precision multiplication operations, using the same techniques that you employ when manually multiplying two values.

You likely perform multidigit multiplication by hand using the method shown in Figure 8-3.

Step 1: Multiply 5 × 3

```
  123
× 45
 ---
  15 (5 × 3)
```

Step 2: Multiply 5 × 2

```
  123
× 45
 ---
  15
 100 (5 × 20)
```

Step 3: Multiply 5 × 1

```
  123
× 45
 ---
  15
 100
 500 (5 × 100)
```

Step 4: Multiply 4 × 3

```
  123
× 45
 ---
  15
 100
 500
 120 (40 × 3)
```

Step 5: Multiply 4 × 2

```
  123
× 45
 ---
  15
 100
 500
 120
 800 (40 × 20)
```

Step 6: Multiply 4 × 1

```
  123
× 45
 ---
  15
 100
 500
 120
 800
4000 (40 × 100)
```

Step 7: Add partial products together

```
  123
× 45
 ---
  15
 100
 500
 120
 800
+4000
 ---
 5535
```

Figure 8-3: Multidigit multiplication

The ARM does extended-precision multiplication in the same manner, but with words and double words rather than digits, as shown in Figure 8-4.

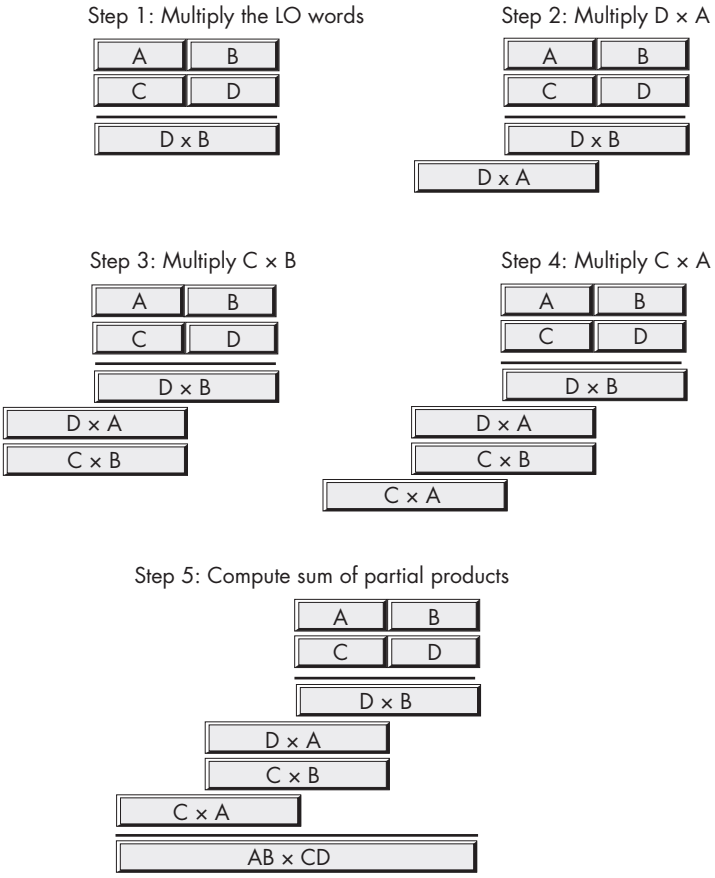


Figure 8-4: Extended-precision multiplication

When performing an extended-precision multiplication, remember that you must also perform an extended-precision addition at the same time. Adding up all the partial products requires several additions.

The `umul` and `smul` instructions you've seen thus far multiply two *n*-bit operands (32 or 64 bits), producing an *n*-bit result, ignoring any overflow. You can't easily use these instructions for multiprecision multiplication operations. Fortunately, the ARM CPU provides two sets of extended-precision multiplication instructions that will do the job: one set for 32 × 32 multiplications (producing a 64-bit result), and a second set for 64 × 64 multiplications (producing a 128-bit result).

Here are the instructions that produce 64-bit results:

```
smull  Xdest, Wsrc1, Wsrc2        // Xdest = Wsrc1 * Wsrc2 (signed long)
umull  Xdest, Wsrc1, Wsrc2        // Xdest = Wsrc1 * Wsrc2 (unsigned long)

smnegl Xdest, Wsrc1, Wsrc2        // Xdest = -(Wsrc1 * Wsrc2)
umnegl Xdest, Wsrc1, Wsrc2        // Xdest = -(Wsrc1 * Wsrc2)
```

```
smaddl Xdest, Wsrc1, Wsrc2, Xsrc3  // Xdest = (Wsrc1 * Wsrc2) + Xsrc3
umaddl Xdest, Wsrc1, Wsrc2, Xsrc3  // Xdest = (Wsrc1 * Wsrc2) + Xsrc3

smsubl Xdest, Wsrc1, Wsrc2, Xsrc3  // Xdest = (Wsrc1 * Wsrc2) - Xsrc3
umsubl Xdest, Wsrc1, Wsrc2, Xsrc3  // Xdest = (Wsrc1 * Wsrc2) - Xsrc3
```

The `smull` (signed multiply long) and `umull` (unsigned multiply long) instructions multiply the 32-bit registers to produce a 64-bit result, storing the result in the 64-bit destination register. The `smnegl` and `umnegl` also multiply two 32-bit values but negate the 64-bit result before storing it in the destination register.

The `smaddl`/`umaddl` and `smsubl`/`umsubl` instructions multiply their 32-bit operands, producing a 64-bit result, then add or subtract a 64-bit register from the result before storing the result into the 64-bit destination register. You could use the `smaddl`/`umaddl` instruction, for example, to multiply C × B and simultaneously add in D × A in Figure 8-4.

The 32 × 32 multiplication instructions are less useful than they seem because the existing *mxxx* instructions will accept 64-bit operands (producing a 64-bit result). You can easily zero- or sign-extend a 32-bit value into a 64-bit register and use the standard multiply instructions to achieve the same result as the long multiply instructions.

You could use the 32-bit long multiply instructions to synthesize larger multiplications (for example, a 128-bit multiplication). However, the ARM provides two additional instructions that are better suited for this: `smulh` and `umulh` (signed and unsigned multiply high):

```
smulh  Xdest, Xsrc1, Xsrc2   // Xdest = (Xsrc1 * Xsrc2) asr 64
umulh  Xdest, Xsrc1, Xsrc2   // Xdest = (Xsrc1 * Xsrc2) lsr 64
```

These instructions multiply the two 64-bit source operands and store the HO 64 bits of the 128-bit result into the destination register. The standard `mul` instruction produces the LO 64 bits of the result, so between the `mul` and `smulh`/`umulh` instructions, you can compute the full 128-bit result:

```
// Multiply X0 × X1, producing a 128-bit result in X3:X2
// (unsigned).

    mul    x2, x0, x1
    umulh  x3, x0, x1
```

For signed multiplication, simply substitute `smulh` for `umulh`.

To multiply larger values together, you can use the `mul`, `umulh`, and `smulh` instructions to implement the algorithm depicted in Figure 8-4. Listing 8-1 demonstrates how to multiply two 128-bit values (producing a 256-bit result) by using 64-bit instructions.

```
// Listing8-1.S
//
// 128-bit multiplication
```

```
#include "aoaa.inc"

            .code
            .extern printf

ttlStr:     wastr   "Listing 8-1"

fmtStr1:    .ascii  "%016lx_%016lx * %016lx_%016lx  = \n"
            wastr   "    %016lx_%016lx_%016lx_%016lx\n"

op1:        .qword  0x10001000100010001000100010001000
op2:        .qword  0x10000000000000000000000000000000

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// mul128
//
// Multiplies two unsigned 128-bit values passed on the stack by
// doing a 128x128-bit multiplication, producing a 256-bit
// result
//
// Stores result to location pointed at by X8

          ❶ proc    mul128

            args    a128
            qword   m128.mp    // Multiplier
            qword   m128.mc    // Multiplicand
            enda    a128

            locals  m128
            qword   m128.saveX01
            qword   m128.saveX23
            qword   m128.saveX45
            qword   m128.saveX67
            byte    stkSpace, 64
            endl    m128

            enter   m128.size

          ❷ stp     x0, x1, [fp, #m128.saveX01]  // Preserve
            stp     x2, x3, [fp, #m128.saveX23]  // these
            stp     x4, x5, [fp, #m128.saveX45]  // register
            stp     x6, x7, [fp, #m128.saveX67]  // values.


// Load operands into registers:

          ❸ ldr     x0, [fp, #m128.mp]
            ldr     x1, [fp, #m128.mp+8]
```

```
            ldr     x2, [fp, #m128.mc]
            ldr     x3, [fp, #m128.mc+8]

// X5:X4 = X0 * X2

            mul     x4, x0, x2
            umulh   x5, x0, x2

// X6:X7 = X1 * X2, then X5 = X5 + X7 (and save carry for later):

            mul     x7, x1, x2
            umulh   x6, x1, x2
            adds    x5, x5, x7

// X7 = X0 * X3, then X5 = X5 + X7 + C (from earlier):

            mul     x7, x0, x3
            adcs    x5, x5, x7
            umulh   x7, x0, x3
            adcs    x6, x6, x7  // Add in carry from adcs earlier.

// X7:X2 = X3 * X1

            mul     x2, x3, x1
            umulh   x7, x3, x1

            adc     x7, x7, xzr  // Add in C from previous adcs.
            adds    x6, x6, x2   // X6 = X6 + X2
            adc     x7, x7, xzr  // Add in carry from adds.

// X7:X6:X5:X4 contains 256-bit result at this point:

        ❹ stp     x4, x5, [x8]      // Save result to location
            stp     x6, x7, [x8, #16] // pointed at by X8.

            ldp     x0, x1, [fp, #m128.saveX01] // Restore
            ldp     x2, x3, [fp, #m128.saveX23] // saved
            ldp     x4, x5, [fp, #m128.saveX45] // registers.
            ldp     x6, x7, [fp, #m128.saveX67]
            leave
            endp    mul128

// Here is the asmMain function:


            proc    asmMain, public
            locals  am
            oword   product
            byte    stkSpace, 128
            endl    am

            enter   am.size

            str     xzr, [fp, #product]
```

```
// Test the mul128 function:

          ❺ lea     x2, op1
            ldp     x0, x1, [x2]
            stp     x0, x1, [sp]

            lea     x2, op2
            ldp     x0, x1, [x2]
            stp     x0, x1, [sp, #16]
            add     x8, fp, #product
            bl      mul128

// Print the result:

          ❻ lea     x0, op1         // Note: display HO
            ldr     x1, [x0, #8]    // dwords first so the
            mstr    x1, [sp]        // values appear normal.

            ldr     x2, [x0]
            mstr    x2, [sp, #8]

            lea     x0, op2
            ldr     x3, [x0, #8]
            mstr    x3, [sp, #16]
            ldr     x4, [x0]
            mstr    x4, [sp, #24]

            ldr     x5, [fp, #product+24]
            mstr    x5, [sp, #32]

            ldr     x6, [fp, #product+16]
            mstr    x6, [sp, #40]

            ldr     x7, [fp, #product+8]
            mstr    x7, [sp, #48]

            ldr     x0, [fp, #product]
// Under macOS, all arguments must be on stack for printf,
// under Linux, only eighth argument is on stack.

EightthArg   =       56  // For macOS
//EightthArg =       0   // For Linux

             str     x0, [sp, #EighthArg]


             lea     x0, fmtStr1
             bl      printf

             leave   // Returns to caller
             endp    asmMain
```

The `mul128` procedure ❶ multiplies two 128-bit values passed on the stack (note that this is not ARM ABI–compliant). Although X0 through X7 are volatile in the ARM ABI, this function is nice and preserves those registers ❷. The code loads the two 128-bit values from the stack into the X1:X0 and X3:X2 register pairs ❸. The 128-bit multiplication algorithm follows, as described in the program comments.

The code stores the 256-bit result into the memory location passed to this function in the X8 register ❹; then the `mul128` function restores the preserved registers and returns to the caller. The main program calls `mul128` ❺ and displays the result (in hexadecimal form) ❻.

Here's the `build` command and output from Listing 8-1:

```
$ ./build Listing8-1
$ ./Listing8-1
Calling Listing8-1:
1000100010001000_1000100010001000 * 1000000000000000_0000000000000000  =
    0100010001000100_0100010001000100_0000000000000000_1000100010001000
Listing8-1 terminated
```

The code works only for unsigned operands. To multiply two signed values, you must change the `umulh` instructions to `smulh`.

Listing 8-1 is fairly straightforward because it is possible to keep the partial products in various registers. If you need to multiply larger values together, you will need to maintain the partial products in temporary (memory) variables. Other than that, the algorithm that Listing 8-1 uses generalizes to any number of words.

8.1.5 Division

You cannot synthesize a general *n*-bit / *m*-bit division operation by using the `sdiv` and `udiv` instructions. A generic extended-precision division requires a sequence of shift and subtract operations, which takes quite a few instructions and runs much slower. This section presents the algorithm for extended-precision division.

As with multiplication, the best way to understand how the computer performs division is to study how you were probably taught to do long division by hand. Consider the steps you'd take to manually divide 3,456 by 12, as shown in Figure 8-5.

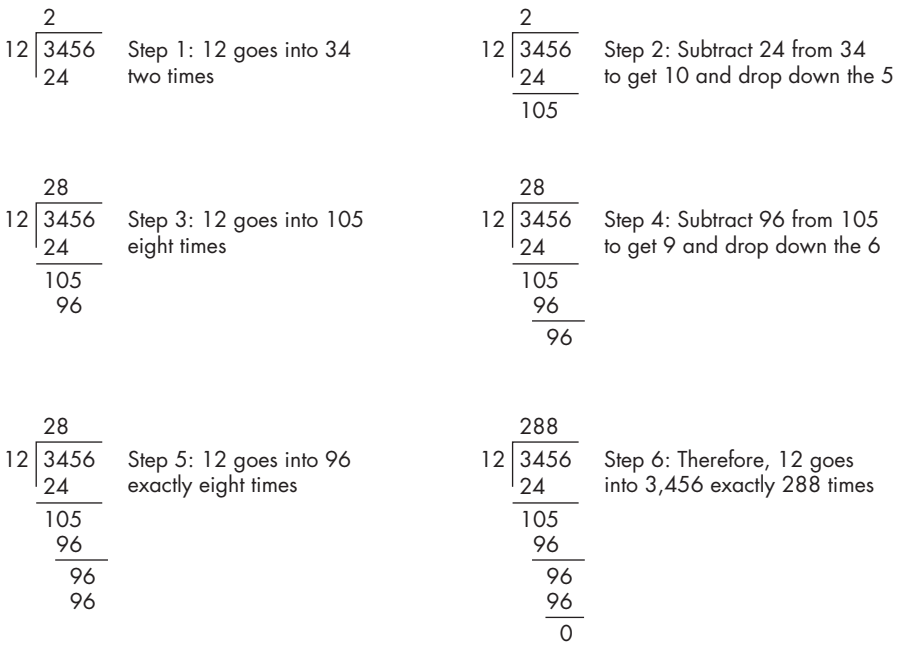


Figure 8-5: Manual digit-by-digit division operation

This algorithm is easier in binary because you don't have to guess at each step how many times 12 goes into the remainder, nor do you have to multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm, the divisor goes into the remainder exactly zero or one times. For example, Figure 8-6 shows how to divide 27 by 3 in binary (that is, dividing 11011 by 11).

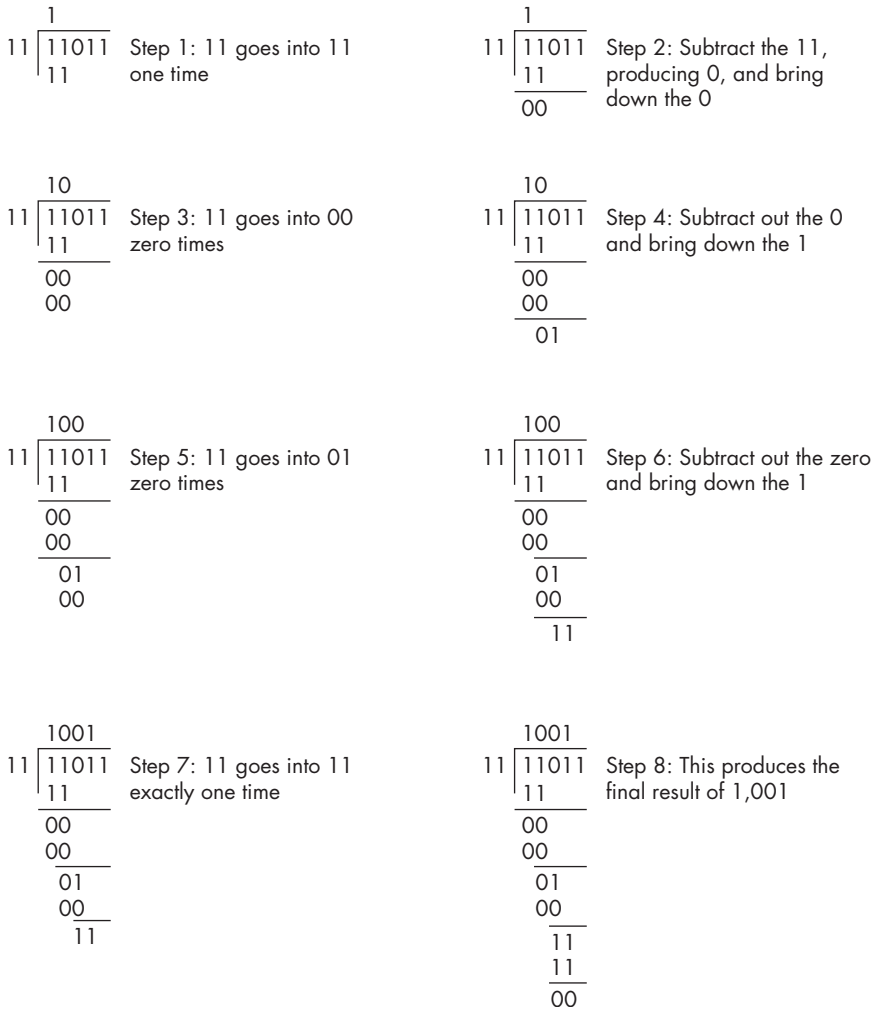


Figure 8-6: Longhand division in binary

The following algorithm implements this binary division operation in a way that computes the quotient and the remainder at the same time:

```
Quotient := Dividend;
Remainder := 0;
for i := 1 to NumberBits do

    Remainder:Quotient := Remainder:Quotient LSL 1;
    if Remainder >= Divisor then

        Remainder := Remainder - Divisor;
        Quotient := Quotient + 1;

    endif
endfor
```

NumberBits is the number of bits in the Remainder, Quotient, Divisor, and Dividend variables. LSL is the shift-left operator. The statement Quotient := Quotient + 1; sets the LO bit of Quotient to 1 because this algorithm previously shifted Quotient 1 bit to the left. Listing 8-2 implements this algorithm.

```
// Listing8-2.S
//
// 128-bit by 128-bit division

#include "aoaa.inc"

            .data

// op1 is a 128-bit value. Initial values were chosen
// to make it easy to verify the result.

op1:        .qword   0x2000400060008000A000C000E0001000
op2:        .qword   2
op3:        .qword   0xEEEECCCCAAAA88886666444422221111
result:     .qword   0
remain:     .qword   0

            .code
            .extern  printf

ttlStr:     wastr    "Listing 8-2"
fmtStr1:    .ascii   "quotient  = "
            wastr    "%016lx_%016lx\n"

fmtStr2:    .ascii   "remainder = "
            wastr    "%016lx_%016lx\n"

fmtStr3:    .ascii   "quotient (2)  = "
            wastr    "%016lx_%016lx\n"

// Return program title to C++ program:

            proc     getTitle, public
            lea      x0, ttlStr
            ret
            endp     getTitle

// div128
//
// This procedure does a general 128 / 128 division operation
// using the following algorithm (all variables are assumed
// to be 128-bit objects):
//
// Quotient := Dividend
// Remainder := 0
// for i := 1 to NumberBits do
//
//  Remainder:Quotient := Remainder:Quotient SHL 1
```

```
//  if Remainder >= Divisor then
//
//      Remainder := Remainder - Divisor
//      Quotient := Quotient + 1
//
// endif
// endfor
//
// Data passed:
//
// 128-bit dividend, by reference in X0
// 128-bit divisor, by reference in X1
//
// Data returned:
//
// Pointer to 128-bit quotient in X8
// Pointer to 128-bit remainder in X9

          ❶ proc    div128

#define remainderL  x10
#define remainderH  x11
#define dividendL   x12
#define dividendH   x13
#define quotientL   dividendL
#define quotientH   dividendH
#define divisorL    x14
#define divisorH    x15

            locals  d128
            dword   saveX0
            qword   saveX1011
            qword   saveX1213
            qword   saveX1415
            byte    stkSpace, 64
            endl    d128

quotient    =       dividend        // Alias to dividend

            enter   d128.size       // Set up activation record.

// Preserve registers div128 modifies:

          ❷ str     x0, [fp, #saveX0]
            stp     x10, x11, [fp, #saveX1011]
            stp     x12, x13, [fp, #saveX1213]
            stp     x14, x15, [fp, #saveX1415]

// Initialize remainder with 0:

          ❸ mov     remainderL, #0
            mov     remainderH, #0
```

```
// Copy the dividend to local storage:

            ldp     dividendL, dividendH, [x0]

// Copy the divisor to local storage:

            ldp     divisorL, divisorH, [x1]

            mov     w0, #128           // Count off bits in W0.

// Compute Remainder:Quotient := Remainder:Quotient LSL 1
//
// Note: adds x, x, x is equivalent to lsl x, x, #1
//       adcs x, x, x is equivalent to rol x, x, #1
//                     (if rol existed)
//
// The following four instructions perform a 256-bit
// extended-precision shift (left) dividend through
// remainder:

repeatLp:   adds    dividendL, dividendL, dividendL
            adcs    dividendH, dividendH, dividendH
            adcs    remainderL, remainderL, remainderL
            adc     remainderH, remainderH, remainderH

// Do a 128-bit comparison to see if the remainder
// is greater than or equal to the divisor:

            cmp     remainderH, divisorH
            bhi     isGE
            blo     notGE

            cmp     remainderL, divisorL
            bhi     isGE
            blo     notGE

// Remainder := Remainder - Divisor

isGE:       subs    remainderL, remainderL, divisorL
            sbc     remainderH, remainderH, divisorH

// Quotient := Quotient + 1:

            adds    quotientL, quotientL, #1
            adc     quotientH, quotientH, xzr

// Repeat for 128 bits:

notGE:      subs    w0, w0, #1
            bne     repeatLp

// Okay, copy the quotient (left in the Dividend variable)
// and the remainder to their return locations:

          ❹ stp     quotientL, quotientH, [x8]
            stp     remainderL, remainderH, [x9]
```

```
// Restore the registers div128 modified:

        ❺ ldr     x0, [fp, #saveX0]
          ldp     x10, x11, [fp, #saveX1011]
          ldp     x12, x13, [fp, #saveX1213]
          ldp     x14, x15, [fp, #saveX1415]
          leave   // Return to caller.
          endp    div128

// Here is the asmMain function:

          proc    asmMain, public

          locals  am
          byte    am.stkSpace, 64
          endl    am

          enter   am.size         // Sets up activation record


// Test the div128 function:

        ❻ lea     x0, op1
          lea     x1, op2
          lea     x8, result
          lea     x9, remain
          bl      div128

// Print the results:

          ldr     x1, [x8, #8]    // X8 still points at result.
          mstr    x1, [sp]
          ldr     x2, [x8]
          mstr    x2, [sp, #8]

          lea     x0, fmtStr1
          bl      printf

          lea     x9, remain      // Assume printf munged X9,
          ldr     x1, [x9, #8]    // must reload.
          mstr    x1, [sp]
          ldr     x2, [x9]
          mstr    x2, [sp, #8]

          lea     x0, fmtStr2
          bl      printf


// Test the div128 function (again):

          lea     x0, op3
          lea     x1, op2
          lea     x8, result
          lea     x9, remain
          bl      div128
```

```
// Print the results:

            ldr     x1, [x8, #8]    // X8 still points at result.
            mstr    x1, [sp]
            ldr     x2, [x8]
            mstr    x2, [sp, #8]

            lea     x0, fmtStr3
            bl      printf

            lea     x9, remain      // Must reload
            ldr     x1, [x9, #8]    // (because of printf).
            mstr    x1, [sp]
            ldr     x2, [x9]
            mstr    x2, [sp, #8]

            lea     x0, fmtStr2
            bl      printf

            leave   // Returns to caller
            endp    asmMain
```

The `div128` function ❶ is a 128 × 128-bit division operation that simultaneously produces the quotient and the remainder. Unlike the extended-precision multiplication given earlier, this function passes its arguments by reference (in X0 and X1) rather than by value on the stack. It stores the 128-bit quotient in the location pointed at by X8 and the remainder in the location pointed at by X9. As in the multiplication code, the `div128` function ❷ preserves all the volatile registers it modifies.

Next is the division algorithm ❸, as described in the program comments. The code stores the quotient and remainder away ❹ and then restores the preserved registers ❺. The main program ❻ demonstrates the `div128` function with a pair of calls, along with the code to display the results.

Here's the `build` command and program output:

```
$ ./build Listing8-2
$ ./Listing8-2
Calling Listing8-2:
quotient  = 1000200030004000_5000600070000800
remainder = 0000000000000000_0000000000000000
quotient (2)  = 7777666655554444_3333222211110888
remainder = 0000000000000000_0000000000000001
Listing8-2 terminated
```

This code does not check for division by 0 (it will produce the quotient 0xFFFF_FFFF_FFFF_FFFF if you attempt to divide by 0). It handles only unsigned values and is very slow, a couple of orders of magnitude worse than the `sdiv`/`udiv` instructions. To handle division by 0, check the divisor against 0 prior to running this code and return an appropriate error code if the divisor is 0. To deal with signed values, note the signs, take the operands'

absolute values, do the unsigned division, and then fix the sign afterward by setting the result negative if the operand signs were different.

8.1.6 Negation

The `neg` instruction doesn't provide a generic extended-precision form. However, a negation is equivalent to subtracting a value from 0, so you can easily simulate an extended-precision negation by using the `subs` and `sbcs` instructions.

The following code provides a simple way to negate a (320-bit) value by subtracting that value from 0, using an extended-precision subtraction:

```
ldr  x0, [fp, #value320]
subs x0, xzr, x0
str  x0, [fp, #value320]

ldr  x0, [fp, #value320+8]
sbcs x0, xzr, x0
str  x0, [fp, #value320+8]

ldr  x0, [fp, #value320+16]
sbcs x0, xzr, x0
str  x0, [fp, #value320+16]

ldr  x0, [fp, #value320+24]
sbcs x0, xzr, x0
str  x0, [fp, #value320+24]

ldr  x0, [fp, #value320+32]
sbcs x0, xzr, x0
str  x0, [fp, #value320+32]
```

You can extend this algorithm to any number of bits (or reduce it to fewer bits) by using the scheme I presented for extended-precision subtraction.

8.1.7 AND

Performing an *n*-byte AND operation is easy: simply AND the corresponding bytes between the two operands, saving the result. For example, to perform the AND operation with all operands 128 bits long, you could use the following code:

```
ldp x0, x1, [fp, #source1]
ldp x2, x3, [fp, #source2]
and x2, x2, x0
and x3, x3, x1
stp x2, x3, [fp, #dest]
```

To extend this technique to any number of dwords, logically AND the corresponding dwords together in the operands.

When testing the flags after an AND sequence, remember that the ands instruction will set the flags only for that particular portion of the AND sequence. If you convert the last and to an ands instruction, it will properly set the N flag but will not properly set the Z flag. To set the Z flag (indicating a 0 result for the entire 128 bits), you can use ccmp (conditional compare) to test the Z flag from the ands instruction and compare X2 with 0 (see section 6.1.4, "Conditional Instructions," on page 297):

```
ldp  x0, x1, [fp, #source1]
ldp  x2, x3, [fp, #source2]
and  x2, x2, x0
ands x3, x3, x1
stp  x2, x3, [fp, #dest]
ccmp x2, #0, 0b0100, eq   // Sets Z if X3 == 0 && X2 == 0
```

If you need to test both the N and Z flags after this sequence, consider using the tbz/tbnz instructions to test the HO bit of register X3, which contains the sign bit.

8.1.8 OR

Multibyte logical OR operations are performed in the same way as multibyte AND operations: you OR the corresponding bytes in the two operands together. For example, to logically OR two 256-bit values, use the following code:

```
ldp x0, x1, [fp, #source1]
ldp x2, x3, [fp, #source1+16]
ldp x4, x5, [fp, #source2]
ldp x6, x7, [fp, #source2+16]

orr x0, x0, x4
orr x1, x1, x5
orr x2, x2, x6
orr x3, x3, x7

stp x0, x1, [fp, #dest]
stp x2, x3, [fp, #dest+16]
```

Remember that the orr instruction does not affect any flags (and there is no orrs instruction). If you need to test the zero flag after an extended-precision OR, you must compare all the resulting double words to 0.

You can also use the V*n* registers to perform extended-precision logical operations, up to 128 bits at a time. See section 11.13, "Use of SIMD Instructions in Real Programs," on page 699 for more details.

8.1.9 XOR

As with other logical operations, extended-precision XOR operations will XOR the corresponding bytes in the two operands to obtain the extended-precision result. The following code sequence operates on two 128-bit

operands, computes their exclusive-OR, and stores the result into a 128-bit variable:

```
ldp x0, x1, [fp, #source1]
ldp x2, x3, [fp, #source2]
eor x2, x2, x0
eor x3, x3, x1
stp x2, x3, [fp, #dest]
```

The comment about the zero flag in the previous section applies here, as well as the comment about V*n* registers.

8.1.10 NOT

The `mvn` instruction inverts all the bits in the specified operand. Perform an extended-precision NOT by executing the `mvn` instruction on all the affected operands. For example, to perform a 128-bit NOT operation on the value in X1:X0, execute the following instructions:

```
mvn x0, x0
mvn x1, x1
```

If you execute the `mvn` instruction twice, you wind up with the original value. Also, exclusive-ORing a value with all 1s (such as 0xFF, 0xFFFF, 0xFFFF_FFFF, or 0xFFFF_FFFF_FFFF_FFFF) performs the same operation as the `mvn` instruction.

8.1.11 Shift Operations

Extended-precision shift operations on the ARM are somewhat problematic. Traditionally, the way you accomplish an extended-precision shift is to shift a bit out of one register into the carry flag, then rotate that carry bit into another register. Unfortunately, the ARM doesn't provide such instructions, so a different approach is necessary.

The exact approach depends on two things, as described in the following subsections: the number of bits to shift and the direction of the shift.

8.1.11.1 Shift Left

A 128-bit `lsl` (logical shift left) takes the form shown in Figure 8-7.

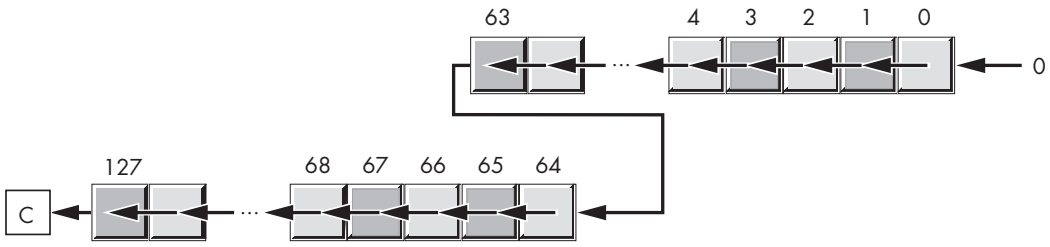


Figure 8-7: The 128-bit shift-left operation

To accomplish this with machine instructions, you must first shift the LO dword to the left (for example, using the `lsls` instruction) and capture the output from bit 63 (conveniently, the carry flag does this for us). Next, shift this bit into the LO bit of the HO dword while simultaneously shifting all the other bits to the left (and capturing the output by using the carry flag). No instruction specifically rotates the carry flag into a register, but you can use the magic instruction `adc/adcs` to do this if you supply appropriate operands.

Remember, a shift left is the same thing as a multiplication by 2. Adding a value to itself is the very definition of a multiplication by 2. Therefore, the `lsls` and `adds` instructions can both shift an operand to the left, moving the overflow bit into the carry flag. In order for `adds` to behave like a shift-left operation, you must supply the same operand in both source positions:

```
adds x0, x0, x0  // Same as lsl x0, x0, #1
```

The `adcs` instruction (with the same operands) will also shift all the bits to the left one position and shift the carry flag into bit 0 (as well as shift the HO bit into the carry flag at the end of the operation). This is, effectively, a single-bit *rotate-through-carry-left* operation, as illustrated in Figure 8-8.

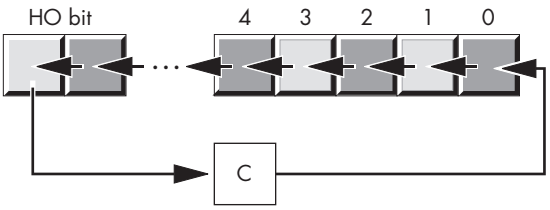


Figure 8-8: The rotate-through-carry-left operation

You can use the `adds` and `adcs` instructions to implement a 128-bit shift. For example, to shift the 128-bit quantity in X1:X0 one position to the left, use the following instructions:

```
adds x0, x0, x0
adcs x1, x1, x1
```

The `adds` instruction shifts a 0 into bit 0 of the 128-bit operand and shifts bit 63 into the carry flag. The `adcs` instruction then shifts the carry flag into bit 64 and shifts bit 127 into the carry flag, giving you exactly the result you want, as shown in Figure 8-9.

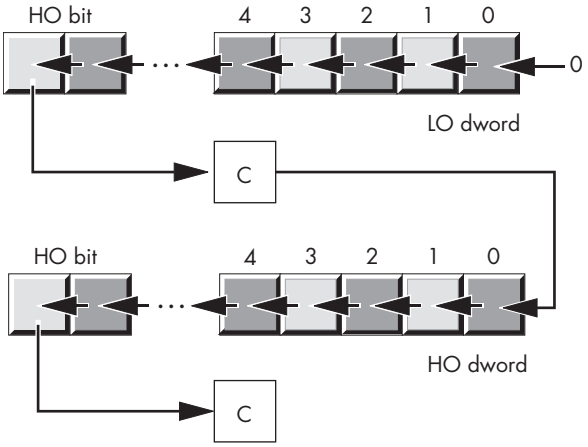


Figure 8-9: Extended-precision shift left using adds/adcs

Using this technique, you can shift an extended-precision value only 1 bit at a time. You cannot shift an extended-precision operand several bits by using a register, nor can you specify a constant value greater than 1 when using this technique.

To perform a shift left on an operand larger than 128 bits, use additional `adcs` instructions. An extended-precision shift-left operation always starts with the least-significant double word, and each succeeding `adcs` instruction operates on the next-most-significant double word. For example, to perform a 192-bit shift-left operation on a memory location, you could use the following instructions:

```
adds x0, x0, x0
adcs x1, x1, x1
adcs x2, x2, x2
```

If you need to shift your data by 2 or more bits, you can either repeat the preceding sequence the desired number of times for a constant number of shifts or place the instructions in a loop to repeat them a certain number of times. For example, the following code shifts the 192-bit value in X0, X1, and X2 to the left by the number of bits specified in W3:

```
ShiftLoop:
    adds x0, x0, x0
    adcs x1, x1, x1
    adcs x2, x2, x2
    subs w3, w3, #1
    bne  ShiftLoop
```

The only problem with this multibit shift is that it can run rather slowly when shifting more than a few bits to the left. In general, we say that this algorithm is O(*n*), meaning the runtime is proportional to the number of bits we shift to the left.

An instruction to shift multiple bits simultaneously, as the `lsl` instruction can do, would help solve this problem. If a `rol` instruction existed, you could use it to shift the 128 bits in X1:X0 to the left 8 bits:

```
rol     x2, x0, #8       // Shift H0 8 bits into L0 8
and     x2, x2, #0xFF    // bits and clear other bits.
lsl     x0, x0, #8       // Shift X0 8 bits.
lsl     x1, x1, #8       // Shift X1 8 bits.
orr     x1, x1, x2       // Merge in L0 8 bits.
```

Unfortunately, the ARM CPU's instruction set has no `rol` instruction; however, you can use the `ror` instruction to do anything a `rol` instruction would do. For any bit shift that occurs in the range 1–63, `rol(`*n*`)` is equivalent to `ror((64 - `*n*`) % 64)`, where `ro`*x*`(`*n*`)` means "rotate left/right *n* bits." For the special case of `rol(0)`, `ror(0) ((64 - 0) % 64) is 0)` will also rotate the value 0 bits. Therefore, you can replace the previous noncompiling code with this:

```
ror     x2, x0, #64-8    // Shift H0 8 bits into L0 8
and     x2, x2, #0xFF    // bits and clear other bits.
lsl     x0, x0, #8       // Shift X0 8 bits.
lsl     x1, x1, #8       // Shift X1 8 bits.
orr     x1, x1, x2       // Merge in L0 8 bits.
```

When *n* is greater than 2 or 3, this sequence will execute much faster than the `adds/adcs` loop given earlier.

Figures 8-10 through 8-14 show the operations for this extended-precision shift left.

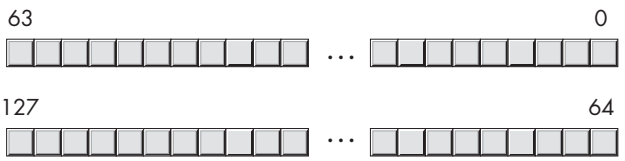


Figure 8-10: Extended-precision shift left using `ror`, before the shift

In Figure 8-11, the algorithm makes a temporary copy of bits 0 to 63 and rotates the value to the left by 8 bits.

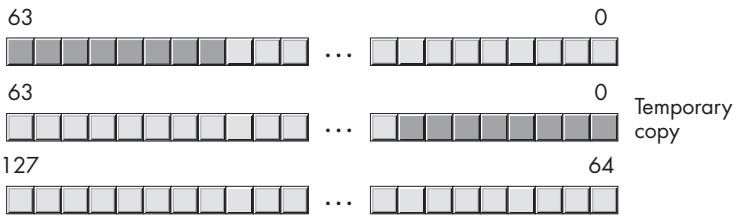


Figure 8-11: Step 1: Making a temporary copy and shifting bits

Figure 8-12 shows shifting the original value to the left 8 bits (which clears the LO bits) and clearing the HO temporary bits (via an AND operation).

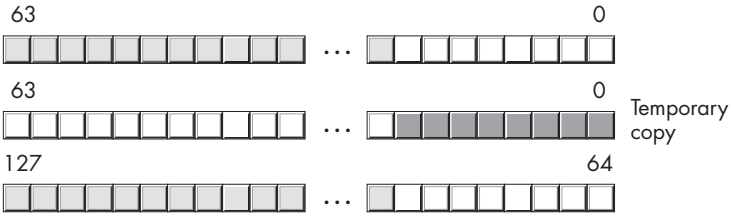


Figure 8-12: Step 2: Shifting and clearing bits

Figure 8-13 shows the merging of the temporary and HO dwords (OR operation).

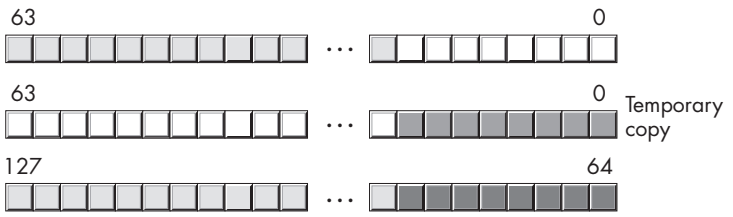


Figure 8-13: Step 3: Merging the temporary and HO dwords

Figure 8-14 shows the result after the shift.

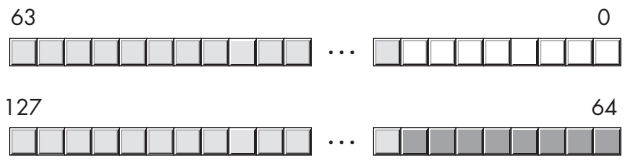


Figure 8-14: Step 4: After the shift

To implement a variable extended-precision shift-left operation, the code needs to generate a bitmask to clear the LO bits (the `and` instructions in the previous code). As it turns out, you can generate the mask for an *n*-bit shift by using the following code:

```
mov x3, #1
lsl x3, x3, x4   // Assume X4 contains the shift count.
sub x3, x3, #1   // Generates 1 bits in positions 0 to (n-1)
and x2, x2, x3   // Clears unused bits of X2
```

The trick here is that `lsl`(*n*) produces 2^n. Then, $2^n - 1$ is all 1 bits from bit 0 to position $n - 1$.

8.1.11.2 Shift Right and Arithmetic Shift Right

Unfortunately, no trick like using the adds/adcs instructions allows you to perform a *rotate through carry right* operation (shifting all the bits right through the carry, and shifting the original carry back into the HO bit). Therefore, to do an extended-precision shift right (or arithmetic shift right), you must use the ror instruction again. Here's an example that shifts a 128-bit value in X1:X0 to the right 8 bits:

```
ror x2, x1, #8            // Shifts bits 64-71 into HO
and x2, x2, #0xFF << 56   // 8 bits and clears bits 64-119
lsr x1, x1, #8            // Shifts X1 8 bits
lsr x0, x0, #8            // Shifts X0 8 bits
orr x0, x0, x2            // Merges in bits 56-63
```

The code for the extended-precision arithmetic shift-right operation is similar:

```
ror x2, x1, #8            // Shifts bits 64-71 into HO
and x2, x2, #0xFF << 56   // 8 bits and clears bits 64-119
asr x1, x1, #8            // Arithmetic shift X1 8 bits
lsr x0, x0, #8            // Shifts X0 8 bits
orr x0, x0, x2            // Merges in bits 56-63
```

In this case, you substitute an asr instruction for the lsr on the HO dword. Note that you continue to use a lsr instruction on the LO dword; lsr is necessary to shift 0s into the HO bits so that the orr instruction properly merges the bits shifted out of the HO dword.

As a last example, here's a 192-bit arithmetic shift right that shifts the bits in X2:X1:X0 to the right 4 bits:

```
ror x3, x2, #4            // Temp copy holding bits 128-131
And x3, x3, #0xF << 60    // Clears all but HO 4 bits of temp
asr x2, x2, #4            // Arithmetic shift right X2 4 bits
ror x4, x2, #4            // Temp (2) copy holding bits 64-67
And x4, x4, #0xF << 60    // Clears all but HO 4 bits of temp2
lsr x2, x2, #4            // Shifts the original 3 dwords 4 bits
lsr x1, x1, #4
lsr x0, x0, #4
orr x1, x1, x3            // Merges in bits 124-127
orr x0, x0, x4            // Merges in bits 60-63
```

The Neon instructions allow you to shift 128-bit values left and right; see Chapter 11 for details.

8.2 Operating on Different-Size Operands

Occasionally, you may need to do a computation on a pair of operands that are not the same size (*mixed-size*, or *mixed-mode*, arithmetic). For example, you may need to add a word and a double word together or subtract a byte

value from a word value. To do so, extend the smaller operand to the size of the larger operand, then operate on two same-size operands. For signed operands, sign-extend the smaller operand to the same size as the larger operand; for unsigned values, zero-extend the smaller operand. This works for any operation.

The following examples demonstrate adding a byte variable, a half-word variable, and a dword variable:

```
locals  lcl
byte    var1
hword   var2
align   3
dword   var3
endl    lcl
 .
 .
 .
// Unsigned addition (8-bit + 16-bit addition
// producing a 16-bit result):

ldrb    w0, [fp, #var1] // Zero-extends byte to 32 bits
ldrh    w1, [fp, #var2] // Zero-extends hword to 32 bits
add     w0, w0, w1      // Adds 32 bits
strh    w0, [fp, #var2] // Store LO 16 bits in var2.

// Signed addition (8-bit + 16-bit addition
// producing a 16-bit result):

ldrsb   w0, [fp, #var1] // Sign-extends byte to 32 bits
ldrsh   w1, [fp, #var2] // Sign-extends hword to 32 bits
add     w0, w0, w1      // Adds 32 bits
strh    w0, [fp, #var2] // Store LO 16 bits in var2.
```

In both cases, the byte variable is loaded into the W0 register, extended to 32 bits, and then added with the half-word operand (also extended to 32 bits).

All these examples add a byte value to a half-word value. By zero- or sign-extending the operands to the same size, you can easily add any two different-size variables together.

As a last example, consider adding an 8-bit signed value to a qword (128-bit) value:

```
ldrsb   x0, [fp, #var1] // Sign-extends byte to 64 bits
asr     x1, x0, #63     // Sneaky sign-extend to 128 bits
ldp     x2, x3, [fp, #var3]
adds    x2, x2, x0      // Adds LO dwords
adc     x3, x3, x1      // Adds HO dwords
stp     x2, x3, [fp, #var3]
```

The trick in this code is the `asr` instruction. This instruction sign-extends X0 into X1:X0 by copying the sign bit in X0 throughout X1 (an

arithmetic shift right by 63 bits effectively copies bit 63 into bits 0–62). Once X0 has been sign-extended into X1, you have a 128-bit value in X1:X0 that you can add to the 128-bit value in variable `var3`.

The previous examples in this chapter assumed that the different-size operands were memory variables. They used the `ldrb/ldrsb` and `ldrh/ldrsh` instructions to zero- and sign-extend 8- and 16-bit operands to 32 bits (which could also extend their operands to 64 bits by supplying a 64-bit register). Although these examples did not demonstrate mixing 32- and 64-bit operands, you could also have used the `ldrsw` instruction to sign-extend 32 bits to 64.

If your operands are already in registers (not memory), you can use the `uxtb/uxth/uxtw` and `sxtb/sxth/sxtw` instructions to zero- or sign-extend the operands. For example, the following code sign-extends the 32-bit value in W0 to 128 bits:

```
// Assume 8-bit value is in W0 and 128-bit value is in X3:X2.
// Add byte in W0 to 128-bit value in X3:X2.

sxtb    x0, w0          // Sign-extends byte to 64 bits
asr     x1, x0, #63     // Sneaky sign-extend to 128 bits
adds    x2, x2, x0      // Adds LO dwords
adc     x3, x3, x1      // Adds HO dwords
```

When adding smaller values to 32- or 64-bit registers that don't require sign-extending the smaller value to 128 bits or more, you can use the sign-extension modifiers for Operand2 in arithmetic instructions to zero- and sign-extend the smaller values to the larger size:

```
// Add 8-bit unsigned value in W0 to 32-bit value in W1:

add    w1, w1, w0, uxtb #0

// Add 8-bit signed value in W0 to 32-bit value in W1:

add    w1, w1, w0, sxtb #0

// Add 16-bit unsigned value in W0 to 32-bit value in W1:

add    w1, w1, w0, uxth #0

// Add 16-bit signed value in W0 to 32-bit value in W1:

add    w1, w1, w0, sxth #0

// Add 32-bit unsigned value in W0 to 64-bit value in X1:

add    x1, x1, w0, uxtw #0

// Add 32-bit signed value in W0 to 64-bit value in X1:

add    x1, x1, w0, sxtw #0
```

To add bytes and half words to 64-bit dwords, just change the W1 registers to X1 in this code.

8.3 Moving On

Extended-precision arithmetic is difficult or impossible in HLLs but is fairly easy in assembly language. This chapter described the extended-precision arithmetic, comparison, and logical operations in ARM assembly language. It concluded by discussing mixed-mode (mixed-size) arithmetic, where the operands have differing sizes.

Armed with the information from this chapter, it's easy to handle arithmetic and logical operations that are difficult to achieve in most HLLs. The next chapter, which covers numeric-to-string conversions, will use these extended-precision operations when converting values larger than 64 bits.

8.4 For More Information

- One arithmetic feature missing from the ARM instruction set is *decimal arithmetic* (base-10), meaning if the need arises, you'll have to perform that arithmetic in software. Though most of the code is in C, visit the General Decimal Arithmetic site if you want to implement decimal arithmetic: *https://speleotrove.com/decimal/*.
- Donald Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (Addison-Wesley Professional, 1997) contains lots of useful information about decimal arithmetic and extended-precision arithmetic, though the text is generic and describes how to do this in MIXAL assembly language rather than ARM assembly language.

TEST YOURSELF

Assume all variables are unsigned integers and are local in the current activation record.

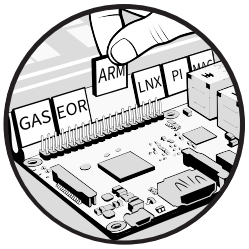
1. Provide the code to compute $x = y + z$, assuming the following:
   a. x, y, and z are 128-bit integers.
   b. x and y are 96-bit integers, and z is a 64-bit integer.
   c. x, y, and z are 48-bit integers.
2. Provide the code to compute $x = y - z$, assuming the following:
   a. x, y, and z are 192-bit integers.
   b. x, y, and z are 96-bit integers.

(continued)

3. Provide the code to compute $x = y \times z$, assuming x, y, and z are 128-bit unsigned integers.
4. Assuming x and y are unsigned 128-bit integers, convert the following to assembly language:
   a. `if( x == y )` then *code*
   b. `if( x < y )` then *code*
   c. `if( x > y )` then *code*
   d. `if( x != y )` then *code*
5. Assuming x and y are signed 128-bit integers, convert the following to assembly language:
   a. $x = -x$
   b. $x = -y$
6. Assuming x, y, and z are all 128-bit integer values, convert the following to assembly language:
   a. $x = y$ & z (bitwise logical AND)
   b. $x = y$ | z (bitwise logical OR)
   c. $x = y$ ^ z (bitwise logical XOR)
   d. $x = \sim y$ (bitwise logical NOT)
   e. $x = y$ << 1 (bitwise shift left)
   f. $x = y$ >> 1 (bitwise shift right)
7. Assuming x and y are signed 128-bit values, convert $x = y$ >> 1 to assembly language (bitwise arithmetic shift right).
8. Provide the assembly code to rotate the 128-bit value in x through the carry flag (left by 1 bit).

9

NUMERIC CONVERSION



This chapter discusses basic conversions between various numeric formats, including integer to decimal string, integer to hexadecimal string, floating-point to string, hexadecimal string to integer, decimal string to integer, and real string to floating-point. It also covers error handling for string-to-numeric conversions, as well as performance enhancements. Finally, it introduces standard-precision conversions (for 8-, 16-, 32-, and 64-bit integer formats) and extended-precision conversions (for example, 128-bit integer/string conversions).

In this chapter, you'll begin to solve problems directly in assembly language, rather than translating a solution from an HLL as you did in previous chapters. Some examples here first present code that solves a problem with an HLL-based solution, then provide an optimized assembly language

solution. This should help you learn to solve assembly language problems without relying on HLLs, thereby producing higher-quality programs.

9.1 Converting Numeric Strings to Values

Up to this point, this book has relied on the C stdlib to perform numeric I/O (writing numeric data to the display and reading numeric data from the user). However, the library doesn't provide extended-precision numeric I/O facilities (and even 64-bit numeric I/O is questionable; this book has been using a GCC extension to `printf()` to do 64-bit numeric output). Therefore, it's time to break down how to do numeric I/O in assembly language.

Because most OSes support only character or string input and output, you won't actually do numeric I/O. Instead, you'll write functions that convert between numeric values and strings, then do string I/O. The examples in this section work with 64-bit (non-extended-precision) and 128-bit values, but the algorithms are general and extend to any number of bits.

9.1.1 Numeric Values to Hexadecimal Strings

In this section, you'll learn to convert numeric values (bytes, half words, words, double words, and so on) to a character string containing the equivalent hexadecimal characters for the value. To begin, you need a function that converts a 4-bit nibble into a single ASCII character in the range `'0'` to `'9'` or `'A'` to `'F'`. In an HLL such as C, you could write this as follows:

```
// Assume nibbleIn is in the range 0-15:

charOut = nibbleIn + '0';
if( charOut > '9' ) charOut = charOut + ('A' - '9' - 1);
```

You can convert any numeric value in the range 0 to 9 to its corresponding ASCII character by ORing the numeric value with `'0'` (0x30). Unfortunately, this maps numeric values in the range 0xA through 0xF to 0x3A through 0x3F, so the C code checks to see if it produces a value greater than 0x3A and adds `7` (`'A' – '9' – 1`) to produce a final character code in the range 0x41 to 0x46 (`'A'` through `'F'`).

With a function that converts a nibble to the appropriate ASCII character, you can convert bytes, half words, and so on by taking all the nibbles in the number and calling the function on each one to produce the corresponding output character. However, because ARM assembly language programs generally deal with objects no smaller than a byte, it's more straightforward and efficient to write a function that converts a byte value to two ASCII characters. Let's call this function `btoh` (byte to hex).

Listing 9-1 shows a straightforward `btoh` implementation. This function expects a single-byte value in X1 (ignoring bits 8 to 63 in X1) and returns the two characters in bits 0 to 15 of X1. Listing 9-1 converts a C algorithm into assembly language by using the techniques described in Chapter 7.

```
// Listing9-1.S

#include "aoaa.inc"

            proc    btoh_simple
            and     x1, x1, #0xFF   // Ensure only 8 bits.
            mov     x0, x1          // Save LO nibble.

            // Process the HO nibble:

          ❶ lsr     x1, x1, #4      // Move HO nibble to LO posn.
            orr     x1, x1, #'0'    // Convert to 0x30 to 0x3F.
            cmp     x1, #'9'        // See if 0x3A to 0x3F.
            bls     le9as
            add     x1, x1, #7      // Convert 0x3A to 0x3F to
le9as:                              // 'A' through 'F'.

            // Process the LO nibble:

          ❷ and     x0, x0, #0xF    // Strip away HO nibble.
            orr     x0, x0, #'0'    // Convert to 0x30 to 0x3F.
            cmp     x0, #'9'        // See if 0x3A to 0x3F.
            bls     le9bs
            add     x0, x0, #7      // Convert 0x3A to 0x3F to
le9bs:                              // 'A' through 'F'.
            // Merge the 2 bytes into X1.

            orr     x1, x1, x0, lsl #8
            ret
            endp    btoh_simple
```

This function returns the character corresponding to the HO nibble in bits 0 through 7 ❶ and the character corresponding to the LO nibble in bits 8 through 15 ❷. This is because you'll generally use this function to build up character strings containing the converted hexadecimal value. Character strings are inherently *big-endian*, with the most significant digit appearing in the lowest memory address (so the number will be read from left to right when you print the string). Returning the two characters swapped in X1 allows you to store the two characters as a half-word value into memory by using a single instruction.

You may be wondering why `btoh_simple` passes the value to convert in X1 rather than X0 (the standard "first argument" location). This is in anticipation of functions that will output the characters to a memory buffer (string). For those string-based functions, X0 will contain the address of the buffer.

Because Listing 9-1 is basically hand-compiled C/C++ code, the performance will be about the same as (or worse than) the code produced by an optimizing C/C++ compiler processing the C code given earlier. To write faster code in assembly language, you'll first need to measure the performance of two functions to determine which one is faster. While you can do so with many software tools (performance analyzers, or *profilers*), I've employed a simple solution: write a main program that calls the function

many times, then use the Unix `time` command line utility to measure the amount of time the program takes to run. Listing 9-2 shows such a program, for example.

```
// Listing9-2.S

#include "aoaa.inc"

Include both simple and other code here necessary for a working program.

            proc    asmMain, public

            locals  am                  // Preserve the X20 and
            dword   saveX20             // X21 registers that
            dword   saveX21             // this program uses
            byte    stackspace, 64      // as loop-control
            endl    am                  // variables.

            enter   am.size    // Create activation record.

            str     x20, [fp, #saveX20] // Preserve nonvolatile
            str     x21, [fp, #saveX21] // registers.

// Outer loop executes 10,000,000 times:

            ldr     x20, =10000000
outer:

// Inner loop executes 256 times, once for each byte value.
// It just calls the btoh_*** function and ignores the
// return value. Do this to measure the speed of the
// function.

#define funcToCall btoh_x1 // btoh_x1, btoh2, btoh_nob, or btoh_simple

            mov     x21, #256
inner:      add     x1, x20, #-1
            bl      funcToCall
            adds    x21, x21, #-1
            bne     inner
            adds    x20, x20, #-1
            bne     outer



            mov     x1, #0x9a       // Value to test
            mov     x6, x1          // Save for later.
            bl      funcToCall

            // Print btoh_*** return result:

            and     x2, x1, #0xff   // Print HO nibble first.
            mstr    x2, [sp, #8]
            lsr     x3, x1, #8      // Print LO nibble second.
```

```
        mstr    x3, [sp, #16]
        mov     x1, x6          // Retrieve save value.
        mstr    x1, [sp]
        lea     x0, fmtStr1
        bl      printf
        ldr     x21, [fp, #saveX21] // Restore nonvolatile
        ldr     x20, [fp, #saveX20] // registers.
        leave
        ret

        endp    asmMain
```

An advanced software engineer might find several faults with this technique for measuring the executing time of some code. However, it is simple, is easy to understand and use, and doesn't require any special software tools. While the measurements it produces are not perfect, it's good enough for most purposes.

Here's the `build` command and sample output (using the Unix `time` command to time the running of the program):

```
$ ./build Listing9-2
$ time ./Listing9-2
Calling Listing9-2:
Value=9a, as hex=9A
Listing9-2 terminated
./Listing9-2  3.49s user 0.01s system 98% cpu 3.542 total
```

On my Mac mini M1, this took about 3.5 seconds to run. (Obviously, this will vary by system; for example, on a Raspberry Pi 3, it took about 37 seconds.)

As noted in Chapter 7, branches tend to run slower than straight-line code. Listing 9-2 uses branches to handle cases when the converted character is `'0'` through `'9'` or `'A'` through `'F'`. I wrote a version using the `csel` instruction to differentiate these two cases after ORing or adding `'0'` to the nibble value. The code ran in 2.5 seconds (on a Mac mini M1). However, this was achieved by not preserving the X1 and X2 registers. Saving X1 and X2 to memory and restoring them increased the execution time to 4.68 seconds.

You've just discovered a big time sink in ARM assembly code: accessing memory is very slow (and the `ldp/stp` instructions are much slower than the `ldr/str` instructions). This is why Arm defined nonvolatile registers, so you don't have to preserve certain working registers in memory. Nevertheless, preserving volatile registers is sometimes worthwhile to ensure that programs are correct. Assembly language code can quickly become complex, and having a function stomp on registers you forgot to save in your calling code can lead to long debugging sessions. A fast program with defects is never as good as a slower program that works properly.

When writing 32-bit ARM code for a Raspberry Pi 400 (for the second volume of this series), I discovered that using a 256-element lookup table (with each element containing the two characters corresponding to the

hexadecimal value) was faster than the standard algorithm. When I tried that approach in 64-bit ARM assembly, the runtime was 4.6 seconds. Once again, memory accesses (at least on the Apple M1 CPU) are expensive. On a different system, such as a Pi 3, 4, or 5, you will get different results.

Once you can convert a single byte to a pair of hexadecimal characters, creating a string, output to the display is straightforward. We can call the `btoh` (byte to hex) function for each byte in the number and store the corresponding characters away in a string. With this function, you can write `btoStr` (byte to string), `hwtoStr` (half word to string), `wtoStr` (word to string), and `dtoStr` (double word to string) functions. This chapter expands several of the lower-level functions (`btoStr`, `hwtoStr`, and `wtoStr`) and uses procedure calls to the smaller functions for the larger-sized conversions (`dtoStr`). In Chapter 13, I discuss macros that will provide another way to easily expand these functions.

The approach this book takes is to try to write fast conversion code. If you would prefer to save space rather than increase speed, see the following "Reducing Code Size" box for details.

REDUCING CODE SIZE

To reduce code size and make these functions easier to write, you can code `hwtoStr` to call `btoStr` twice (and concatenate their output), code `wtoStr` to call `hwtoStr` twice, code `dtoStr` to call `wtoStr` twice, and so on. This produces functions with just a few instructions each, but the performance suffers. For example, assuming you've written the functions in this fashion, consider a call to `dtoStr`: it calls `wtoStr` twice; `wtoStr` calls `hwtoStr` twice, which calls `btoStr` twice (which ultimately calls the `btoh` function). This means that `dtoStr` makes 22 total calls. As most of these (except possibly `btoh`) are higher-level functions, they should preserve any registers they modify. If each function saves a couple of registers, this results in 28 writes to, and 28 reads from, memory to preserve and restore the register values. As you saw with the `btoh` function, preserving registers can be expensive.

A higher-performance alternative, albeit requiring more code, is to call `btoh` (with no register preservation) as many times as necessary to convert each of the data types to the appropriately sized string. The higher-level function can preserve the registers exactly once and make multiple calls to `btoh` (such as the high-performance `btoh_x1` function that doesn't preserve registers). Within that function (for example, `dtoStr`), it is aware that `btoh` might wipe out X0 and X1; the higher-level function preserves those registers, so its caller doesn't have to, and treats them as volatile across calls to `btoh`. This way, the registers are saved and restored only once across the call to the higher-level function.

Another possible performance improvement is to ditch the `btoh` function entirely and expand it inline in the higher-level functions. Before software engineers recoil in horror from this suggestion, remember these are very low-level functions that are generally part of a library rather than an application program

(other than via linking in the appropriate library). For low-level library code, optimization generally pays off handsomely. For those with doubts, consider using macros (see Chapter 13), which give you the benefit of straight-line efficient code along with the structured nature of procedure calls.

All the binary-to-hexadecimal string functions will accept two parameters: a value to convert in the X1 register, and a pointer to a string buffer to hold the string result in X0. These functions will assume that the buffer is sufficiently large to hold the string result: `btoStr` requires a 3-character buffer, `hwtoStr` requires a 5-character buffer, `wtoStr` requires a 9-character buffer, and `dtoStr` requires a 17-character buffer. Each byte in the value requires two characters in the buffer. In addition to the character data, the buffer must also include 1 byte for the zero-terminating byte. The caller is responsible for ensuring that the buffer is large enough.

To implement these four hexadecimal-to-string functions, I'll start by writing four hexadecimal-to-buffer functions. There are two differences between the `*tobuf` and `*tostr` functions (where the * indicates a substitution of `b`, `hw`, `w`, or `d`, as per regular expression syntax):

- The `*tobuf` functions do not preserve any registers. They modify the values in X0 and X2.
- The `*tobuf` functions leave X0 pointing at the zero-terminating byte at the end of the string, which is often useful; the `*tostr` functions preserve X0's value (pointing at the first character of the output buffer).

I will also take this opportunity to introduce another assembly language feature: multiple entry points to a function. The `btobuf`, `htobuf`, `wtobuf`, and `dtobuf` functions all contain common code. Listing 9-3 merges all these functions into a single function (`dtobuf`) with separate entry points into the code sequence for the other three functions.

```
// Listing9-3.S

  Usual header code snipped

// dtobuf
//
// Convert a dword to a string of 16 hexadecimal digits.
//
// Inputs:
//  X0-     Pointer to the buffer. Must have at least
//          17 bytes available.
//  X1-     Value to convert
//
// Outputs:
//  X0-     Points at zero-terminating byte at the end
//          of the converted string
//
```

```
// Note:    This function does not preserve any registers.
//          It is the caller's responsibility to preserve
//          registers.
//
//          Registers modified: X0, X2

            proc    dtobuf

#define AtoF   ('A'-'9'-1)

            // Process the HO nibble:

          ❶ lsr     x2, x1, #60
            orr     w2, w2, #'0'    // Convert to 0x30 to 0x3F.
            cmp     w2, #'9'        // See if 0x3A to 0x3F.
            bls     dec15           // Skip if 0 to 9.
            add     w2, w2, #AtoF   // If it was A to F
 dec15:
            strb    w2, [x0], #1    // Store byte to memory.

            // Process nibble 14:

            lsr     x2, x1, #56     // See comments for HO nibble.
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec14
            add     w2, w2, #AtoF
dec14:      strb    w2, [x0], #1

            // Process nibble 13:

            lsr     x2, x1, #52
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec13
            add     w2, w2, #AtoF
dec13:      strb    w2, [x0], #1

            // Process nibble 12:

            lsr     x2, x1, #48
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec12
            add     w2, w2, #AtoF
dec12:      strb    w2, [x0], #1

            // Process nibble 11:

            lsr     x2, x1, #44
            and     x2, x2, 0xf
```

```
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec11
            add     w2, w2, #AtoF
dec11:      strb    w2, [x0], #1

            // Process nibble 10:

            lsr     x2, x1, #40
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec10
            add     w2, w2, #AtoF
dec10:      strb    w2, [x0], #1

            // Process nibble 9:

            lsr     x2, x1, #36
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec9
            add     w2, w2, #AtoF
dec9:       strb    w2, [x0], #1

            // Process nibble 8:

            lsr     x2, x1, #32
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec8
            add     w2, w2, #AtoF
dec8:       strb    w2, [x0], #1

// Entry point for wtobuf
//
// wtobuf
//
// Convert a word to a string of 8 hexadecimal digits.
//
// Inputs:
//  X0-     Pointer to the buffer. Must have at least
//          9 bytes available.
//  X1-     Value to convert
//
// Outputs:
//  X0-     Points at zero-terminating byte at the end
//          of the converted string
//
// Note:    This function does not preserve any registers.
//          It is the caller's responsibility to preserve
//          registers.
```

```
//
//          Registers modified: X0, X2

❷ wtobuf:
            // Process nibble 7:

            lsr     x2, x1, #28 // See comments for nibble 15.
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec7
            add     w2, w2, #AtoF
dec7:       strb    w2, [x0], #1

            // Process nibble 6:

            lsr     x2, x1, #24
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec6
            add     w2, w2, #AtoF
dec6:       strb    w2, [x0], #1

            // Process nibble 5:

            lsr     x2, x1, #20
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec5
            add     w2, w2, #AtoF
dec5:       strb    w2, [x0], #1

            // Process nibble 4:

            lsr     x2, x1, #16
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec4
            add     w2, w2, #AtoF
dec4:       strb    w2, [x0], #1

// Entry point for htobuf:
//
// htobuf
//
// Convert a half word to a string of 4 hexadecimal digits.
//
// Inputs:
//  X0-     Pointer to the buffer. Must have at least
//          5 bytes available.
//  X1-     Value to convert
```

```
//
// Outputs:
//  X0-     Points at zero-terminating byte at the end
//          of the converted string
//
// Note:    This function does not preserve any registers.
//          It is the caller's responsibility to preserve
//          registers.
//
//          Registers modified: X0, X2

❸ htobuf:
            // Process nibble 3:

            lsr     x2, x1, #12 // See comments for nibble 15.
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec3
            add     w2, w2, #AtoF
dec3:       strb    w2, [x0], #1

            // Process nibble 2:

            lsr     x2, x1, #8
            and     x2, x2, 0xf
            orr     w2, w2, #'0'
            cmp     w2, #'9'
            bls     dec2
            add     w2, w2, #AtoF
dec2:       strb    w2, [x0], #1

// Entry point for btobuf:
//
// btobuf
//
// Convert a byte to a string of two hexadecimal digits.
//
// Inputs:
//  X0-     Pointer to the buffer. Must have at least
//          3 bytes available.
//  X1-     Value to convert
//
// Outputs:
//  X0-     Points at zero-terminating byte at the end
//          of the converted string
//
// Note:    This function does not preserve any registers.
//          It is the caller's responsibility to preserve
//          registers.
//
//          Registers modified: X0, X2
```

```
                // Process nibble 1:

❹ btobuf:
                lsr     x2, x1, #4      // See comments for nibble 15.
                and     x2, x2, 0xf
                orr     w2, w2, #'0'
                cmp     w2, #'9'
                bls     dec1
                add     w2, w2, #AtoF
  dec1:         strb    w2, [x0], #1

                // Process LO nibble:

                and     x2, x1, 0xf
                orr     x2, x2, #'0'
                cmp     w2, #'9'
                bls     dec0
                add     w2, w2, #AtoF
  dec0:         strb    w2, [x0], #1

                strb    wzr, [x0]       // Zero-terminate.
                ret
                endp    dtobuf
```

The `dtobuf` function begins by processing the HO nibble (nibble 15) of the dword ❶. For performance reasons, this code uses an unrolled loop, processing each nibble individually. Each nibble uses the standard algorithm for converting a binary value to a hexadecimal character.

After this code processes the HO eight hex digits, you'll notice an entry point for the `wtobuf` function ❷. Code calling `wtobuf` transfers control into the middle of the `dtobuf` function (literally). This works because `dtobuf` doesn't push anything onto the stack or otherwise alter the environment that would require special work by `wtobuf` on entry. Likewise, entry points for `htobuf` ❸ and `btobuf` ❹ are at nibbles 3 and 1, respectively. By merging these functions into a single section of code, you save all the code that would be used for `wtobuf`, `htobuf`, and `btobuf`.

I made several failed attempts at optimizing this code. First, I tried saving 8 bytes in a register and wrote the data to memory a dword at a time rather than a byte at a time. This ran slower (on my Mac mini M1). I also tried eliminating branches in the code by using `csel` instructions. Surprisingly, that code ran slower too. I even tried using a `ubfx` instruction (see Chapter 12), which still ran slower than the code with branches. I timed these versions on a Mac mini M1 and a Raspberry Pi 400. While the timings on the two machines varied greatly, the relative performance of the three algorithms remained the same (the branch version was always faster). Sometimes, getting clever with different algorithms can hurt you. That's why you should always test the performance of your code (preferably on multiple architectures).

With the `*tobuf` functions out of the way, writing the `*toStr` functions is relatively easy. The `*toStr` functions simply call the `*tobuf` functions and

preserve the registers that the *tobuf functions modify. Listing 9-4 provides the code for these functions (note that *Listing9-4.S*, from the online files, also includes the code for the dtobuf function; to avoid redundancy, I've removed that code from the listing).

```
// Listing9-4.S
//
// btoStr, htoStr, wtoStr, and dtoStr functions
// Also includes btobuf, htobuf, wtobuf, and
// dtobuf functions

            #include    "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-4"

            .data

// Buffer space used by main program

buffer:     .space      256,0

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

❶ Insert the code for dtobuf here. See Listing 9-3.

// btoStr-
//
// Inputs:
//
//  X0- Pointer to buffer that will hold the result
//      (must allocate at least 3 bytes for buffer)
//  X1- Value to print (in L0 byte)
//
// Outputs:
//
//  Buffer pointed at by X0 receives the two-character
//  conversion of the value in X1 to a hexadecimal string.
//
//  Preserves all registers.

          ❷ proc    btoStr

            str     x2, [sp, #-16]!
            stp     x0, lr, [sp, #-16]!
```

```
            bl      btobuf

            // Restore registers and return:

            ldp     x0, lr, [sp], #16
            ldr     x2, [sp], #16
            ret
            endp    btoStr

// htoStr
//
// Inputs:
//
//  X0- Pointer to buffer that will hold the result
//      (must allocate at least 5 bytes for buffer)
//  X1- Value to print (in LO hword)
//
// Outputs:
//
//  Buffer pointed at by X0 receives the four-character
//  conversion of the hword value in X1 to a hexadecimal string.
//
//  Preserves all registers

          ❸ proc    htoStr

            str     x2, [sp, #-16]!
            stp     x0, lr, [sp, #-16]!

            bl      htobuf

            // Restore registers and return:

            ldp     x0, lr, [sp], #16
            ldr     x2, [sp], #16
            ret
            endp    htoStr

// wtoStr
//
// Inputs:
//
//  X0- Pointer to buffer that will hold the result
//      (must allocate at least 9 bytes for buffer)
//  X1- Value to print (in LO word)
//
// Outputs:
//
//  Buffer pointed at by X0 receives the eight-character
//  conversion of the word value in X1 to a hexadecimal string.
//
//  Preserves all registers

          ❹ proc    wtoStr
```

```
            str     x2, [sp, #-16]!
            stp     x0, lr, [sp, #-16]!

            bl      wtobuf

            // Restore registers and return:

            ldp     x0, lr, [sp], #16
            ldr     x2, [sp], #16
            ret
            endp    wtoStr

// dtoStr
//
// Inputs:
//
//  X0- Pointer to buffer that will hold the result
//      (must allocate at least 17 bytes for buffer)
//  X1- Value to print
//
// Outputs:
//
//  Buffer pointed at by X0 receives the 16-character
//  conversion of the dword value in X1 to a hexadecimal string.
//
//  Preserves all registers

          ❺ proc    dtoStr

            str     x2, [sp, #-16]!
            stp     x0, lr, [sp, #-16]!

            bl      dtobuf

            // Restore registers and return:

            ldp     x0, lr, [sp], #16
            ldr     x2, [sp], #16
            ret
            endp    dtoStr

// Utility functions to print bytes, hwords, words, and dwords:

pbStr:      wastr   "Byte=%s\n"

            proc    pByte

            locals  pb
            qword   pb.saveX0X1
            byte    pb.buffer, 32
            byte    pb.stkSpace, 64
            endl    pb

            enter   pb.size
            stp     x0, x1, [fp, #pb.saveX0X1]
```

```
                mov     x1, x0
                add     x0, fp, #pb.buffer  // lea x0, stkSpace
                bl      btoStr

                lea     x0, pbStr
                add     x1, fp, #pb.buffer
                mstr    x1, [sp]
                bl      printf

                ldp     x0, x1, [fp, #pb.saveX0X1]
                leave
                endp    pByte

phStr:          wastr   "Hword=%s\n"

                proc    pHword

                locals  ph
                qword   ph.saveX0X1
                byte    ph.buffer, 32
                byte    ph.stkSpace, 64
                endl    ph

                enter   ph.size
                stp     x0, x1, [fp, #ph.saveX0X1]

                mov     x1, x0
                add     x0, fp, #ph.buffer  // lea x0, stkSpace
                bl      htoStr

                lea     x0, phStr
                add     x1, fp, #ph.buffer
                mstr    x1, [sp]
                bl      printf

                ldp     x0, x1, [fp, #ph.saveX0X1]
                leave
                endp    pHword

pwStr:          wastr   "Word=%s\n"

                proc    pWord

                locals  pw
                qword   pw.saveX0X1
                byte    pw.buffer, 32
                byte    pw.stkSpace, 64
                endl    pw

                enter   pw.size
                stp     x0, x1, [fp, #pw.saveX0X1]

                mov     x1, x0
                add     x0, fp, #pw.buffer  // lea x0, stkSpace
                bl      wtoStr
```

```
                lea     x0, pwStr
                add     x1, fp, #pw.buffer
                mstr    x1, [sp]
                bl      printf

                ldp     x0, x1, [fp, #pw.saveX0X1]
                leave
                endp    pWord

pdStr:          wastr   "Dword=%s\n"

                proc    pDword

                locals  pd
                qword   pd.saveX0X1
                byte    pd.buffer, 32
                byte    pd.stkSpace, 64
                endl    pd

                enter   pd.size
                stp     x0, x1, [fp, #pd.saveX0X1]

                mov     x1, x0
                add     x0, fp, #pd.buffer  // lea x0, stkSpace
                bl      dtoStr

                lea     x0, pdStr
                add     x1, fp, #pd.buffer
                mstr    x1, [sp]
                bl      printf

                ldp     x0, x1, [fp, #pd.saveX0X1]
                leave
                endp    pDword

// Here is the asmMain function:

                proc    asmMain, public

                // Local storage:

                locals  am
                byte    stackspace, 64
                endl    am

                enter   am.size             // Create activation record.

                ldr     x0, =0x0123456789abcdef
                bl      pByte
                bl      pHword
                bl      pWord
                bl      pDword

                leave
```

```
        ret

        endp    asmMain
```

As noted, I've pulled the `dtobuf` function out of this listing; insert that code ❶. The `btoStr` function ❷ saves the X0, X2, and LR registers on the stack (the registers that will be modified by calls to the `*tobuf` functions), calls the `btobuf` function to write the two hex digits to the buffer pointed at by X0, then restores the registers and returns. The code does largely the same for `htoStr` ❸, `wtoStr` ❹, and `dtoStr` ❺, the only difference being the conversion function they call.

Here's the `build` command and sample output for the program in Listing 9-4:

```
$ ./build Listing9-4
$ ./Listing9-4
Calling Listing9-4:
Byte=EF
Hword=CDEF
Word=89ABCDEF
Dword=0123456789ABCDEF
Listing9-4 terminated
```

As the assembly code appearing in this book calls C/C++ standard library functions for I/O, these binary-to-hexadecimal-string functions will all produce zero-terminated C-compatible strings. They are easy enough to modify to produce other string formats, if need be. See Chapter 14 for more on string functions.

9.1.2 Extended-Precision Hexadecimal Values to Strings

Extended-precision hexadecimal-to-string conversion is easy: it's simply an extension of the normal hexadecimal conversion routines from the previous section. For example, Listing 9-5 is a 128-bit hexadecimal conversion function, `qtoStr`, which expects a pointer to a 128-bit value in X2:X1 and a pointer to a buffer in X0. *Listing9-5.S* is largely based on *Listing9-4.S*; to avoid redundancy, I've included just the `qtoStr` function here.

```
// Listing9-5.S
//
// qtoStr
//
// Inputs:
//
//  X0-     Pointer to buffer that will hold the result
//          (must allocate at least 33 bytes for buffer)
//  X2:X1-  Value to print
//
// Outputs:
//
```

```
//  Buffer pointed at by X0 receives the 32-character
//  conversion of the dword value in X2:X1 to a hexadecimal string.
//
//  Preserves all registers

            proc    qtoStr

            str     x2, [sp, #-16]!
            stp     x0, lr, [sp, #-16]!
            str     x1, [sp, #-16]!     // Save for later.

            mov     x1, x2              // Convert HO dword first.
            bl      dtobuf
            ldr     x1, [sp], #16       // Restore X1 value.
            bl      dtobuf

            // Restore registers and return:

            ldp     x0, lr, [sp], #16
            ld4     x2, [sp], #16
            ret
            endp    qtoStr
```

The function in Listing 9-5 calls `dtobuf` twice to convert the 128-bit qword value to a string by converting first the HO dword, then the LO dword, and concatenating their results. To extend this conversion to any number of bytes, simply convert the HO bytes down to the LO bytes of the large object.

9.1.3 Unsigned Decimal Values to Strings

Decimal output is a little more complicated than hexadecimal output because, unlike for hexadecimal values, the HO bits of a binary number affect the LO digits of the decimal representation. Therefore, you must create the decimal representation for a binary number by extracting one decimal digit at a time from the number.

The most common solution for unsigned decimal output is to successively divide the value by 10 until the result becomes 0. The remainder after the first division is a value in the range 0 to 9, which corresponds to the LO digit of the decimal number. Successive divisions by 10 (and their corresponding remainder) extract successive digits from the number.

Iterative solutions to this problem generally allocate storage for a string of characters large enough to hold the entire number. The code then extracts the decimal digits in a loop and places them in the string one by one. At the end of the conversion process, the routine prints the characters in the string in reverse order (remember, the divide algorithm extracts the LO digits first and the HO digits last, the opposite of the way you need to print them).

This section employs a *recursive solution* because it is a little more elegant. This solution begins by dividing the value by 10 and saving the remainder in a local variable. If the quotient is not 0, the routine recursively calls

itself to output any leading digits first. On return from the recursive call (which outputs all the leading digits), the recursive algorithm outputs the digit associated with the remainder to complete the operation. For example, here's how the operation works when printing the decimal value 789:

1. Divide 789 by 10. The quotient is 78, and the remainder is 9.
2. Save the remainder (9) in a local variable and recursively call the routine with the quotient.
3. Recursive entry 1: divide 78 by 10. The quotient is 7, and the remainder is 8.
4. Save the remainder (8) in a local variable and recursively call the routine with the quotient.
5. Recursive entry 2: divide 7 by 10. The quotient is 0, and the remainder is 7.
6. Save the remainder (7) in a local variable. Because the quotient is 0, don't call the routine recursively.
7. Output the remainder value saved in the local variable (7). Return to the caller (recursive entry 1).
8. Return to recursive entry 1: output the remainder value saved in the local variable in recursive entry 1 (8). Return to the caller (original invocation of the procedure).
9. Original invocation: output the remainder value saved in the local variable in the original call (9). Return to the original caller of the output routine.

Listing 9-6 provides an implementation of this recursive algorithm for 64-bit unsigned integers.

```
// Listing9-6.S
//
// u64toBuf function

            #include    "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-6"
fmtStr1:    .asciz      "Value(%llu) = string(%s)\n"

            .align      3
qwordVal:   .dword      0x1234567890abcdef
            .dword      0xfedcba0987654321

            .data
buffer:     .space      256,0

            .code
            .extern     printf

// Return program title to C++ program:
```

```
            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// u64ToStr
//
//  Converts a 64-bit unsigned integer to a string
//
//  Inputs:
//      X0-     Pointer to buffer to receive string
//      X1-     Unsigned 64-bit integer to convert
//
//  Outputs:
//      Buffer- Receives the zero-terminated string
//
//  Buffer must have at least 21 bytes allocated for it.
//  This function preserves all registers.

          ❶ proc    u64ToStr
            stp     x0, x1, [sp, #-16]!
            stp     x2, x3, [sp, #-16]!
            str     lr, [sp, #-16]!

            bl      u64ToBuf

            ldr     lr, [sp], #16
            ldp     x2, x3, [sp], #16
            ldp     x0, x1, [sp], #16
            ret
            endp    u64ToStr

// u64ToBuf
//
//  Converts a 64-bit unsigned integer to a string
//
//  Inputs:
//      X0-     Pointer to buffer to receive string
//      X1-     Unsigned 64-bit integer to convert
//
//  Outputs:
//      X0-     Points at zero-terminating byte
//      Buffer- Receives the zero-terminated string
//
//  Buffer must have at least 21 bytes allocated for it.
//
//  Caller must preserve X0, X1, X2, and X3!

          ❷ proc    u64ToBuf
            cmp     x1, xzr         // See if X1 is 0.
            bne     u64ToBufRec

            // Special case for zero, just write
            // "0" to the buffer. Leave X0 pointing
```

```
                // at the zero-terminating byte.

                mov     w1, #'0'
                strh    w1, [x0], #1    // Also emits zero byte
                ret
                endp    u64ToBuf

// u64ToBufRec is the recursive version that handles
// nonzero values:

            ❸ proc    u64ToBufRec
              stp     x2, lr, [sp, #-16]! // Preserve remainder.

              // Divide X1 by 10 and save quotient and remainder:

            ❹ mov     x2, #10
              udiv    x3, x1, x2      // X3 = quotient
              msub    x2, x3, x2, x1  // X2 = remainder

              // Make recursive call if quotient is not 0:

              cmp     x3, xzr
              beq     allDone

            ❺ mov     x1, x3              // Set up for call.
              bl      u64ToBufRec

              // When this function has processed all the
              // digits, write them to the buffer. Also
              // write a zero-terminating byte, in case
              // this is the last digit to output.

❻ allDone:    orr     w2, w2, #'0'    // Convert to char.
              strh    w2, [x0], #1    // Bump pointer after store.
            ❼ ldp     x2, lr, [sp], #16
              ret
              endp    u64ToBufRec

// Here is the "asmMain" function.

              proc    asmMain, public

              enter   64              // Reserve space on stack.

// Test u64ToBuf:

              mov     x1, 0xFFFF
              movk    x1, 0xFFFF, lsl #16
              movk    x1, 0xFFFF, lsl #32
              movk    x1, 0xFFFF, lsl #48
              lea     x0, buffer
              bl      u64ToStr

              lea     x2, buffer
```

```
        mstr    x2, [sp, #8]
        mov     x1, 0xFFFF
        movk    x1, 0xFFFF, lsl #16
        movk    x1, 0xFFFF, lsl #32
        movk    x1, 0xFFFF, lsl #48
        mstr    x1, [sp]
        lea     x0, fmtStr1
        bl      printf

        leave
        ret
        endp    asmMain
```

The `u64toStr` function ❶ is a facade that preserves the registers while calling the `u64ToBuf` procedure. The `u64ToBuf` function ❷ handles the special case when X1 contains 0 (the recursive code terminates when the result is 0). If X1 is 0 upon entry, this code immediately writes a `'0'` character to the output buffer, increments X0, and returns. If X1 is nonzero, it transfers control to the recursive `u64toBufRec` function ❸ to process the value. For performance reasons, `u64ToBufRec` preserves only X2 (which contains the remainder value on recursive calls) and LR.

The recursive function computes the quotient and remainder ❹. The quotient is left in X3, and the remainder is in X2. If the quotient was nonzero, there are still more HO digits to process: copy the quotient into X1 and make the recursive call to `u64toBufRec` ❺. On the return from the recursive call ❻ (or if the recursive call was skipped), all HO digits have been emitted to the buffer, so convert the current digit to a character and add it to the end of the buffer. Note that the post-increment addressing mode automatically increments X0 to point at the zero-terminated byte emitted by the `strh` instruction. The code restores the value in X2 ❼, in the event that this was a recursive call.

Here's the `build` command and sample output for Listing 9-6:

```
$ ./build Listing9-6
$ ./Listing9-6
Calling Listing9-6:
Value(18446744073709551615) = string(18446744073709551615)
Listing9-6 terminated
```

Unlike hexadecimal output, there's no need to provide a byte-size, hword-size, or word-size numeric-to-decimal-string conversion function. Simply zero-extending the smaller values to 64 bits is sufficient. Unlike the hexadecimal conversions, no leading zeros are emitted by the `u64toStr` function, so the output is the same for all sizes of variables (64 bits and smaller).

This code has several opportunities for optimization. Since decimal-to-string conversions are common (most program output uses this function) and the algorithm is not as fast as hexadecimal conversion, optimizing this code is probably worthwhile.

It's easy enough to get rid of the recursion and do an iterative version of `u64toStr`. This eliminates the need to preserve the registers and return

address on multiple recursive calls (typically, one recursive call for each digit converted) and having to build the activation record on each call. Listing 9-7 takes this one step further, unraveling the loop (up to 20 iterations, one for each possible digit).

```
// Listing9-7.S
//
// u64toStr function (nonrecursive, straight-line
// code version)

            #include    "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-7"
fmtStr1:    .asciz      "low=%s, "
fmtStr2:    .asciz      "hi=%s\n"

loData:     .dword      0, 1, 10, 100, 1000, 10000, 100000
            .dword      1000000, 10000000, 100000000
            .dword      1000000000, 10000000000, 100000000000
            .dword      1000000000000, 10000000000000
            .dword      100000000000000, 1000000000000000
            .dword      10000000000000000, 100000000000000000
            .dword      1000000000000000000, 10000000000000000000
            .equ        dataCnt, .-loData

hiData:     .dword      9, 9, 99, 999, 9999, 99999, 999999
            .dword      9999999, 99999999, 999999999
            .dword      9999999999, 99999999999, 999999999999
            .dword      9999999999999, 99999999999999
            .dword      999999999999999, 9999999999999999
            .dword      99999999999999999, 999999999999999999
            .dword      9999999999999999999
            .dword      -1

            .data
buffer:     .space      256, 0

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// u64ToBuf
//
//  Converts a 64-bit unsigned integer to a string
//
//  Inputs:
//      X0-     Pointer to buffer to receive string
//      X1-     Unsigned 64-bit integer to convert
```

```
//
//  Outputs:
//      Buffer- Receives the zero-terminated string
//      X0-     Points at zero-terminating byte in string
//
//  Buffer must have at least 21 bytes allocated for it.
//  Note: Caller is responsible for preserving X0-X7!

        ❶ proc    u64ToBuf

        ❷ mov     x4, #10
          mov     x5, xzr
          mov     x6, xzr
          mov     x7, xzr

          // Handle the LO digit here:

        ❸ udiv    x2, x1, x4      // X2 = quotient
          msub    x3, x2, x4, x1  // X3 = remainder
          orr     x3, x3, #'0'
          orr     x5, x3, x5, lsl #8
          cmp     x2, #0
          beq     allDone1

          // Handle the 10's digit here:

        ❹ udiv    x1, x2, x4      // X1 = quotient
          msub    x3, x1, x4, x2  // X3 = remainder
          orr     x3, x3, #'0'
          orr     x5, x3, x5, lsl #8
          cmp     x1, #0
          beq     allDone2

          // Handle the 100's digit here:

          udiv    x2, x1, x4      // X2 = quotient
          msub    x3, x2, x4, x1  // X3 = remainder
          orr     x3, x3, #'0'
          orr     x5, x3, x5, lsl #8
          cmp     x2, #0
          beq     allDone3

          // Handle the 1000's digit here:

          udiv    x1, x2, x4      // X1 = quotient
          msub    x3, x1, x4, x2  // X3 = remainder
          orr     x3, x3, #'0'
          orr     x5, x3, x5, lsl #8
          cmp     x1, #0
          beq     allDone4

          // Handle the 10,000's digit here:

          udiv    x2, x1, x4      // X2 = quotient
          msub    x3, x2, x4, x1  // X3 = remainder
```

```
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8
cmp     x2, #0
beq     allDone5

// Handle the 100,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8
cmp     x1, #0
beq     allDone6

// Handle the 1,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x6, x3, #'0'
cmp     x2, #0
beq     allDone7

// Handle the 10,000,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x1, #0
beq     allDone8

// Handle the 100,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x2, #0
beq     allDone9

// Handle the 1,000,000,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x1, #0
beq     allDone10

// Handle the 10,000,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
```

```
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x2, #0
beq     allDone11

// Handle the 100,000,000,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x1, #0
beq     allDone12

// Handle the 1,000,000,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x2, #0
beq     allDone13

// Handle the 10,000,000,000,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x1, #0
beq     allDone14

// Handle the 100,000,000,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x7, x3, #'0'
orr     x6, x3, x6, lsl #8
cmp     x2, #0
beq     allDone15

// Handle the 1,000,000,000,000,000's digit here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x7, x3, x7, lsl #8
cmp     x1, #0
beq     allDone16

// Handle the 10,000,000,000,000,000's digit here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
```

```
                orr     x3, x3, #'0'
                orr     x7, x3, x7, lsl #8
                cmp     x2, #0
                beq     allDone17

                // Handle the 100,000,000,000,000,000's digit here:

                udiv    x1, x2, x4      // X1 = quotient
                msub    x3, x1, x4, x2  // X3 = remainder
                orr     x3, x3, #'0'
                orr     x7, x3, x7, lsl #8
                cmp     x1, #0
                beq     allDone18

                // Handle the 1,000,000,000,000,000,000's digit here:

                udiv    x2, x1, x4      // X2 = quotient
                msub    x3, x2, x4, x1  // X3 = remainder
                orr     x3, x3, #'0'
                orr     x7, x3, x7, lsl #8
                cmp     x2, #0
                beq     allDone19

❺               udiv    x1, x2, x4      // X1 = quotient
                msub    x3, x1, x4, x2  // X3 = remainder
                orr     x3, x3, #'0'
                orr     x7, x3, x7, lsl #8

allDone20:      str     x7, [x0], #6
                str     x6, [x0], #8
                str     x5, [x0], #7
                ret

                // When this function has processed all the
                // digits, write them to the buffer. Also
                // write a zero-terminating byte, in case
                // this is the last digit to output.

❻ allDone1:     strh    w5, [x0], #1
                ret

  allDone2:     strh    w5, [x0], #2
                strb    wzr, [x0]
                ret

  allDone3:     str     w5, [x0], #3
                ret

  allDone4:     str     w5, [x0], #4
                strb    wzr, [x0]
                ret

  allDone5:     str     x5, [x0], #4
                lsr     x5, x5, #32
```

```
            strh    w5, [x0], #1
            ret

allDone6:   str     w5, [x0], #4
            lsr     x5, x5, #32
            strh    w5, [x0], #2
            strb    wzr, [x0]
            ret

❼ allDone7: strb    w6, [x0], #1
            str     x5, [x0], #7
            ret

allDone8:   strh    w6, [x0], #2
            str     x5, [x0], #7    // Writes an extra garbage byte
            ret

allDone9:   str     w6, [x0], #3
            str     x5, [x0], #7
            ret

allDone10:
            str     w6, [x0], #4
            str     x5, [x0], #7
            ret

allDone11:
            str     x6, [x0], #5
            str     x5, [x0], #7
            ret

allDone12:
            str     x6, [x0], #6
            str     x5, [x0], #7
            ret

allDone13:
            str     x6, [x0], #7
            str     x5, [x0], #7
            ret

allDone14:
            str     x6, [x0], #8
            str     x5, [x0], #7
            ret

❽ allDone15:
            strb    w7, [x0], #1
            str     x6, [x0], #8
            str     x5, [x0], #7
            ret

allDone16:
            strh    w7, [x0], #2
            str     x6, [x0], #8
```

```
            str     x5, [x0], #7
            ret

  allDone17:
            str     w7, [x0], #3
            str     x6, [x0], #8
            str     x5, [x0], #7
            ret

  allDone18:
            str     w7, [x0], #4
            str     x6, [x0], #8
            str     x5, [x0], #7
            ret

  allDone19:
            str     x7, [x0], #5
            str     x6, [x0], #8
            str     x5, [x0], #7
            ret
            endp    u64ToBuf

// u64ToStr
//
//  Version of u64ToBuf that preserves the registers

          ❾ proc    u64ToStr
            stp     x0, x1, [sp, #-16]! // Preserve registers.
            stp     x2, x3, [sp, #-16]!
            stp     x4, x5, [sp, #-16]!
            stp     x6, x7, [sp, #-16]!
            str     lr, [sp, #-16]!
            bl      u64ToBuf
            ldr     lr, [sp], #16
            ldp     x6, x7, [sp], #16   // Restore registers.
            ldp     x4, x5, [sp], #16
            ldp     x2, x3, [sp], #16
            ldp     x0, x1, [sp], #16
            ret
            endp    u64ToStr

// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            qword   am.x20_x21
            dword   am.x22
            byte    stk, 64
            endl    am

            enter   am.size             // Create act rec.
```

```
            // Preserve nonvolatile registers:

            stp     x20, x21, [fp, #am.x20_x21]
            str     x22, [fp, #am.x22]

            lea     x20, loData
            lea     x21, hiData
            mov     x22, xzr
loop:
            lea     x0, buffer
            ldr     x1, [x20, x22, lsl #3]
            bl      u64ToStr

            lea     x0, fmtStr1
            lea     x1, buffer
            mstr    x1, [sp]
            bl      printf

            lea     x0, buffer
            ldr     x1, [x21, x22, lsl #3]
            bl      u64ToStr

            lea     x0, fmtStr2
            lea     x1, buffer
            mstr    x1, [sp]
            bl      printf

            add     x22, x22, #1
            cmp     x22, #(dataCnt / 8)
            blo     loop

            ldr     x22, [fp, #am.x22]
            ldp     x20, x21, [fp, #am.x20_x21]

            leave
            endp    asmMain
```

The `u64ToBuf` function ❶ is a variant of `u64ToStr` that doesn't preserve any registers. It stomps on X0 through X7, and the caller is responsible for saving any registers it needs preserved.

This function initializes X4 with the constant 10 ❷, because each digit conversion will divide and multiply by this constant, which must be in a register. Reserving X4 for this constant spares the code from having to reload the constant all the time. This code zeros out X5, X6, and X7, which will hold the characters of the converted string; this also initializes the zero-terminating byte (which can be in various locations in these registers, depending on the number of output digits).

The function converts the binary number to a string of digits by using the same basic "divide and remainder" algorithm as did the program in Listing 9-6 ❸. The function divides the value by 10; the remainder is a value in the range 0 to 9 that the function converts to the corresponding ASCII character. The code shifts the converted digit into its final output

position in the X5, X6, or X7 register. Digits 1 through 6, the HO digits, wind up in X5; digits 7 through 14 in X6; and digits 15 through 20 in X7. Zero bytes fill in all the unused digit positions. For example, if the number has only three digits, X6 and X7 will contain 0, and bits 24 through 63 in X5 will all contain 0.

A separate sequence of divide/remainder instructions is used for each possible output digit in the conversion (hence the name *expanded/straight-line code*) ❹. The sequence is roughly the same for each digit conversion, though two variants alternate between the value in X1 and X2, as the quotient from the division becomes the value to divide in the next step. Whenever the quotient becomes 0, the conversion is complete, and control transfers to a different location to write the converted digits to the buffer. Only a single branch in the function will be taken, as these branches fall through to the next instruction sequence until the conversion is complete. Additionally, these digit conversion sequences may place the converted digit into a different output register based on the digit's final position.

If the code falls all the way through to digit 20, there is no test for a 0 result; the quotient will always be 0 at that point, so the function simply stores away the digits into the buffer and returns ❺.

If the number has six digits or fewer, the function writes the characters in X5 to the buffer ❻. X5 will always contain the LO digits of the number. By placing a maximum of six characters in X5, the HO 2 bytes of X5 will always be 0 (and provide the zero-terminating byte for larger strings). For numbers with fewer than six digits, the code must explicitly write a zero-terminating byte to the buffer. For values with 7 to 14 digits, the function writes out registers X6 and X5 (in that order) to the buffer ❼. X5 provides the zero-terminating byte, so the code doesn't need to explicitly write any 0 bytes. For values with 15 or more digits, the code writes out the data in registers X7, X6, and X5 (X5 provides the zero-terminating byte) ❽.

The actual `u64ToStr` function ❾ is a short facade that preserves all the register values across a call to `u64ToBuf`. By breaking `u64ToStr` into these two functions, it is possible to call `u64ToBuf` directly if you want to leave X0 pointing at the end of the string (though you must preserve X1 through X7 if necessary). Also, putting the register preservation code in `u64ToStr` allows the `u64ToBuf` code to avoid restoring registers before all the `ret` instructions (or avoid yet another branch to code that handles restoring the registers).

Here's the `build` command and sample output from Listing 9-7:

```
$ ./build Listing9-7
$ time ./Listing9-7
Calling Listing9-7:
low=0, hi=9
low=1, hi=9
low=10, hi=99
low=100, hi=999
low=1000, hi=9999
low=1000, hi=9999
low=100000, hi=999999
low=1000000, hi=9999999
```

```
low=10000000, hi=99999999
low=100000000, hi=999999999
low=1000000000, hi=9999999999
low=10000000000, hi=99999999999
low=100000000000, hi=999999999999
low=1000000000000, hi=9999999999999
low=10000000000000, hi=99999999999999
low=100000000000000, hi=999999999999999
low=1000000000000000, hi=9999999999999999
low=10000000000000000, hi=99999999999999999
low=100000000000000000, hi=999999999999999999
low=1000000000000000000, hi=18446744073709551615
Listing9-7 terminated
```

I modified both versions of `u64toStr` in order to time their execution. For the recursive version, I got the following timing on my Mac mini:

```
Listing9-7a  404.58s user 0.42s system 99% cpu 6:46.25 total
```

For the straight-line code, the runtime was as follows:

```
Listing9-7a  173.60s user 0.15s system 99% cpu 2:53.78 total
```

The latter code ran about 2.3 times faster than the recursive version, a big win.

I also created a version of `u64ToStr` that first counted the number of output digits (using a binary search), then branched to the appropriate code to convert exactly that many digits. Alas, the code ran slightly slower than Listing 9-7. I also tried a variant that emitted the HO digits first (dividing by 1e+19, the successively lower values by 10). It was a little faster than the digit count version, and a little slower than Listing 9-7. I've included the source code for both experiments in the online files for your perusal.

9.1.4 Signed Integer Values to Strings

To convert a signed integer value to a string, first check whether the number is negative. If it is, emit a hyphen (-) character and negate the value, then call the `u64toStr` function to finish the job. Listing 9-8 shows the relevant code.

```
// Listing9-8.S

Code taken from Listing 9-7 goes here.

// i64ToStr
//
//  Converts a signed 64-bit integer to a string
//  If the number is negative, this function will
//  print a '-' character followed by the conversion
//  of the absolute value of the number.
//
```

```
// Inputs:
//
//      X0- Pointer to buffer to hold the result.
//          Buffer should be capable of receiving
//          as many as 22 bytes (including zero-
//          terminating byte).
//      X1- Signed 64-bit integer to convert
//
// Outputs:
//
//      Buffer- Contains the converted string

            proc    i64ToStr

            locals  i64
            dword   i64.x0
            byte    i64.stk, 32
            endl    i64

            enter   i64.size

            // Need to preserve X1 in
            // case this code negates it.

            str     x1, [fp, #i64.x0]

            cmp     x1, #0
            bpl     isPositive

            mov     w1, #'-'    // Emit '-'
            strb    w1, [x0], #1

            // Negate X0 and convert
            // unsigned value to integer:

            ldr     x1, [fp, #i64.x0]
            neg     x1, x1

isPositive: bl      u64ToStr
            ldr     x1, [fp, #i64.x0]
            leave
            endp    i64ToStr

Code taken from Listing 9-7 goes here.
```

Listing 9-8 shows only the `i64ToStr` function (the rest of the program is taken from Listing 9-7). The full source code is available online.

9.1.5 Extended-Precision Unsigned Integers to Strings

The only operation in the entire string-conversion algorithm that requires extended-precision arithmetic is the divide-by-10 operation. Listing 9-9 implements a 128-bit decimal output routine utilizing this technique.

I modified the `div128` algorithm from Chapter 8 to do an explicit divide-by-10 operation (speeding `div128` up a little) and modified the recursive conversion routine from Listing 9-6 to perform the conversion.

```
// Listing9-9.S
//
// u128toStr function

            #include    "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-9"
fmtStr1:    .asciz      "Value = %s\n"

qdata:      .qword      1
            .qword      21
            .qword      302
            .qword      4003
            .qword      50004
            .qword      600005
            .qword      7000006
            .qword      80000007
            .qword      900000008
            .qword      1000000009
            .qword      11000000010
            .qword      120000000011
            .qword      1300000000012
            .qword      14000000000013
            .qword      150000000000014
            .qword      1600000000000015
            .qword      17000000000000016
            .qword      180000000000000017
            .qword      1900000000000000018
            .qword      20000000000000000019
            .qword      210000000000000000020
            .qword      2200000000000000000021
            .qword      23000000000000000000022
            .qword      240000000000000000000023
            .qword      2500000000000000000000024
            .qword      260000000000000000000000025
            .qword      270000000000000000000000026
            .qword      280000000000000000000000027
            .qword      290000000000000000000000028
            .qword      300000000000000000000000029
            .qword      310000000000000000000000030
            .qword      320000000000000000000000031
            .qword      330000000000000000000000032
            .qword      340000000000000000000000033
            .qword      350000000000000000000000034
            .qword      360000000000000000000000035
            .qword      370000000000000000000000036
            .qword      380000000000000000000000037
            .qword      390000000000000000000000038
            .qword      340282366920938463463374607431768211455
qcnt        =           (.-qdata)/16
```

```
            .data
buffer:     .space      256,0

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// div10
//
// This procedure does a general 128-bit / 10 division operation
// using the following algorithm (assume all variables except
// Remainder are 128-bit objects; Remainder is 64 bits):
//
// Quotient := Dividend;
// Remainder := 0;
// for i := 1 to NumberBits do
//
//  Remainder:Quotient := Remainder:Quotient SHL 1;
//  if Remainder >= 10 then
//
//     Remainder := Remainder - 10;
//     Quotient := Quotient + 1;
//
//  endif
// endfor
//
// Data passed:
//
// 128-bit dividend in X6:X5
//
// Data returned:
//
// 128-bit quotient in X6:X5
// 64-bit remainder in X4
//
// Modifies X1

          ❶ proc    div10

#define remainder  x4
#define dividendL  x5
#define dividendH  x6
#define quotientL  dividendL
#define quotientH  dividendH

// Initialize remainder with 0:

            mov     remainder, #0
```

```
// Copy the dividend to local storage:

            mov     w1, #128            // Count off bits in W0.

// Compute Remainder:Quotient := Remainder:Quotient LSL 1
//
// Note: adds x, x, x is equivalent to lsl x, x, #1
//       adcs x, x, x is equivalent to rol x, x, #1
//                    (if rol existed)
//
// The following four instructions perform a 256-bit
// extended-precision shift (left) dividend through
// remainder.

repeatLp:   adds    dividendL, dividendL, dividendL
            adcs    dividendH, dividendH, dividendH
            adc     remainder, remainder, remainder

// Do a comparison to see if the remainder
// is greater than or equal to 10:

            cmp     remainder, #10
            blo     notGE

// Remainder := Remainder - Divisor

isGE:       sub     remainder, remainder, #10

// Quotient := Quotient + 1

            adds    quotientL, quotientL, #1
            adc     quotientH, quotientH, xzr

// Repeat for 128 bits:

notGE:      subs    w1, w1, #1
            bne     repeatLp

            ret     // Return to caller.
            endp    div10

// u128toStr:
//
//  Converts a 128-bit unsigned integer to a string
//
//  Inputs:
//      X0-     Pointer to buffer to receive string
//      X1-     Points at the unsigned 128-bit integer to convert
//
//  Outputs:
//      Buffer- Receives the zero-terminated string
//
//  Buffer must have at least 40 bytes allocated for it.
```

```
            ❷ proc    u128toStr
              stp     x0, x1, [sp, #-16]!
              stp     x4, x5, [sp, #-16]!
              stp     x6, lr, [sp, #-16]!

              ldp     x5, x6, [x1]    // Test value for 0.
              orr     x4, x5, x6
              cmp     x4, xzr         // Z = 1 if X6:X5 is 0.
              bne     doRec128

              // Special case for zero, just write
              // "0" to the buffer

              mov     w4, #'0'
              strb    w4, [x0], #1
              b.al    allDone2

doRec128:     bl      u128toStrRec    // X6:X5 contain value.

              // Restore registers:

allDone2:     strb    wzr, [x0]       // Zero-terminating byte
              ldp     x6, lr, [sp], #16
              ldp     x4, x5, [sp], #16
              ldp     x0, x1, [sp], #16
              ret
              endp    u128toStr

// u128toStrRec is the recursive version that handles
// nonzero values.
//
// Value to convert is passed in X6:X5.

            ❸ proc    u128toStrRec
              stp     x4, lr, [sp, #-16]!

              // Convert LO digit to a character:

              bl      div10           // Quotient -> X6:X5, Rem -> W4

              // Make recursive call if quotient is not 0:

              orr     lr, x5, x6      // Use LR as a temporary.
              cmp     lr, #0
              beq     allDone

              // New value is quotient (X6:X5) from above:

              bl      u128toStrRec

              // When this function has processed all the
              // digits, write them to the buffer:

allDone:      orr     w4, w4, #'0'    // Convert to char.
              strb    w4, [x0], #1    // Bump pointer after store.
```

```
            // Restore state and return:

            ldp     x4, lr, [sp], #16    // Restore prev char.
            ret
            endp    u128toStrRec

// Here is the asmMain function.

            proc    asmMain, public

            locals  am
            dword   am.x2021
            byte    stk, 64
            endl    am

            enter   am.size              // Reserve space on stack.

            stp     x20, x21, [fp, #am.x2021]

            lea     x20, qdata
            mov     x21, #qcnt
loop:       mov     x1, x20
            lea     x0, buffer
            bl      u128toStr

            lea     x1, buffer
            mstr    x1, [sp]
            lea     x0, fmtStr1
            bl      printf

            add     x20, x20, #16       // Next value to convert
            subs    x21, x21, #1
            bne     loop

            ldp     x20, x21, [fp, #am.x2021]
            leave
            ret
            endp    asmMain
```

The code includes an optimized version of the 128-bit division function that divides a number by 10 ❶. This is followed by the nonrecursive entry point for `u128toStr`, which handles 0 as a special case and calls the recursive version for all other values ❷, and the recursive code for `u128toStr` ❸. As these functions are nearly identical to the recursive 64-bit string output functions, refer to that code (in Listing 9-6) for more details.

One issue with the `u128toStr` function is that it is much slower than the other numeric-to-string functions. This is all due to the performance of the `div10` subroutine. Because the 128-bit divide-by-10 algorithm is so slow, I won't bother improving the performance of the `u128toStr` conversion function. Unless you can come up with a very high-performance `div10` subroutine (perhaps using multiplication by a reciprocal; see section 9.6, "For More Information," on page 603), trying to optimize `u128toStr` is probably

a waste of time. Fortunately, this function likely won't be called often, so its performance won't matter much.

Here's the `build` command and sample output from Listing 9-9:

```
$ ./build Listing9-9
$ ./Listing9-9
Calling Listing9-9:
Value = 1
Value = 21
Value = 302
Value = 4003
Value = 50004
Value = 600005
Value = 7000006
Value = 80000007
Value = 900000008
Value = 1000000009
Value = 11000000010
Value = 120000000011
Value = 1300000000012
Value = 14000000000013
Value = 150000000000014
Value = 1600000000000015
Value = 17000000000000016
Value = 180000000000000017
Value = 1900000000000000018
Value = 20000000000000000019
Value = 21000000000000000020
Value = 22000000000000000021
Value = 23000000000000000022
Value = 24000000000000000023
Value = 25000000000000000000024
Value = 26000000000000000000025
Value = 27000000000000000000026
Value = 28000000000000000000027
Value = 29000000000000000000028
Value = 30000000000000000000029
Value = 31000000000000000000030
Value = 32000000000000000000031
Value = 33000000000000000000032
Value = 34000000000000000000033
Value = 35000000000000000000034
Value = 36000000000000000000035
Value = 37000000000000000000036
Value = 38000000000000000000037
Value = 30000000000000000000038
Value = 340282366920938463463374607431768211455
Listing9-9 terminated
```

I will leave it to you to create a 128-bit signed-integer conversion function, since the code is almost identical to `i64toStr` (see Listing 9-8); you just have to supply 128-bit negation and comparison operations. As a hint, for the comparison, just check the HO dword to see if the sign bit is set.

9.1.6 Formatted Conversions

The code in the previous sections converted signed and unsigned integers to strings by using the minimum number of necessary character positions. To create nicely formatted tables of values, you will need to write functions that provide appropriate padding in front of the string of digits before actually emitting the digits. Once you have the "unformatted" versions of these routines, implementing the formatted versions is easy.

The first step is to write `iSize` and `uSize` routines that compute the minimum number of character positions needed to display the value. One algorithm to accomplish this is similar to the numeric string conversion routines. The only difference is that you initialize a counter to 0 upon entry into the routine and increment this counter rather than outputting a digit on each recursive call. (Don't forget to increment the counter inside `iSize` if the number is negative; you must allow for the output of the minus sign.) After the calculation is complete, these routines should return the size of the operand in the X0 register.

However, thanks to its use of recursion and division, such a conversion scheme is slow. A brute-force conversion using a binary search is shown in Listing 9-10.

```
// Listing9-10.S
//
// u64Size function: Computes the size
// of an unsigned 64-bit integer (in
// print positions)

            #include    "aoaa.inc"

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-10"
fmtStr:     .asciz      "Value = %llu, size=%d\n"

// Values to test the u64Size function:

dVals:      .dword      1
            .dword      10
            .dword      100
            .dword      1000
            .dword      10000
            .dword      100000
            .dword      1000000
            .dword      10000000
            .dword      100000000
            .dword      1000000000
            .dword      10000000000
            .dword      100000000000
            .dword      1000000000000
            .dword      10000000000000
            .dword      100000000000000
            .dword      1000000000000000
            .dword      10000000000000000
```

```
            .dword      100000000000000000
            .dword      1000000000000000000
            .dword      10000000000000000000
dCnt        =           (.-dVals) / 8

            .code
            .extern     printf

// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// u64Size
//
//  Counts the number of output positions
//  required for an integer-to-decimal-
//  string conversion
//
//  Uses a binary search to quickly
//  count the digits required by a value
//
// Input:
//  X1- Unsigned integer to count
//
// Output:
//  X1- Digit count
//
// Table of digit counts and values:
//
//   1: 1
//   2: 10
//   3: 100
//   4: 1,000
//   5: 10,000
//   6: 100,000
//   7: 1,000,000
//   8: 10,000,000
//   9: 100,000,000
//  10: 1,000,000,000
//  11: 10,000,000,000
//  12: 100,000,000,000
//  13: 1,000,000,000,000
//  14: 10,000,000,000,000
//  15: 100,000,000,000,000
//  16: 1,000,000,000,000,000
//  17: 10,000,000,000,000,000
//  18: 100,000,000,000,000,000
//  19: 1,000,000,000,000,000,000
//  20: 10,000,000,000,000,000,000

          ❶ proc    u64Size
            stp     x0, x2, [sp, #-16]!
```

```
            ❷ mov     x2, x1
              ldr     x0, =1000000000 // 10: 1,000,000,000
              cmp     x2, x0
              bhs     ge10

              ldr     x0, =10000
              cmp     x2, x0
              bhs     ge5

              // Must be 1 to 4 digits here:

              mov     x1, #1
              cmp     x2, #1000
              cinc    x1, x1, hs
              cmp     x2, #100
              cinc    x1, x1, hs
              cmp     x2, #10
              cinc    x1, x1, hs
              ldp     x0, x2, [sp], #16
              ret

// Must be 5 to 9 digits here:

ge5:          ldr     x0, =1000000    // 7: 1,000,000
              cmp     x2, x0
              bhs     ge7

              // Must be 5 or 6 digits:

              mov     x1, #5
              ldr     x0, =100000     // 6: 100,000
              cmp     x2, x0
              cinc    x1, x1, hs
              ldp     x0, x2, [sp], #16
              ret

// Must be 7 to 9 digits here:

ge7:          mov     x1, #7
              ldr     x0, =10000000   // 8: 10,000,000
              cmp     x2, x0
              cinc    x1, x1, hs
              ldr     x0, =100000000  // 9: 100,000,000
              cmp     x2, x0
              cinc    x1, x1, hs
              ldp     x0, x2, [sp], #16
              ret

// Handle 10 or more digits here:

ge10:         ldr     x0, =100000000000000    // 15: 100,000,000,000,000
              cmp     x2, x0
              bhs     ge15
```

```
                // 10 to 14 digits here:

                ldr     x0, =1000000000000      // 13: 1,000,000,000,000
                cmp     x2, x0
                bhs     ge13

                // 10 to 12 digits here:

                mov     x1, #10
                ldr     x0, =10000000000        // 11: 10,000,000,000
                cmp     x2, x0
            ❸   cinc    x1, x1, hs
                ldr     x0, =100000000000       // 12: 100,000,000,000
                cmp     x2, x0
                cinc    x1, x1, hs
                ldp     x0, x2, [sp], #16
                ret

// 13 or 14 digits here:

ge13:           mov     x1, #13
                ldr     x0, =10000000000000     // 14: 10,000,000,000,000
                cmp     x2, x0
                cinc    x1, x1, hs
                ldp     x0, x2, [sp], #16
                ret

// 15 to 20 digits here:

ge15:           ldr     x0, =100000000000000000 // 18: 100,000,000,000,000,000
                cmp     x2, x0
                bhs     ge18

                // 15, 16, or 17 digits here:

                mov     x1, #15
                ldr     x0, =1000000000000000   // 16: 1,000,000,000,000,000
                cmp     x2, x0
                cinc    x1, x1, hs
                ldr     x0, =10000000000000000  // 17: 10,000,000,000,000,000
                cmp     x2, x0
                cinc    x1, x1, hs
                ldp     x0, x2, [sp], #16
                ret

// 18 to 20 digits here:

ge18:           mov     x1, #18
                ldr     x0, =1000000000000000000  // 19: 1,000,000,000,000,000,000
                cmp     x2, x0
                cinc    x1, x1, hs
                ldr     x0, =10000000000000000000 // 20 digits
                cmp     x2, x0
                cinc    x1, x1, hs
```

```
        ldp     x0, x2, [sp], #16
        ret
        endp    u64Size
```

The actual `u64Size` function ❶ uses a binary search algorithm to quickly scan through all the possible values to determine the digit count. It begins by dividing the search space in half, by comparing the input value (moved to X2) against a 10-digit value ❷. In the usual binary search fashion, the two sections of code will test for numbers with 1 to 9 digits and 10 to 20 digits. In each of those ranges, the search is (roughly) broken into halves again and again until the algorithm zeros in on the exact number of digits. When the code gets down to 2 to 4 digits, it uses some straight-line code and a series of `cinc` instructions to rapidly handle the last few cases without executing a branch ❸.

Here's the `build` command and sample output:

```
$ ./build Listing9-10
$ ./Listing9-10
Calling Listing9-10:
Value = 1, size=1
Value = 10, size=2
Value = 100, size=3
Value = 1000, size=4
Value = 10000, size=5
Value = 100000, size=6
Value = 1000000, size=7
Value = 10000000, size=8
Value = 100000000, size=9
Value = 1000000000, size=10
Value = 10000000000, size=11
Value = 100000000000, size=12
Value = 1000000000000, size=13
Value = 10000000000000, size=14
Value = 100000000000000, size=15
Value = 1000000000000000, size=16
Value = 10000000000000000, size=17
Value = 100000000000000000, size=18
Value = 1000000000000000000, size=19
Value = 10000000000000000000, size=20
Listing9-10 terminated
```

For signed integers, add the function in Listing 9-11 to the code in Listing 9-10 (find a full Listing 9-11 in the book's downloadable code files at *https://artofarm.randallhyde.com*).

```
// Listing9-11.S
//
// i64Size:
//
// Computes the number of character positions that
// the i64toStr function will emit
```

```
            proc    i64Size
            str     lr, [sp, #-16]!

            cmp     x1, #0          // If less than zero,
            bge     isPositive      // negate and treat
                                    // like an uns64.
            neg     x1, x1

            bl      u64Size
            add     x1, x1, #1      // Adjust for "-".
            ldr     lr, [sp], #16
            ret

isPositive: bl      u64Size
            ldr     lr, [sp], #16
            ret
            endp    i64Size
```

For extended-precision size operations, the binary search approach quickly becomes unwieldy (64 bits is bad enough). The best solution is to divide your extended-precision value by a power of 10 (say, 1e+16). This will reduce the size of the number by 16 digits. Repeat this process as long as the quotient is greater than 64 bits, keeping track of the number of times you've divided the number by 1e+16. When the quotient fits into 64 bits (19 or 20 digits), call the 64-bit `u64Size` function and add in the number of digits you eliminated with the division operation (16 for each division by 1e+16). I'll leave this implementation to you.

Once you have the `i64Size` and `u64Size` routines, writing the formatted output routines `u64toStrSize` or `i64toStrSize` is easy. On initial entry, these routines call the corresponding `i64Size`/`u64Size` routine to determine the number of character positions for the number. If the value that the `i64Size`/`u64Size` routine returns is greater than or equal to the value of the minimum size parameter (passed into `u64toStrSize` or `i64toStrSize`), no other formatting is necessary. If the value of the parameter size is greater than the value `i64Size`/`u64Size` returns, the program must compute the difference between these two values and emit that many spaces (or other filler characters) to the output string before the numeric conversion (assuming right-justification of the value, which is what this chapter presents).

Listing 9-12 shows the `utoStrSize`/`itoStrSize` functions (full source code appears online); here, I omit everything but the `utoStrSize`/`itoStrSize` functions themselves.

```
// Listing9-12.S (partial)
//
// u64ToSizeStr
//
//  Converts an unsigned 64-bit integer to
//  a character string, using a minimum field
//  width
//
```

```
//  Inputs:
//      X0- Pointer to buffer to receive string
//
//      X1- Unsigned 64-bit integer to convert
//          to a string
//
//      X2- Minimum field width for the string
//          (maximum value is 1,024). Note: if
//          the minimum field width value is less
//          than the actual output size of the
//          integer, this function will ignore
//          the value in X2 and use the correct
//          number of output positions for the
//          value.
//
//  Outputs:
//
//      Buffer- Receives converted characters.
//              Buffer must be at least 22 bytes
//              or X1 + 1 bytes long.

          ❶ proc    u64ToStrSize
            stp     x0, lr, [sp, #-16]!
            stp     x1, x2, [sp, #-16]!
            stp     x23, x24, [sp, #-16]!
            stp     x25, x26, [sp, #-16]!

            // Initialize x25 and x26 with
            // appropriate functions to call:

            lea     x25, u64Size
            lea     x26, u64ToStr

            b.al    toSizeStr
            endp    u64ToStrSize

////////////////////////////////////////////////////
//
// i64ToStrSize:
//
//  Just like u64ToStrSize, but handles signed integers
//
//  Inputs:
//      X0- Pointer to buffer to receive string
//
//      X1- Signed 64-bit integer to convert
//          to a string
//
//      X2- Minimum field width for the string
//          (maximum value is 1,024). Note: if
//          the minimum field width value is less
//          than the actual output size of the
//          integer, this function will ignore
//          the value in X2 and use the correct
```

```
//          number of output positions for the
//          value.
//
//      Note:   Don't forget that if the number
//              is negative, the '-' consumes
//              an output position.
//
//  Outputs:
//      Buffer- Receives converted character.
//              Buffer must be at least 22 bytes
//              or X2 + 1 bytes long.

          ❷ proc    i64ToStrSize
            stp     x0, lr, [sp, #-16]!
            stp     x1, x2, [sp, #-16]!
            stp     x23, x24, [sp, #-16]!
            stp     x25, x26, [sp, #-16]!

            // Initialize x25 and x26 with
            // appropriate functions to call:

            lea     x25, i64Size
            lea     x26, i64ToStr

            b.al    toSizeStr   // Technically, this could just fall through.
            endp    i64ToStrSize

////////////////////////////////////////////////////
//
// toSizeStr:
//
//  Special function to handle signed and
//  unsigned conversions for u64ToSize and i64ToSize

          ❸ proc    toSizeStr

            mov     x24, x1 // Save for now.
          ❹ blr     x25     // Compute size of number.

            // Compute difference between actual size
            // and desired size. Set to the larger of
            // the two:

          ❺ cmp     x2, x1
            csel    x23, x2, x1, ge

            // Just as a precaution, limit the
            // size to 1,024 characters (including
            // the zero-terminating byte):

            mov     x2, #1023   // Don't count 0 byte here.
            cmp     x23, x2
            csel    x23, x23, x2, ls
```

```
            // Compute the number of spaces to emit before
            // the first digit of the number:

            subs    x23, x23, x1
            beq     spacesDone

            // Emit that many spaces to the buffer:

          ❻ mov     x1, #0x2020
            movk    x1, #0x2020, lsl #16
            movk    x1, #0x2020, lsl #32
            movk    x1, #0x2020, lsl #48
            b.al    tst8

            // Handle sequences of eight spaces:

whl8:       str     x1, [x0], #8
            sub     x23, x23, #8
tst8:       cmp     x23, #8
            bge     whl8

            // If four to seven spaces, emit four
            // spaces here:

            cmp     x23, #4
            blt     try2
            str     w1, [x0], #4
            sub     x23, x23, #4

            // If two or three spaces, emit two
            // here:

try2:       cmp     x23, #2
            blt     try1
            strh    w1, [x0], #2
            sub     x23, x23, #2

            // If one space left, emit it here:

try1:       cmp     x23, #1
            blt     spacesDone
            strb    w1, [x0], #1

            // Okay, emit the digits here:

spacesDone: mov     x1, x24 // Retrieve value.
          ❼ blr     x26     // XXXToStr

            ldp     x25, x26, [sp], #16
            ldp     x23, x24, [sp], #16
            ldp     x1, x2,   [sp], #16
            ldp     x0, lr,   [sp], #16
            ret
            endp    toSizeStr
```

```
////////////////////////////////////////////////////
//
// printSize
//
// Utility used by the main program to
// compute sizes and print them

          ❽ proc    printSize

            locals  ps
            dword   stk, 64
            endl    ps

            enter   ps.size

            mov     x6, x1
            lea     x0, buffer
            blr     x27         // Call XXXToStrSize.

            mov     x1, x6
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            lea     x3, buffer
            mstr    x3, [sp, #16]
            lea     x0, fmtStr
            bl      printf

            leave
            endp    printSize


values:     .dword  1, 10, 100, 1000, 10000, 100000, 1000000
            .dword  10000000, 100000000, 1000000000, 10000000000
            .dword  100000000000, 1000000000000, 10000000000000
            .dword  100000000000000, 1000000000000000
            .dword  10000000000000000, 100000000000000000
            .dword  1000000000000000000, 10000000000000000000
            .dword  0x7fffffffffffffff
            .set    valSize, (.-values)/8

negValues:  .dword  -1, -10, -100, -1000, -10000, -100000, -1000000
            .dword  -10000000, -100000000, -1000000000, -10000000000
            .dword  -100000000000, -1000000000000, -10000000000000
            .dword  -100000000000000, -1000000000000000
            .dword  -10000000000000000, -100000000000000000
            .dword  -1000000000000000000, -10000000000000000000
            .dword  0x8000000000000000

sizes:      .word   5, 6, 7, 8, 9, 10, 15, 15, 15, 15
            .word   20, 20, 20, 20, 20, 25, 25, 25, 25, 25, 30

////////////////////////////////////////////////////
//
// Here is the asmMain function:
```

```
          ❾ proc    asmMain, public

            locals  am
            qword   am.x26x27
            qword   am.x24x25
            byte    am.stk, 64
            endl    am

            enter   am.size     // Activation record
            stp     x26, x27, [fp, #am.x26x27]
            stp     x24, x25, [fp, #am.x24x25]

// Test unsigned integers:

            lea     x27, u64ToStrSize
            lea     x24, values
            lea     x25, sizes
            mov     x26, #valSize
tstLp:      ldr     x1, [x24], #8
            ldr     w2, [x25], #4
            bl      printSize
            subs    x26, x26, #1
            bne     tstLp

            lea     x27, i64ToStrSize
            lea     x24, negValues
            lea     x25, sizes
            mov     x26, #valSize
ntstLp:     ldr     x1, [x24], #8
            ldr     w2, [x25], #4
            bl      printSize
            subs    x26, x26, #1
            bne     ntstLp

            ldp     x26, x27, [fp, #am.x26x27]
            ldp     x24, x25, [fp, #am.x24x25]
            leave
            endp    asmMain
```

The `u64toStrSize` function ❶ simply loads up X25 and X26 with appropriate addresses and branches to the generic `toSizeStr` function to handle the real work. The `i64ToStrSize` function ❷ does the same thing for signed integer conversions.

The `toSizeStr` function ❸ handles the real work. First, it calls the appropriate `toSize` function (whose address was passed in X25) to compute the minimum number of print positions the value will require ❹. It then computes the number of fill characters required in front of the digits to right-justify the number in the output field ❺. It emits the required number of filler characters ❻ before outputting the numeric string ❼. Probably the only thing worth noting here is that the code attempts to output eight spaces at a time in order to improve performance, as long as there are at least eight padding characters, then four, then two, and finally one.

The `printSize` procedure ❽ is a little utility function that the `asmMain` procedure uses to display values, and the `asmMain` procedure ❾ tests the `u64ToStrSize` and `i64ToStrSize` procedures.

Here's the `build` command and sample output for Listing 9-12 (remember that the actual main program appears only in the online source code):

```
$ ./build Listing9-12
$ ./Listing9-12
Calling Listing9-12:
                    1:   5='    1'
                   10:   6='    10'
                  100:   7='    100'
                 1000:   8='    1000'
                10000:   9='    10000'
               100000:  10='    100000'
              1000000:  15='       1000000'
             10000000:  15='      10000000'
            100000000:  15='     100000000'
           1000000000:  15='    1000000000'
          10000000000:  20='         10000000000'
         100000000000:  20='        100000000000'
        1000000000000:  20='       1000000000000'
       10000000000000:  20='      10000000000000'
      100000000000000:  20='     100000000000000'
     1000000000000000:  25='         1000000000000000'
    10000000000000000:  25='        10000000000000000'
   100000000000000000:  25='       100000000000000000'
  1000000000000000000:  25='      1000000000000000000'
-8446744073709551616:  25='     -8446744073709551616'
 9223372036854775807:  30='           9223372036854775807'
                   -1:   5='   -1'
                  -10:   6='   -10'
                 -100:   7='   -100'
                -1000:   8='   -1000'
               -10000:   9='   -10000'
              -100000:  10='   -100000'
             -1000000:  15='      -1000000'
            -10000000:  15='     -10000000'
           -100000000:  15='    -100000000'
          -1000000000:  15='   -1000000000'
         -10000000000:  20='        -10000000000'
        -100000000000:  20='       -100000000000'
       -1000000000000:  20='      -1000000000000'
      -10000000000000:  20='     -10000000000000'
     -100000000000000:  20='    -100000000000000'
    -1000000000000000:  25='        -1000000000000000'
   -10000000000000000:  25='       -10000000000000000'
  -100000000000000000:  25='      -100000000000000000'
 -1000000000000000000:  25='     -1000000000000000000'
 8446744073709551616:  25='      8446744073709551616'
-9223372036854775808:  30='          -9223372036854775808'
Listing9-12 terminated
```

The output is *value*:*size*='*conversion*'.

9.2 Converting Floating-Point Values to Strings

Thus far, this chapter has dealt with converting integer numeric values to character strings (typically for output to the user). This section discusses converting floating-point values to a string, which is just as important.

Converting floating-point values to strings can take one of two forms:

- Decimal notation conversion (such as ±*xxx*.*yyy* format)
- Exponential (or scientific) notation conversion (such as ±*x*.*yyyyy*e±*zz* format)

Regardless of the final output format, you'll need two distinct operations to convert a value in floating-point form to a character string. First, you must convert the mantissa to an appropriate string of digits. Second, you convert the exponent to a string of digits.

However, this isn't a simple case of converting two integer values to a decimal string and concatenating them (with an *e* between the mantissa and exponent). First of all, the mantissa is not an integer value; it is a fixed-point fractional binary value. Simply treating it as an *n*-bit binary value (where *n* is the number of mantissa bits) will almost always result in an incorrect conversion. Second, while the exponent is, more or less, an integer value, it represents a power of 2, not a power of 10. Displaying that power of 2 as an integer value is not appropriate for decimal floating-point representation. These two issues (fractional mantissa and binary exponent) are the source of the major complications associated with converting a floating-point value to a string.

NOTE *The exponent is actually a biased-exponent value. However, that's easy to convert to a signed binary integer.*

Double-precision floating-point values have a 53-bit mantissa (including the implied bit). This is not a 53-bit integer. Instead, those 53 bits represent a value from 1.0 to slightly less than 2.0. (See section 2.13, "IEEE Floating-Point Formats," on page 93 for more details on the IEEE 64-bit floating-point format.) The double-precision format can represent numbers from 0 to about 5×10^{-324} (around $\pm 1 \times 10^{\pm 308}$ using normalized values).

To output the mantissa in decimal form with approximately 16 digits of precision, successively multiply or divide the floating-point value by 10 until the number is from 1e+15 to just less than 1e+16 (that is, 9.9999 . . . e+15). Once the exponent is in the appropriate range, the mantissa bits form a 16-digit integer value (no fractional part), which can be converted to a decimal string to obtain the 16 digits that make up the mantissa value.

To convert the exponent to an appropriate decimal string, track the number of multiplications or divisions by 10. For each division by 10, add 1 to the decimal exponent value; for each multiplication by 10, subtract 1 from the decimal exponent value. At the end of the process, subtract 16 from the decimal exponent value (as this process produces a value whose exponent is 16) and convert the decimal exponent value to a string.

The conversions in the following sections assume that you always want to produce a mantissa with 16 significant digits. To produce formatted output with fewer significant digits, see section 9.2.4, "Double-Precision Values to Strings," on the next page.

9.2.1 Floating-Point Exponent to String of Decimal Digits

To convert the exponent to a string of decimal digits, use the following algorithm:

1. If the number is 0.0, directly produce the mantissa output string of "0000000000000000" (notice the space at the beginning of the string), set the exponent to 0, and you're done. Otherwise, continue with the following steps.
2. Initialize the decimal exponent to 0.
3. If the exponent is negative, emit a hyphen (-) character and negate the value; if it is positive, emit a space character.
4. If the value of the (possibly negated) exponent is less than 1.0, skip to step 8.
5. Positive exponents: Compare the number against successively smaller powers of 10, starting with 10^{+256}, then 10^{+128}, then 10^{+64}, then . . . , then 10^{0}. After each comparison, if the current value is greater than the power of 10, divide by that power of 10 and add the power-of-10 exponent (256, 128, . . . , 0) to the decimal exponent value.
6. Repeat step 5 until the exponent is 0 (that is, the value is in the range $1.0 \le value < 10.0$).
7. Skip to step 10.
8. Negative exponents: Compare the number against successful larger powers of 10 starting with 10^{-256}, then, 10^{-128}, then 10^{-64}, then . . . , then 10^{0}. After each comparison, if the current value is less than the power of 10, divide by that power of 10 and subtract the power-of-10 exponent (256, 128, . . . , 0) from the decimal exponent value.
9. Repeat step 8 until the exponent is 0 (that is, the value is in the range $1.0 \le value < 10.0$).
10. At this point, the exponent value is a reasonable number that can be converted to an integer value by using standard unsigned-to-string conversions (see section 9.1.3, "Unsigned Decimal Values to Strings," on page 495).

9.2.2 Floating-Point Mantissa to String of Digits

To convert the mantissa to a string of digits, you can't simply treat the 53-bit mantissa produced in the previous section as an integer value, since it still represents an integer from 1.0 to just less than 2.0. However, if you multiply that floating-point value (which has been converted to a value from 1.0 to slightly less than 10.0) by 10^{+15}, this effectively produces an integer with the digits shifted to the left 15 print positions (16 digits being the number of

output digits possible with a double-precision value). You can then convert this "integer" to a string. The result will consist of the 16 mantissa digits. To convert the mantissa to a string, do the following:

1. Multiply the value produced by the exponent calculation in the previous section by 1e+15. This produces a number with the decimal digits shifted to the left by 15 print positions.
2. Grab the 52-bit mantissa and OR in an implicit bit 52 equal to 1, and zero-extend this 53-bit value to 64 bits.
3. Convert the resulting 64-bit value to a string by using the unsigned integer-to-string function given earlier in this chapter (see section 9.1.3, "Unsigned Decimal Values to Strings," on page 495).

9.2.3 Strings in Decimal and Exponential Format

To produce a decimal string (rather than a number in exponential form), the remaining task is to properly place the decimal point into the string of digits. If the exponent is greater than or equal to 0, you need to insert the decimal point in position *exponent* + 1, starting from the first mantissa digit produced in the previous section. For example, if the mantissa conversion produced `1234567890123456` and the exponent is 3, then you would insert a decimal point before the character at index 4 (3 + 1), yielding `1234.567890123456` as the result.

If the exponent is greater than 16, insert *exponent* – 16 zero characters at the end of the string (or return an error if you don't want to allow conversions of values larger than 1e+16 to decimal form). If the exponent is less than 0, insert `0.` followed by *abs*(*exp*) – 1 zero characters in front of the string of digits. If the exponent is less than –16 (or another arbitrary value), you might elect to return an error or automatically switch to exponential form.

Producing exponential output is slightly easier than decimal output. Always insert a decimal point between the first and second characters in the converted mantissa string and then follow the string with `e`*±xxx*, where *±xxx* is the exponent value's string conversion. For example, if the mantissa conversion produces `1234567890123456` and the exponent is –3, the resulting string will be `1.234567890123456e-003` (note the leading 0s on the exponent digits).

9.2.4 Double-Precision Values to Strings

This section presents the code that will convert a double-precision value to a string in either decimal or exponential form, with separate functions for the two output formats. As Listing 9-13 is rather long, I've broken it into pieces and annotated each section.

```
// Listing9-13.S
//
// Floating-point (double) to string conversion
//
```

```
// Provides both exponential (scientific notation)
// and decimal output formats
            #include    "aoaa.inc"

          ❶ .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-13"
fmtStr1:    .asciz      "r64ToStr: value='%s'\n"
fmtStr2:    .asciz      "fpError: code=%lld\n"
fmtStr3:    .asciz      "e64ToStr: value='%s'\n"
newlines:   .asciz      "\n\n"
expStr:     .asciz      "\n\nTesting e64ToStr:\n\n"

// r10str_1: A global character array that will
// hold the converted string

          ❷ .data
r64str_1:   .space      32, 0

            .code
            .extern     printf

// tenTo15: Used to multiply a value from 1.0
// to less than 2.0 in order to convert the mantissa
// to an actual integer

❸ tenTo15:    .double     1.0e+15

// potPos, potNeg, and expTbl:
//
// Power of 10s tables (pot) used to quickly
// multiply or divide a floating-point value
// by powers of 10. expTbl is the power-of-
// 10 exponent (absolute value) for each of
// the entries in these tables.

❹ potPos:     .double     1.0e+0
              .double     1.0e+1
              .double     1.0e+2
              .double     1.0e+4
              .double     1.0e+8
              .double     1.0e+16
              .double     1.0e+32
              .double     1.0e+64
              .double     1.0e+128
              .double     1.0e+256
expCnt        =           (.-potPos) / 8

potNeg:       .double     1.0e-0
              .double     1.0e-1
              .double     1.0e-2
              .double     1.0e-4
              .double     1.0e-8
              .double     1.0e-16
              .double     1.0e-32
```

```
                .double     1.0e-64
                .double     1.0e-128
                .double     1.0e-256

expTbl:         .dword      0
                .dword      1
                .dword      2
                .dword      4
                .dword      8
                .dword      16
                .dword      32
                .dword      64
                .dword      128
                .dword      256

// Maximum number of significant digits for
// a double-precision value:

❺ maxDigits     =           16

// Return program title to C++ program:

              ❻ proc        getTitle, public
                lea         x0, ttlStr
                ret
                endp        getTitle
```

As is typical for sample programs in this chapter, Listing 9-13 begins with a read-only data section ❶ containing the program's title string and various format strings used by `printf()` calls in the main program. The single data variable in this program is `r64str_1` ❷, a 32-byte character string used to hold the converted string. The program is responsible for ensuring that all conversions will fit into 32 bytes.

Listing 9-13 places several read-only constants in the `.code` section so the program can directly access these constants by using the PC-relative addressing mode (rather than using multiple instructions to take the address of the object and access it indirectly). The first such constant is `tenTo15` ❸, which holds the value 1.0e+15. The conversion code uses this constant to multiply a floating-point value in the range 1.0 to slightly less than 10.0 by 1e+15, thereby obtaining a value slightly less than 1e+16 when converting the mantissa to an integer value.

The `potPos`, `potNeg`, and `expTbl` tables ❹ contain the positive and negative powers of 10 (*pot*) tables used to multiply the floating-point value by various powers of 10 when massaging the value into the range 1.0 to 10.0. The `expTbl` contains the absolute value of the exponent corresponding to the same entry in the `potPos` and `potNeg` tables. The code adds or subtracts this value from the accumulated decimal exponent while converting the mantissa to the range 1.0 to 10.0.

The `maxDigits` manifest constant ❺ specifies the number of significant digits supported by this conversion code (16 digits for double-precision

floating-point numbers). Finally, this code section contains the ubiquitous `getTitle` function ❻ that returns the address of the program's title string to the C++ shell code.

The following code converts a floating-point value to a string:

```
// Listing9-13.S (cont.)
//
// u53toStr
//
//  Converts a 53-bit unsigned integer to a string containing
//  exactly 16 digits (technically, it does 64-bit arithmetic,
//  but is limited to 53 bits because of the 16-digit output
//  format)
//
// Inputs:
//  X0-     Pointer to buffer to receive string
//  X1-     Unsigned 53-bit integer to convert
//
// Outputs:
//  Buffer- Receives the zero-terminated string
//  X0-     Points at zero-terminating byte in string
//
//  Buffer must have at least 17 bytes allocated for it.
//
// This code is a bit simplified from the u64toStr function
// because it always emits exactly 16 digits
// (never any leading 0s).

        ❶ proc    u53toStr

          stp     x1, x2, [sp, #-16]!
          stp     x3, x4, [sp, #-16]!
          str     x5, [sp, #-16]!

          mov     x4, #10     // Mul/div by 10 using X4
          mov     x5, xzr     // Holds string of 8 chars

          // Handle LO digit here. Note that the LO
          // digit will ultimately be moved into
          // bit positions 56-63 of X5 because numeric
          // strings are, intrinsically, big-endian (with
          // the HO digit appearing first in memory).

        ❷ udiv    x2, x1, x4      // X2 = quotient
          msub    x3, x2, x4, x1  // X3 = remainder
          orr     x3, x3, #'0'
          orr     x5, x3, x5, lsl #8

          // The following is an unrolled loop
          // (for speed) that processes the
          // remaining 15 digits.
          //
```

```
// Handle digit 1 here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 2 here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 3 here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 4 here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 5 here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 6 here:

udiv    x2, x1, x4      // X2 = quotient
msub    x3, x2, x4, x1  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Handle digit 7 here:

udiv    x1, x2, x4      // X1 = quotient
msub    x3, x1, x4, x2  // X3 = remainder
orr     x3, x3, #'0'
orr     x5, x3, x5, lsl #8

// Store away LO 8 digits:

str     x5, [x0, #8]
mov     x5, xzr
```

```
    // Handle digit 8 here:

❸   udiv    x2, x1, x4      // X2 = quotient
    msub    x3, x2, x4, x1  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 9 here:

    udiv    x1, x2, x4      // X1 = quotient
    msub    x3, x1, x4, x2  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 10 here:

    udiv    x2, x1, x4      // X2 = quotient
    msub    x3, x2, x4, x1  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 11 here:

    udiv    x1, x2, x4      // X1 = quotient
    msub    x3, x1, x4, x2  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 12 here:

    udiv    x2, x1, x4      // X2 = quotient
    msub    x3, x2, x4, x1  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 13 here:

    udiv    x1, x2, x4      // X1 = quotient
    msub    x3, x1, x4, x2  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 14 here:

    udiv    x2, x1, x4      // X2 = quotient
    msub    x3, x2, x4, x1  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8

    // Handle digit 15 here:

    udiv    x1, x2, x4      // X1 = quotient
    msub    x3, x1, x4, x2  // X3 = remainder
    orr     x3, x3, #'0'
    orr     x5, x3, x5, lsl #8
```

```
            // Store away HO 8 digits:

            str     x5, [x0]
            strb    wzr, [x0, #maxDigits]!  // Zero-terminating byte

            ldr     x5, [sp], #16
            ldp     x3, x4, [sp], #16
            ldp     x1, x2, [sp], #16
            ret
            endp    u53toStr
```

The `u53ToStr` function ❶ is responsible for converting a 53-bit unsigned integer to a string of exactly 16 digits. In theory, this code could have used the `u64toSizeStr` function from Listing 9-12 to convert the 53-bit value (zero-extended to 64 bits) into a string. However, the conversion of floating-point mantissa to string always produces a 16-character string (with leading 0s, if necessary), so the decimal integer-to-string conversion can be more efficient than the `u64toSizeStr` function, which could produce variable-length strings. To prioritize saving space, if you're already using the `u64toSizeStr` function in your code, you could remove `u53ToStr` and substitute a call to `u64toSizeStr` (specifying `'0'` as the fill character).

The conversion algorithm `u53ToStr` uses is straightforward and brute-force: it converts the LO eight digits to a sequence of eight characters and emits them ❷, then converts the HO eight digits to a sequence of eight characters and emits them ❸. It both cases, it uses the divide-by-10 and remainder of division-by-10 algorithms to convert each digit to a character (see the discussion of `u64ToStr` in Listing 9-6 for more details).

This function is used by `FPDigits` to convert the mantissa to a string of decimal digits:

```
// Listing9-13.S (cont.)
//
// FPDigits
//
//  Used to convert a floating-point value
//  in D0 to a string of digits
//
// Inputs:
//  D0-     Double-precision value to convert
//  X0-     Pointer to buffer to receive chars
//
// Outputs:
//  X0-     Still points at buffer
//  X1-     Contains exponent of the number
//  X2-     Contains sign (space or '-')

            proc    FPDigits
            str     lr,       [sp, #-16]!
            str     d0,       [sp, #-16]!
            stp     d1, d2,   [sp, #-16]!
            stp     x22, x23, [sp, #-16]!
```

```
                stp     x24, x25, [sp, #-16]!
                stp     x26, x27, [sp, #-16]!

                mov     x2, #' '        // Assume sign is +.

#define fp1 d2                          // D2 holds 1.0.

                fmov    fp1, #1.0

                 // Special case for 0.0:

             ❶  fcmp    d0, #0.0
                bne     d0not0

                // Check for -0.0:

             ❷  fmov    x1, d0
                ands    x1, x1, #0x8000000000000000
                beq     posZero
                mov     x2, #'-'

posZero:
                mov     x1, #0x3030
                movk    x1, #0x3030, lsl #16
                movk    x1, #0x3030, lsl #32
                movk    x1, #0x3030, lsl #48
                str     x1, [x0]
                str     x1, [x0, #8]
                mov     x1, #0          // Exponent = 0

                // For debugging purposes, zero-terminate this
                // string (the actual code just grabs 16 bytes,
                // so this isn't strictly necessary):

                strb    w0, [x0, #16]
                b.al    fpdDone

// If the number is nonzero, deal with it here. Note
// that the flags were set by comparing D0 to 0.0 earlier.

❸ d0not0:       bge     fpIsPositive    // See if positive or negative.

                // If negative, negate and change the sign
                // character to '-'.

                fabs    d0, d0
                mov     x2, #'-'

// Get the number from 1.0 to <10.0 so you can figure out
// what the exponent is. Begin by checking to see if you have
// a positive or negative exponent.

fpIsPositive:
                mov     x1, xzr         // Initialize exponent.
```

```
          ❹ fcmp    d0, fp1
            bge     posExp

            // The value is in the range 0.0 to 1.0,
            // exclusive, at this point. That means this
            // number has a negative exponent. Multiply
            // the number by an appropriate power of 10
            // until you get it in the range 1 through 10.

            lea     x27, potNeg
            lea     x26, potPos
            lea     x25, expTbl
            mov     x24, #expCnt

// Search through the potNeg table until you find a power
// of 10 that is less than the value in D0:

cmpNegExp:
          ❺ subs    x24, x24, #1
            blt     test1       // Branch if X24 < 1.

            ldr     d1, [x27, x24, lsl #3]  // D1 = potNeg[X24 * 8]
            fcmp    d1, d0      // Repeat while
            ble     cmpNegExp   // table <= value.

            // Eliminate the current exponent indexed by
            // X24 by multiplying by the corresponding
            // entry in potPos:

            ldr     x22, [x25, x24, lsl #3] // X22 = expTbl[X24 * 8]
            sub     x1, x1, x22
            ldr     d1, [x26, x24, lsl #3]  // D1 = potPos[X24 * 8]
            fmul    d0, d0, d1
            b.al    cmpNegExp

// If you get to this point, you've indexed through
// all the elements in the potNeg and it's time to stop.
//
// If the remainder is *exactly* 1.0, you can branch
// on to InRange1_10; otherwise, you still have to multiply
// by 10.0 because you've overshot the mark a bit.

test1:      fcmp    d0, fp1
            beq     inRange1_10

            fmov    d1, #10.0
            fmul    d0, d0, d1
            sub     x1, x1, #1      // Decrement exponent.
            b.al    inRange1_10

// At this point, you have a number that is 1 or greater.
// Once again, your task is to get the value from 1.0 to <10.0.
```

```
posExp:
            lea     x26, potPos
            lea     x25, expTbl
            mov     x24, #expCnt

❻ cmpPosExp: subs   x24, x24, #1
            blt     inRange1_10     // If X24 < 1

            ldr     d1, [x26, x24, lsl #3]  // D1 = potPos[X24 * 8]
            fcmp    d1, d0
            bgt     cmpPosExp

            ldr     x22, [x25, x24, lsl #3] // X22 = expTbl[X24 * 8]
            add     x1, x1, x22
            fdiv    d0, d0, d1
            b.al    cmpPosExp

// Okay, at this point the number is in the range 1 <= x < 10.
// Let's multiply it by 1e+15 to put the most significant digit
// into the 16th print position, then convert the result to
// a string and store away in memory.

❼ inRange1_10:
            ldr     d1, tenTo15
            fmul    d0, d0, d1
            fcvtau  x22, d0     // Convert to unsigned integer.

            // Convert the integer mantissa to a
            // string of digits:

            stp     x0, x1, [sp, #-16]!
            mov     x1, x22
            bl      u53toStr
            ldp     x0, x1, [sp], #16

fpdDone:
            ldp     x26, x27,   [sp], #16
            ldp     x24, x25,   [sp], #16
            ldp     x22, x23,   [sp], #16
            ldp     d1, d2,     [sp], #16
            ldr     d0,         [sp], #16
            ldr     lr,         [sp], #16
            ret
            endp    FPDigits
```

`FPDigits` converts an arbitrary double-precision mantissa to a string of decimal digits. It assumes that the floating-point value to convert is held in the D0 register and that X0 contains a pointer to the buffer that will hold the string conversion. This function also converts the binary (power-of-2) exponent to a decimal integer, returns the exponent value in the X1 register, and returns the value's sign (a space character, indicating a nonnegative value, or -) in the X2 register.

FPDigits begins by first checking for the special case of 0.0 ❶. If D0 contains 0, this function initializes the string buffer to 0000000000000000 (sixteen 0 characters) and returns with X0 containing 0 and X2 containing a space character. The code checks for the special case of -0.0 and returns X2 containing a minus sign if the result is -0.0 ❷. Next, FPDigits checks the sign of the floating-point value and sets X2 to a '-', if appropriate ❸. The code also initializes the decimal exponent accumulator (held in X0) to 0.

After setting the sign, the FPDigits function checks the floating-point value's exponent to see if it is positive or negative ❹. The code handles values with positive or negative exponents independently. If the exponent is negative, the cmpNegExp loop searches through the potNeg table looking for the value that is greater than the value in D0 ❺. When the loop finds such a value, it multiplies D0 by that entry in potNeg and then subtracts the corresponding entry in expTbl from the decimal exponent value held in X1. The cmpNegExp loop repeats this process until the value in D0 is greater than 1.0. Whenever the result isn't greater than 1.0, the code multiplies the value in D0 by 10.0, because the code needs to adjust for the multiplication by 0.1 that has taken place. If, on the other hand, the exponent was positive ❻, the cmpPosExp loop does the same task but divides by entries in the potPos table and adds the corresponding entry in expTbl to the decimal exponent value held in X1.

Once the cmpPosExp or cmpNegExp loop gets the value into the range 1.0 to just less than 10.0, it multiplies the value by 10^{15} and converts it to an integer (in X22) ❼. Then FPDigits calls the u53toStr function to convert this integer to a string of exactly 16 digits. The function returns the sign character (space for nonnegative values, '-' for negative values) in X2 and the decimal exponent in X1.

Note that FPDigits converts only the mantissa to a string of digits. This is the base code used by the r64ToStr and e64ToStr functions that convert floating-point values into recognizable strings. Before presenting those functions, there is one utility function to explain: chkNaNINF.

Certain floating-point operations produce invalid results. The IEEE 754 floating-point standard defines three special values to represent these invalid results: NaN (not a number), +INF (infinity), and -INF (negative infinity). Because the ARM floating-point hardware can produce these results, it is important that the conversions of floating-point to string handle these three special values. NaN, +INF, and -INF all have an exponent value containing 0x7FF (and no other valid values use this exponent). If the exponent is 0x7FF and the mantissa bits are all 0s, the value is +INF or -INF (determined by the sign bit). If the mantissa is nonzero, the value is NaN (and you can ignore the sign bit). The chkNaNINF function checks for these values and outputs the strings NaN, INF, or -INF if the number is invalid:

```
// Listing9-13.S (cont.)
//
// chkNaNINF
//
// Utility function used by r64ToStr and e64ToStr to check
```

```
// for NaN and INF
//
// Inputs:
//  D0-     Number to check against NaN and INF
//  X19-    Field width for output
//  X21-    Fill character
//  X22-    (outBuf) Pointer to output buffer
//  X25-    Return address to use if number is invalid
//
// Outputs:
//  Buffer- Will be set to the string NaN, INF,
//          or -INF if the number is not valid
//
//  Note: Modifies value in X0

            proc    chkNaNINF

            // Handle NaN and INF special cases:

          ❶ fmov    x0, d0
            lsr     x0, x0, #52
            and     x0, x0, #0x7ff
            cmp     x0, #0x7ff
            blo     notINFNaN

            // At this point, it's NaN or INF. INF has a
            // mantissa containing 0, NaN has a nonzero
            // mantissa:

          ❷ fmov    x0, d0
            ands    x0, x0, #0x000fffffffffffff
            beq     isINF

            // Is NaN here:

          ❸ ldr     w0, ='N' + ('a' << 8) + ('N' << 16)
            str     w0, [x22]
            mov     x0, #3
            b.al    fillSpecial

            // INF can be positive or negative. Must output a
            // '-' character if it is -INF:

❹ isINF:    fmov    x0, d0
            ands    x0, x0, #0x8000000000000000 // See if -INF.
            bne     minusINF

            ldr     w0, ='I' + ('N' << 8) + ('F' << 16)
            str     w0, [x22]
            mov     x0, #3
            b.al    fillSpecial

❺ minusINF: ldr     w0, ='-' + ('I' << 8) + ('N' << 16) + ('F' << 24)
            str     w0, [x22]
```

```
            strb    wzr, [x22, #4]
            mov     x0, #4

// For NaN and INF, fill the remainder of the string, as appropriate:

❻ fillSpecial:
            b.al    whlLTwidth

fsLoop:     strb    w21, [x22, x0]
            add     x0, x0, #1
 whlLTwidth:
            cmp     x0, x19
            blo     fsLoop
          ❼ mov     lr, x25         // Return to alternate address.

notINFNaN:  ret
            endp    chkNaNINF
```

The code moves the floating-point value in D0 into X0 and then checks the exponent bits to see if they contain 0x7FF ❶. If the exponent does not contain this value, the procedure returns to the caller (using the return address in LR).

If the exponent bits are 0x7FF, the code checks the mantissa to see if it is 0 or nonzero ❷. If it's nonzero, the code emits the character string `NaN` to the buffer pointed at by X22 ❸. If the mantissa is nonzero, the code checks whether the sign bit is set ❹. If not, this code emits `INF` to the output buffer. If the sign bit is set, the code emits `-INF` to the output buffer ❺.

In all three cases (`NaN`, `INF`, or `-INF`), the code transfers to `fillSpecial` ❻, where it adds sufficient padding characters (the padding character is in W21, and the field width is in X19). Rather than return to the caller, this code transfers control to the address held in X25 ❼. The caller (`r64ToStr` or `e64ToStr`) loads the invalid value return address into X25 prior to calling `chkNaNINF`. I could have set a flag, such as the carry flag, and tested it on return. However, I wanted to demonstrate another way to achieve this, and this approach is slightly more elegant (though arguably less readable).

With `chkNaNINF` out of the way, it's time to take a look at the `r64ToStr` function that the user calls to convert floating-point values into strings:

```
// Listing9-13.S (cont.)
//
// r64ToStr
//
// Converts a REAL64 floating-point number to the
// corresponding string of digits. Note that this
// function always emits the string using decimal
// notation. For scientific notation, use the e10ToBuf
// routine.
//
// On entry:
//
//  D0-         (r64) Real64 value to convert
```

```
//
//  X0-         (outBuf) r64ToStr stores the resulting
//              characters in this string.
//
//  X1-         (fWidth) Field width for the number (note
//              that this is an *exact* field width, not a
//              minimum field width)
//
//  X2-         (decDigits) # of digits to display after the
//              decimal pt
//
//  X3-         (fill) Padding character if the number of
//              digits is smaller than the specified field
//              width
//
//  X4-         (maxLength) Maximum string length
//
// On exit:
//
// Buffer contains the newly formatted string. If the
// formatted value does not fit in the width specified,
// r64ToStr will store "#" characters into this string.
//
// Carry-    Clear if success, set if an exception occurs.
//           If width is larger than the maximum length of
//           the string specified by buffer, this routine
//           will return with the carry set.
//
//************************************************************

            proc    r64ToStr

            // Local variables:

            locals  rts
            qword   rts.x0x1
            qword   rts.x2x3
            qword   rts.x4x5
            qword   rts.x19x20
            qword   rts.x21x22
            qword   rts.x23x24

            dword   rts.x25
            byte    rts.digits, 80
            byte    rts.stk, 64
            endl    rts

            enter   rts.size

            // Use meaningful names for the nonvolatile
            // registers that hold local/parameter values:

            #define fpVal d0
            #define fWidth x19      // chkNaNINF expects this here.
```

```
#define decDigits x20
#define fill w21        // chkNaNINF expects this here.
#define outBuf x22      // chkNaNINF expects this here.
#define maxLength x23
#define exponent x24
#define sign w25
#define failAdrs x25    // chkNaNINF expects this here.

// Preserve registers:

stp     x0,   x1, [fp, #rts.x0x1]
stp     x2,   x3, [fp, #rts.x2x3]
stp     x4,   x5, [fp, #rts.x4x5]
stp     x19, x20, [fp, #rts.x19x20]
stp     x21, x22, [fp, #rts.x21x22]
stp     x23, x24, [fp, #rts.x23x24]
str     x25,      [fp, #rts.x25]

// Move parameter values to nonvolatile
// storage:

mov     outBuf, x0
mov     fWidth, x1
mov     decDigits, x2
mov     fill, w3
mov     maxLength, x4

// First, make sure the number will fit into
// the specified string.

cmp     fWidth, maxLength
bhs     strOverflow

// If the width is 0, return an error:

cmp     fWidth, #0
beq     valOutOfRange

// Handle NaN and INF special cases.
// Note: if the value is invalid, control
// transfers to clcAndRet rather than simply
// returning.

❶ lea     failAdrs, clcAndRet
  bl      chkNaNINF

// Okay, do the conversion. Begin by
// processing the mantissa digits:

  add     x0, fp, #rts.digits // lea x0, rts.digits
❷ bl      FPDigits            // Convert r64 to string.
  mov     exponent, x1        // Save away exponent result.
  mov     sign, w2            // Save mantissa sign char.
```

```
// Round the string of digits to the number of significant
// digits you want to display for this number. Note that
// a maximum of 16 digits are produced for a 53-bit value.

          ❸ cmp     exponent, #maxDigits
            ble     dontForceWidthZero
            mov     x0, xzr         // If the exponent is negative or
                                    // too large, set width to 0.
dontForceWidthZero:
            add     x2, x0, decDigits // Compute rounding position.
            cmp     x2, #maxDigits
            bhs     dontRound       // Don't bother if a big #.

            // To round the value to the number of
            // significant digits, go to the digit just
            // beyond the last one you are considering (X2
            // currently contains the number of decimal
            // positions) and add 5 to that digit.
            // Propagate any overflow into the remaining
            // digit positions.

            add     x2, x2, #1          // Index + 1 of last sig digit
            ldrb    w0, [x1, x2]        // Get that digit.

            add     w0, w0, #5          // Round (for example, +0.5)
            cmp     w0, #'9'
            bls     dontRound

            mov     x0, #('0' + 10)     // Force to 0.
whileDigitGT9:
            sub     w0, w0, #10         // Sub out overflow,
            strb    w0, [x1, x2]        // carry, into prev
            subs    x2, x2, #1          // digit (until first
            bmi     hitFirstDigit       // digit in the #).

            ldrb    w0, [x1, x2]        // Increment previous
            add     w0, w0, #1          // digit.
            strb    w0, [x1, x2]

            cmp     w0, #'9'            // Overflow if > '9'
            bhi     whileDigitGT9
            b.al    dontRound

hitFirstDigit:

            // If you get to this point, you've hit the
            // first digit in the number, so you have to
            // shift all the characters down one position
            // in the string of bytes and put a "1" in the
            // first character position.

          ❹ mov     x2, #maxDigits      // Max digits in value
repeatUntilX2eq0:
```

```
            ldrb    w0, [x1, x2]
            add     x2, x2, #1
            strb    w0, [x1, x2]
            subs    x2, x2, #2
            bne     repeatUntilX2eq0

            mov     w0, #'1'
            strb    w0, [x1, x2]

            add     exponent, exponent, #1 // Increment exponent because
                                           // you added a digit.

dontRound:

            // Handle positive and negative exponents separately.

          ❺ mov     x5, xzr             // Index into output buf.
            cmp     exponent, #0
            bge     positiveExponent

            // Negative exponents:
            // Handle values from 0 to 1.0 here (negative
            // exponents imply negative powers of 10).
            //
            // Compute the number's width. Since this
            // value is from 0 to 1, the width
            // calculation is easy: it's just the number of
            // decimal positions they've specified plus
            // 3 (since you need to allow room for a
            // leading "-0."). X2 = number of digits to emit
            // after "."

            mov     x4, #4
            add     x2, decDigits, #3
            cmp     x2, x4
            csel    x2, x2, x4, hs  // If X2 < X4, X2 = X4

            cmp     x2, fWidth
            bhi     widthTooBig

            // This number will fit in the specified field
            // width, so output any necessary leading pad
            // characters. X3 = number of padding characters
            // to output.

          ❻ sub     x3, fWidth, x2
            b.al    testWhileX3ltWidth

whileX3ltWidth:
            strb    fill, [outBuf, x5]
            add     x5, x5, #1          // Index
            add     x2, x2, #1          // Digits processed
testWhileX3ltWidth:
            cmp     x2, fWidth
            blo     whileX3ltWidth
```

```
                // Output " 0." or "-0.", depending on
                // the sign of the number:

                strb    sign, [outBuf, x5]
                add     x5, x5, #1
                mov     w0, #'0'
                strb    w0, [outBuf, x5]
                add     x5, x5, #1
                mov     w0, #'.'
                strb    w0, [outBuf, x5]
                add     x5, x5, #1
                add     x3, x3, #3

                // Now output the digits after the decimal point:

                mov     x2, xzr             // Count the digits here.
                add     x1, fp, #rts.digits // lea x1, rts.digits

        // If the exponent is currently negative, or if
        // you've output more than 16 significant digits,
        // just output a 0 character.

        repeatUntilX3geWidth:
                mov     x0, #'0'
                adds    exponent, exponent, #1
                bmi     noMoreOutput

                cmp     x2, #maxDigits
                bge     noMoreOutput

                ldrb    w0, [x1]
                add     x1, x1, #1

        noMoreOutput:
                strb    w0, [outBuf, x5]
                add     x5, x5, #1          // Index
                add     x2, x2, #1          // Digits processed
                add     x3, x3, #1          // Digit count
                cmp     x3, fWidth
                blo     repeatUntilX3geWidth
                b.al    r64BufDone

        // If the number's actual width was bigger than the width
        // specified by the caller, emit a sequence of '#' characters
        // to denote the error.

❼ widthTooBig:

                // The number won't fit in the specified field
                // width, so fill the string with the "#"
                // character to indicate an error.

                mov     x2, fWidth
                mov     w0, #'#'
```

```
fillPound:  strb    w0, [outBuf, x5]
            add     x5, x5, #1          // Index
            subs    x2, x2, #1
            bne     fillPound
            b.al    r64BufDone

// Handle numbers with a positive exponent here.
//
// Compute # of print positions consumed by output string.
// This is given by:
//
//                   Exponent     // # of digits to left of "."
//           +       2            // Sign + 1's digit
//           +       decDigits    // Add in digits right of "."
//           +       1            // If there is a decimal point

❽ positiveExponent:

            mov     x3, exponent    // Digits to left of "."
            add     x3, x3, #2      // sign posn
            cmp     decDigits, #0   // See if any fractional
            beq     decPtsIs0       // part.

            add     x3, x3, decDigits // Digits to right of "."
            add     x3, x3, #1        // Make room for the "."

decPtsIs0:

            // Make sure the result will fit in the
            // specified field width.

            cmp     x3, fWidth
            bhi     widthTooBig
            beq     noFillChars

            // If the actual number of print positions
            // is less than the specified field width,
            // output leading pad characters here.

            subs    x2, fWidth, x3
            beq     noFillChars

fillChars:  strb    fill, [outBuf, x5]
            add     x5, x5, #1
            subs    x2, x2, #1
            bne     fillChars

noFillChars:

            // Output the sign character:

            strb    sign, [outBuf, x5]
            add     x5, x5, #1
```

```
                // Okay, output the digits for the number here:

                mov     x2, xzr             // Counts # of output chars
                add     x1, fp, #rts.digits // lea x1, rts.digits

                // Calculate the number of digits to output
                // before and after the decimal point:

                add     x3, decDigits, exponent
                add     x3, x3, #1          // Always one digit before "."

    // If we've output fewer than 16 digits, go ahead
    // and output the next digit. Beyond 16 digits,
    // output 0s.

    repeatUntilX3eq0:

                mov     w0, #'0'
                cmp     x2, #maxDigits
                bhs     putChar

                ldrb    w0, [x1]
                add     x1, x1, #1

    putChar:    strb    w0, [outBuf, x5]
                add     x5, x5, #1

                // If the exponent decrements down to 0,
                // output a decimal point:

                cmp     exponent, #0
                bne     noDecimalPt

                cmp     decDigits, #0
                beq     noDecimalPt

                mov     w0, #'.'
                strb    w0, [outBuf, x5]
                add     x5, x5, #1

    noDecimalPt:
                sub     exponent, exponent, #1  // Count down to "." output.
                add     x2, x2, #1    // # of digits thus far
                subs    x3, x3, #1    // Total # of digits to output
                bne     repeatUntilX3eq0

    // Zero-terminate string and leave:

    r64BufDone: strb    wzr, [outBuf, x5]
❾   clcAndRet:  msr     nzcv, xzr    // clc = no error
                b.al    popRet

    strOverflow:
                mov     x0, #-3 // String overflow
                b.al    ErrorExit
```

```
valOutOfRange:
                mov     x0, #-1 // Range error
❿ ErrorExit:    mrs     x1, nzcv
                orr     x1, x1, #(1 << 29)
                msr     nzcv, x1        // stc = error
                strb    wzr, [outBuf]   // Just to be safe

                // Change X0 on return:

                str     x0, [fp, #rts.x0x1]

popRet:
                ldp     x0, x1,   [fp, #rts.x0x1]
                ldp     x2, x3,   [fp, #rts.x2x3]
                ldp     x4, x5,   [fp, #rts.x4x5]
                ldp     x19, x20, [fp, #rts.x19x20]
                ldp     x21, x22, [fp, #rts.x21x22]
                ldp     x23, x24, [fp, #rts.x23x24]
                ldr     x25,      [fp, #rts.x25]
                leave
                endp    r64ToStr
```

The `r64ToStr` function converts the floating-point value in D0 to a string in standard decimal form, supporting output field widths, number of digits after the decimal point, and fill character for leading positions that would normally be blank.

After appropriate initialization, `r64ToStr` first checks for the values `NaN` (not a number), `INF` (infinity), and `-INF` (minus infinity) ❶; these values require special nonnumeric output strings, which must still be padded to `fWidth` characters. The `r64ToStr` calls `FPDigits` to convert the mantissa to a string of decimal digit characters (and obtain the power-of-10 exponent in integer form) ❷. The next step is to round the number based on the number of digits to appear after the decimal point ❸. This code computes the index into the string produced by `FPDigits` one character beyond the number of digits specified by the `decDigits` parameter. It fetches this character (which will be `'0'` through `'9'`) and adds 5 to its ASCII code. If the result is greater than the ASCII code of `'9'`, the code has to bump the previous digit in the string by 1. Of course, if that character contains `'9'`, overflow will occur and the carry has to ripple through to previous digit(s). If the carry ripples all the way to the first character of the string, the code must shift all the characters one position to the right and insert a `'1'` at the beginning of the string ❹.

Next, the code emits the characters associated with the final decimal string. The algorithm splits into two sections ❺, with one section handling positive (and 0) exponents and the other handling negative exponents. For negative exponents, the code will emit any fill characters, the sign of the number (still held in X2), and the `decDigits` digits from the mantissa string conversion ❻. If the field width and `decDigits` are sufficiently large, the code will simply output the `'0'` character for all characters beyond the 16th significant digit. If the number of output digits would exceed the field width

the caller passes, the `widthTooBig` code ❼ will emit # characters to indicate a formatting error (the standard HLL approach to format errors in floating-point conversions).

The code handles floating-point conversions of values greater than or equal to 1.0 (positive exponents) ❽. This code emits necessary padding characters and the value's sign, then calculates the position of the decimal point in the output string and rounds the last digit throughout the string, as previously described. It then outputs the characters returned by `FPDigits` up to that position. Finally, it outputs the decimal point, followed by the remaining fractional digits. If it turns out that the code cannot fit the number into the field width (and decimal digits) specified, it transfers control to `widthTooBig` to produce the error string.

To notify the caller of possible errors, this code clears the carry flag upon return ❾ if the conversion was successful, or sets the carry flag on return if there was an error ❿. This allows the caller to easily test for success/failure with a single `bcs` or `bcc` instruction after the call to `r64ToStr`.

The final output format handled by Listing 9-13 is exponential (scientific) form. Two functions handle this conversion: `expToBuf` and `e64ToStr`. The former handles the formatting of the exponent portion of the output string:

```
// Listing9-13.S (cont.)
//
// expToBuf
//
// Unsigned integer to buffer
// Used to output up to three-digit exponents
//
// Inputs:
//
//  X0-   Unsigned integer to convert
//  X1-   Exponent print width 1-3
//  X2-   Points at buffer (must have at least 4 bytes)
//
// Outputs:
//
//  Buffer contains the string representing the converted
//  exponent.
//
//  Carry is clear on success, set on error.

            proc    expToBuf
            stp     x0, lr, [sp, #-16]!
            stp     x1, x3, [sp, #-16]!
            stp     x4, x5, [sp, #-16]!

            mov     x5, xzr     // Initialize output string.
            mov     x4, #10     // For division by 10

// Verify exponent digit count is in the range 1-3:
```

```
          ❶ cmp     x1, #1
            blo     badExp
            cmp     x1, #3
            bhi     badExp

// Verify the actual exponent will fit in the number of digits:

          ❷ cmp     x1, #2
            blo     oneDigit
            beq     twoDigits

            // Must be 3:

            cmp     x0, #1000
            bhs     badExp

// Convert three-digit value to a string:

          ❸ udiv    x1, x0, x4      // X1 = quotient
            msub    x3, x1, x4, x0  // X3 = remainder
            orr     x3, x3, #'0'
            orr     x5, x3, x5, lsl #8

            udiv    x0, x1, x4      // X0 = quotient
            msub    x3, x0, x4, x1  // X3 = remainder
            orr     x3, x3, #'0'
            orr     x5, x3, x5, lsl #8

            udiv    x1, x0, x4      // X1 = quotient
            msub    x3, x1, x4, x0  // X3 = remainder
            orr     x3, x3, #'0'
            orr     x5, x3, x5, lsl #8

            b.al    outputExp

// Single digit is easy:

oneDigit:
          ❹ cmp     x0, #10
            bhs     badExp

            orr     x5, x0, #'0'
            b.al    outputExp

// Convert value in the range 10-99 to a string
// containing two characters:

twoDigits:
          ❺ cmp     x0, #100
            bhs     badExp

            udiv    x1, x0, x4      // X1 = quotient
            msub    x3, x1, x4, x0  // X3 = remainder
            orr     x3, x3, #'0'
            orr     x5, x3, x5, lsl #8
```

```
            udiv    x0, x1, x4      // X0 = quotient
            msub    x3, x0, x4, x1  // X3 = remainder
            orr     x3, x3, #'0'
            orr     x5, x3, x5, lsl #8

// Store the string into the buffer (includes a 0
// byte in the HO positions of W5):

outputExp:
          ❻ str     w5, [x2]
            ldp     x4, x5, [sp], #16
            ldp     x1, x3, [sp], #16
            ldp     x0, lr, [sp], #16
            msr     nzcv, xzr    // clc = no error
            ret
            leave

badExp:
            ldp     x4, x5, [sp], #16
            ldp     x1, x3, [sp], #16
            ldp     x0, lr, [sp], #16
            mrs     x0, nzcv
            orr     x0, x0, #(1 << 29)
            msr     nzcv, x0        // stc = error
            mov     x0, #-1         // Value out of range ...
            ret
            endp    expToBuf
```

The `expToBuf` function produces a string of exactly one, two, or three digits (based on the parameters the caller passes in X0 and X1). The `expToBuf` function begins by verifying that the exponent digit count is within range ❶ and that the actual exponent will fit in the number of digits specified ❷. The code branches to three separate output conversion code sequences if the exponent output is three digits (the normal case ❸), one digit ❹, or two digits ❺. The code stores those characters into the buffer where X2 points ❻.

The function returns the error status in the carry flag, returning with the carry clear for a successful operation, or the carry set if the exponent is too large or the converted number will not fit in the number of character positions that X1 specifies. Other than this, `expToBuf` is basically a `switch` statement (implemented using `if...then...else` logic) that has three cases: one for each exponent size (one, two, or three characters).

The `e64ToStr` function handles the conversion from double-precision to string using exponential format:

```
// Listing9-13.S (cont.)
//
// e64ToStr
//
// Converts a REAL64 floating-point number to the
// corresponding string of digits. Note that this
```

```
// function always emits the string using scientific
// notation; use the r64ToStr routine for decimal notation.
//
// On entry:
//
//  D0-     (e64) Double-precision value to convert
//
//  X0-     (buffer) e64ToStr stores the resulting characters in
//          this buffer.
//
//  X1-     (width) Field width for the number (note that this
//          is an *exact* field width, not a minimum
//          field width)
//
//  X2-     (fill) Padding character if the number is smaller
//          than the specified field width
//
//  X3-     (expDigs) Number of exponent digits (2 for real32
//          and 3 for real64)
//
//  X4-     (maxLength) Maximum buffer size
//
// On exit:
//
//  Buffer contains the newly formatted string. If the
//  formatted value does not fit in the width specified,
//  e64ToStr will store "#" characters into this string.
//
//  Carry-  Clear if no error, set if error.
//          If error, X0 is
//              -3 if string overflow
//              -2 if bad width
//              -1 if value out of range
//
//-----------------------------------------------------------
//
// Unlike the integer-to-string conversions, this routine
// always right-justifies the number in the specified
// string. Width must be a positive number; negative
// values are illegal (actually, they are treated as
// *really* big positive numbers that will always raise
// a string overflow exception).
//
//***********************************************************

            proc       e64ToStr

#define     e2sWidth   x19      // chkNaNINF expects this here.
#define     e2sExp     x20
#define     e2sFill    x21      // chkNaNINF expects this here.
#define     e2sBuffer  x22      // chkNaNINF expects this here.
#define     e2sMaxLen  x23
#define     e2sExpDigs x24
```

```
#define     e2sSign    w25
#define     eFailAdrs  x25      // chkNaNINF expects this here.
#define     e2sMantSz  x26

            locals  e2s
            qword   e2s.x1x2
            qword   e2s.x3x4
            qword   e2s.x5x19
            qword   e2s.x20x21
            qword   e2s.x22x23
            qword   e2s.x24x25
            qword   e2s.x26x27
            dword   e2s.x0
            dword   e2s.d0
            byte    e2s.digits, 64
            byte    e2s.stack, 64
            endl    e2s

            // Build activation record and preserve registers:

            enter   e2s.size
            str     x0,       [fp, #e2s.x0]
            stp     x1,  x2,  [fp, #e2s.x1x2]
            stp     x3,  x4,  [fp, #e2s.x3x4]
            stp     x5,  x19, [fp, #e2s.x5x19]
            stp     x20, x21, [fp, #e2s.x20x21]
            stp     x22, x23, [fp, #e2s.x22x23]
            stp     x24, x25, [fp, #e2s.x24x25]
            stp     x26, x27, [fp, #e2s.x26x27]
            str     d0,       [fp, #e2s.d0]

            // Move important data to nonvolatile registers:

            mov     e2sBuffer, x0
            mov     e2sWidth, x1
            mov     e2sFill, x2
            mov     e2sExpDigs, x3
            mov     e2sMaxLen, x4

            // See if the width is greater than the buffer size:

            cmp     e2sWidth, e2sMaxLen
            bhs     strOvfl

            strb    wzr, [e2sBuffer, e2sWidth]  // Zero-terminate str.

// First, make sure the width isn't 0:

          ❶ cmp     e2sWidth, #0
            beq     valOutOfRng

// Just to be on the safe side, don't allow widths greater
// than 1024:
```

```
            cmp     e2sWidth, #1024
            bhi     badWidth

// Check for NaN and INF:

          ❷ lea     failAdrs, exit_eToBuf   // Note: X25, used before
            bl      chkNaNINF               // e2sSign (also X25)

// Okay, do the conversion:

          ❸ add     x0, fp, #e2s.digits // lea x1, e2s.digits
            bl      FPDigits        // Convert D0 to digit str.
            mov     e2sExp, x1      // Save away exponent result.
            mov     e2sSign, w2     // Save mantissa sign char.

// Verify that there is sufficient room for the mantissa's sign,
// the decimal point, two mantissa digits, the "E",
// and the exponent's sign. Also add in the number of digits
// required by the exponent (2 for single, 3 for double).
//
// -1.2e+00    :real4
// -1.2e+000   :real8

          ❹ add     x2, e2sExpDigs, #6    // Minimum number of posns
            cmp     x2, e2sWidth
            bls     goodWidth

// Output a sequence of "#...#" chars (to the specified width)
// if the width value is not large enough to hold the
// conversion:

            mov     x2, e2sWidth
            mov     x0, #'#'
            mov     x1, e2sBuffer
fillPnd:    strb    w0, [x1]
            add     x1, x1, #1
            subs    x2, x2, #1
            bne     fillPnd
            b.al    exit_eToBuf

// Okay, the width is sufficient to hold the number; do the
// conversion and output the string here:

goodWidth:
            // Compute the # of mantissa digits to display,
            // not counting mantissa sign, decimal point,
            // "E", and exponent sign:

          ❺ sub     e2sMantSz, e2sWidth, e2sExpDigs
            sub     e2sMantSz, e2sMantSz, #4

            // Round the number to the specified number of
            // print positions. (Note: since there are a
            // maximum of 16 significant digits, don't
```

```
                // bother with the rounding if the field width
                // is greater than 16 digits.)

                cmp     e2sMantSz, #maxDigits
                bhs     noNeedToRound

                // To round the value to the number of
                // significant digits, go to the digit just
                // beyond the last one you are considering (e2sMantSz
                // currently contains the number of decimal
                // positions) and add 5 to that digit.
                // Propagate any overflow into the remaining
                // digit positions.

                add     x1, e2sMantSz, #1
                add     x2, fp, #e2s.digits // lea x2, e2s.digits
                ldrb    w0, [x2, x1]        // Get least sig digit + 1.
                add     w0, w0, #5          // Round (for example, +0.5).
                cmp     w0, #'9'
                bhi     whileDigGT9
                b.al    noNeedToRound

// Sneak this code in here, after a branch, so the
// loop below doesn't get broken up.

firstDigitInNumber:

                // If you get to this point, you've hit the
                // first digit in the number, so you have to
                // shift all the characters down one position
                // in the string of bytes and put a "1" in the
                // first character position.

                ldr     x0, [x2, #8]
                str     x0, [x2, #9]
                ldr     x0, [x2]
                str     x0, [x2, #1]

                mov     x0, #'1'        // Store '1' in 1st
                strb    w0, [x2]        // digit position.

                // Bump exponent by 1, as the shift did
                // a divide by 10.

                add     e2sExp, e2sExp, #1
                b.al    noNeedToRound

// Subtract out overflow and add the carry into the previous
// digit (unless you hit the first digit in the number):

whileDigGT9:
                sub     w0, w0, #10
                strb    w0, [x2, x1]
                subs    x1, x1, #1
                bmi     firstDigitInNumber
```

```
            // Add in carry to previous digit:

            ldrb    w0, [x2, x1]
            add     w0, w0, #1
            strb    w0, [x2, x1]
            cmp     w0, #'9'        // Overflow if char > '9'
            bhi     whileDigGT9

noNeedToRound:
            add     x2, fp, #e2s.digits // lea x2, e2s.digits

// Okay, emit the string at this point. This is pretty easy,
// since all you really need to do is copy data from the
// digits array and add an exponent (plus a few other simple chars).

          ❻ mov     x1, #0      // Count output mantissa digits.
            strb    e2sSign, [e2sBuffer], #1

// Output the first character and a following decimal point
// if there are more than two mantissa digits to output.

            ldrb    w0, [x2]
            strb    w0, [e2sBuffer], #1
            add     x1, x1, #1
            cmp     x1, e2sMantSz
            beq     noDecPt

            mov     w0, #'.'
            strb    w0, [e2sBuffer], #1

noDecPt:

// Output any remaining mantissa digits here.
// Note that if the caller requests the output of
// more than 16 digits, this routine will output 0s
// for the additional digits.

            b.al    whileX2ltMantSizeTest

whileX2ltMantSize:

            mov     w0, #'0'
            cmp     x1, #maxDigits
            bhs     justPut0

            ldrb    w0, [x2, x1]

justPut0:
            strb    w0, [e2sBuffer], #1
            add     x1, x1, #1

whileX2ltMantSizeTest:

            cmp     x1, e2sMantSz
            blo     whileX2ltMantSize
```

```
// Output the exponent:

          ❼ mov     w0, #'e'
            strb    w0, [e2sBuffer], #1
            mov     w0, #'+'
            mov     w4, #'-'
            neg     x5, e2sExp

            cmp     e2sExp, #0
            csel    w0, w0, w4, ge
            csel    e2sExp, e2sExp, x5, ge

            strb    w0, [e2sBuffer], #1

            mov     x0, e2sExp
            mov     x1, e2sExpDigs
            mov     x2, e2sBuffer
            bl      expToBuf
            bcs     error

exit_eToBuf:
            msr     nzcv, xzr    // clc = no error
            ldr     x0, [fp, #e2s.x0]

returnE64:
            ldp     x1,  x2,  [fp, #e2s.x1x2]
            ldp     x3,  x4,  [fp, #e2s.x3x4]
            ldp     x5,  x19, [fp, #e2s.x5x19]
            ldp     x20, x21, [fp, #e2s.x20x21]
            ldp     x22, x23, [fp, #e2s.x22x23]
            ldp     x24, x25, [fp, #e2s.x24x25]
            ldp     x26, x27, [fp, #e2s.x26x27]
            ldr     d0,       [fp, #e2s.d0]
            leave

strOvfl:    mov     x0, #-3
            b.al    error

badWidth:   mov     x0, #-2
            b.al    error

valOutOfRng:
            mov     x0, #-1
error:
            mrs     x1, nzcv
            orr     x1, x1, #(1 << 29)
            msr     nzcv, x1        // stc = error
            b.al    returnE64

            endp    e64ToStr
```

Converting the mantissa to a string is very similar to the routine in `r64ToStr`, though exponential form is a little easier, as the format always places the decimal point immediately after the first mantissa digit. As with

r64ToStr, e64ToStr begins by checking the input parameters to see if they are valid ❶ (returning with the carry flag set and an error code in X0 if an error occurred). After parameter validation, the code checks for NaN or INF ❷. It then calls FPDigits to convert the mantissa to a string of digits ❸ (held in a local buffer). This call also returns the sign of the value as well as a decimal integer exponent.

After calculating the decimal exponent value, the e64ToStr function checks whether the converted value will fit into the space specified by the Width input parameter ❹. If the converted number would be too large, e64ToStr emits a string of # characters to denote an error.

Note that this situation is not considered an error in the sense of returning the carry flag set. If the caller specifies an insufficient field width, the function still succeeds in creating a string conversion; that string just happens to be filled with # characters. The carry flag is set, on error, when e64ToStr cannot produce an output string.

After verifying that the string will fit in the specified field width, the e64ToStr function rounds the result to the specified number of decimal digits ❺. This algorithm is identical to that used by r64ToStr. Next, the code outputs the mantissa digits ❻. Again, this is similar to the way r64ToStr works, except that the decimal point is always placed after the first digit (no need to calculate its position). Finally, the code emits e followed by the exponent's sign character ❼ and then calls expToBuf to convert the exponent to a one-, two-, or three-digit character sequence (specified by the expDigs parameter the caller passes in X3).

The remaining code in Listing 9-13 provides utility functions used by the main program to display data (r64Print and e64Print), along with the asmMain procedure that demonstrates floating-point output using the functions in this section:

```
// Listing9-13.S (cont.)
//
            proc    r64Print

            stp     x0, x1, [sp, #-16]!
            stp     x2, x3, [sp, #-16]!
            stp     x4, x5, [sp, #-16]!
            stp     x6, x7, [sp, #-16]!
            stp     x8, lr, [sp, #-16]!
            sub     sp, sp, #64

            lea     x0, fmtStr1
            lea     x1, r64str_1
            mstr    x1, [sp]
            bl      printf

            add     sp, sp, #64
            ldp     x8, lr, [sp], #16
            ldp     x6, x7, [sp], #16
            ldp     x4, x5, [sp], #16
            ldp     x2, x3, [sp], #16
```

```
            ldp     x0, x1, [sp], #16
            ret
            endp    r64Print

            proc    e64Print
            stp     x0, x1, [sp, #-16]!
            stp     x2, x3, [sp, #-16]!
            stp     x4, x5, [sp, #-16]!
            stp     x6, x7, [sp, #-16]!
            stp     x8, lr, [sp, #-16]!
            sub     sp, sp, #64

            lea     x0, fmtStr3
            lea     x1, r64str_1
            mstr    x1, [sp]
            bl      printf

            add     sp, sp, #64
            ldp     x8, lr, [sp], #16
            ldp     x6, x7, [sp], #16
            ldp     x4, x5, [sp], #16
            ldp     x2, x3, [sp], #16
            ldp     x0, x1, [sp], #16
            ret
            endp    e64Print
```

Note that these functions preserve all the nonvolatile registers because `printf()` can modify them.

The `asmMain` function is a typical demonstration program for the floating-point string-conversion functions. It calls the `r64ToStr` and `e64ToStr` functions with various input parameters to demonstrate the use of these functions:

```
// Listing9-13.S (cont.)
//
❶ r64_1:      .double  1.234567890123456
              .double  0.0000000000000001
              .double  1234567890123456.0
              .double  1234567890.123456
              .double  99499999999999999.0
              .dword   0x7ff0000000000000
              .dword   0xfff0000000000000
              .dword   0x7fffffffffffffff
              .dword   0xffffffffffffffff
              .double  0.0
              .double  -0.0
fCnt          =        (. - r64_1)

rSizes:       .word    12, 12, 2, 7, 0, 0, 0, 0, 0, 2, 2

e64_1:        .double  1.234567890123456e123
              .double  1.234567890123456e-123
e64_3:        .double  1.234567890123456e1
```

```
            .double  1.234567890123456e-1
            .double  1.234567890123456e10
            .double  1.234567890123456e-10
            .double  1.234567890123456e100
            .double  1.234567890123456e-100
            .dword   0x7ff0000000000000
            .dword   0xfff0000000000000
            .dword   0x7fffffffffffffff
            .dword   0xffffffffffffffff
            .double  0.0
            .double  -0.0
eCnt         =       (. - e64_1)

eSizes:     .word    6, 9, 8, 12, 14, 16, 18, 20, 12, 12, 12, 12, 8, 8
expSizes:   .word    3, 3, 2, 2, 2, 2, 3, 3, 2, 2, 2, 2, 2, 2

// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            dword   am.x8x9
            dword   am.x27
            byte    am.stk, 64
            endl    am

            enter   am.size     // Activation record
            stp     x8, x9, [fp, #am.x8x9]
            str     x27,    [fp, #am.x27]

// F output

            mov     x2, #16         // decDigits
fLoop:
            ldr     d0, r64_1
            lea     x0, r64str_1    // Buffer
            mov     x1, #30         // fWidth
            mov     x3, #'.'        // Fill
            mov     x4, 32          // maxLength
            bl      r64ToStr
            bcs     fpError
            bl      r64Print
            subs    x2, x2, #1
            bpl     fLoop

            lea     x0, newlines
            bl      printf

            lea     x5, r64_1
            lea     x6, rSizes
            mov     x7, #fCnt/8
f2Loop:     ldr     d0, [x5], #8
            lea     x0, r64str_1    // Buffer
            mov     x1, #30         // fWidth
```

```
            ldr     w2, [x6], #4    // decDigits
            mov     x3, #'.'        // Fill
            mov     x4, #32         // maxLength
            bl      r64ToStr
            bcs     fpError
            bl      r64Print
            subs    x7, x7, #1
            bne     f2Loop

// E output

            lea     x0, expStr
            bl      printf

            lea     x5, e64_1
            lea     x6, eSizes
            lea     x7, expSizes
            mov     x8, #eCnt/8
eLoop:
            ldr     d0, [x5], #8
            lea     x0, r64str_1    // Buffer
            ldr     w1, [x6], #4    // fWidth
            mov     x2, #'.'        // Fill
            ldr     w3, [x7], #4    // expDigits
            mov     x4, #32         // maxLength
            bl      e64ToStr
            bcs     fpError
            bl      e64Print
            subs    x8, x8, #1
            bne     eLoop
            b.al    allDone

fpError:
            mov     x1, x0
            lea     x0, fmtStr2
            mstr    x1, [sp]
            bl      printf

allDone:
            ldp     x8, x9, [fp, #am.x8x9]
            ldr     x27,    [fp, #am.x27]
            leave
            endp    asmMain
```

Listing 9-13 places the floating-point constant values in the code section rather than a read-only data section ❶, making it easier to modify them when looking at the main program.

The following is the `build` command and sample output for Listing 9-13:

```
% ./build Listing9-13
% 1G
Calling Listing9-13:
r64ToStr: value='.......... 1.2345678901234560'
```

```
r64ToStr: value='............ 1.234567890123456'
r64ToStr: value='........... 1.23456789012345'
r64ToStr: value='.............. 1.2345678901234'
r64ToStr: value='............... 1.234567890123'
r64ToStr: value='................ 1.23456789012'
r64ToStr: value='................. 1.2345678901'
r64ToStr: value='.................. 1.234567890'
r64ToStr: value='................... 1.23456789'
r64ToStr: value='.................... 1.2345678'
r64ToStr: value='..................... 1.234567'
r64ToStr: value='...................... 1.23456'
r64ToStr: value='....................... 1.2345'
r64ToStr: value='........................ 1.234'
r64ToStr: value='......................... 1.23'
r64ToStr: value='.......................... 1.2'
r64ToStr: value='........................... 1'


r64ToStr: value='............... 1.234567890123'
r64ToStr: value='............... 0.000000000000'
r64ToStr: value='.......... 1234567890123456.00'
r64ToStr: value='........... 1234567890.1234560'
r64ToStr: value='............ 99500000000000000'
r64ToStr: value='INF                           '
r64ToStr: value='-INF                          '
r64ToStr: value='NaN                           '
r64ToStr: value='NaN                           '
r64ToStr: value='......................... 0.00'
r64ToStr: value='........................-0.00'


Testing e64ToStr:

e64ToStr: value='######'
e64ToStr: value=' 1.2e-123'
e64ToStr: value=' 1.2e+01'
e64ToStr: value=' 1.23456e-01'
e64ToStr: value=' 1.2345678e+10'
e64ToStr: value=' 1.234567890e-10'
e64ToStr: value=' 1.2345678901e+100'
e64ToStr: value=' 1.234567890123e-100'
e64ToStr: value='INF          '
e64ToStr: value='-INF         '
e64ToStr: value='NaN          '
e64ToStr: value='NaN          '
e64ToStr: value=' 0.0e+00'
e64ToStr: value='-0.0e+00'
Listing9-13 terminated
```

This output demonstrates double-precision floating-point output. If you want to convert a single-precision value to a string, first convert the single-precision value to double-precision and use this code to translate the resulting double-precision value to a string.

9.3 String-to-Numeric Conversions

The routines converting numeric values to strings, and strings to numeric values, have two fundamental differences. First of all, numeric-to-string conversions generally occur without possibility of error (assuming you have allocated a sufficiently large buffer so that the conversion routines don't write data beyond the end of the buffer). String-to-numeric conversions, on the other hand, must handle the real possibility of errors like illegal characters and numeric overflow.

A typical numeric input operation consists of reading a string of characters from the user and then translating this string of characters into an internal numeric representation. For example, in C++ a statement like `cin >> i32;` reads a line of text from the user and converts a sequence of digits appearing at the beginning of that line of text into a 32-bit signed integer (assuming `i32` is a 32-bit `int` object). The `cin >> i32;` statement skips over certain characters, like leading spaces, in the string that may appear before the actual numeric characters. The input string may also contain additional data beyond the end of the numeric input (for example, it is possible to read two integer values from the same input line), and therefore the input conversion routine must determine where the numeric data ends in the input stream.

Typically, C++ achieves this by looking for a character from a set of *delimiter* characters. The delimiter character set could be something as simple as any character that is not a numeric digit; or the set could be the whitespace characters (space, tab, and so on) along with perhaps a few other characters such as a comma (`,`) or another punctuation character. For the sake of example, the code in this section assumes that any leading spaces or tab characters (ASCII code 9) may precede the first numeric digit and that the conversion stops on the first non-digit character it encounters. Possible error conditions are as follows:

- No numeric digits at all at the beginning of the string (following any spaces or tabs).
- The string of digits is a value that would be too large for the intended numeric size (for example, 64 bits).

It will be up to the caller to determine whether the numeric string ends with an invalid character upon return from the function call.

9.3.1 Decimal Strings to Integers

The basic algorithm to convert a string containing decimal digits to a number is the following:

1. Initialize an accumulator variable to 0.
2. Skip any leading spaces or tabs in the string.
3. Fetch the first character after the spaces/tabs.

4. If the character is not a numeric digit, return an error. If the character is a numeric digit, fall through to step 5.
5. Convert the numeric character to a numeric value (using AND 0xf).
6. Set the accumulator = (accumulator × 10) + current numeric value.
7. If overflow occurs, return and report an error. If no overflow occurs, fall through to step 8.
8. Fetch the next character from the string.
9. If the character is a numeric digit, go back to step 5; otherwise, fall through to step 10.
10. Return success, with the accumulator containing the converted value.

For signed integer input, you use this same algorithm with the following modifications:

- If the first non-space/tab character is a hyphen (-), set a flag denoting that the number is negative and skip the - character. If the first character is not -, clear the flag.
- At the end of a successful conversion, if the flag is set, negate the integer result before returning (you must check for overflow on the negate operation).

Listing 9-14 implements the conversion algorithm; I've again broken this listing into several sections to better annotate it. The first section contains the usual format strings, along with various sample strings the main program uses to test the `strtou` and `strtoi` functions.

```
// Listing9-14.S
//
// String-to-numeric conversion

            #include    "aoaa.inc"

false       =           0
true        =           1
tab         =           9

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-14"
fmtStr1:    .ascii      "strtou: String='%s'\n"
            .asciz      "    value=%llu\n"

fmtStr2:    .ascii      "Overflow: String='%s'\n"
            .asciz      "    value=%llx\n"

fmtStr3:    .ascii      "strtoi: String='%s'\n"
            .asciz      "    value=%lli\n"

unexError:  .asciz      "Unexpected error in program\n"
```

```
value1:     .asciz      "  1"
value2:     .asciz      "12 "
value3:     .asciz      " 123 "
value4:     .asciz      "1234"
value5:     .asciz      "1234567890123456789"
value6:     .asciz      "18446744073709551615"
OFvalue:    .asciz      "18446744073709551616"
OFvalue2:   .asciz      "99999999999999999999"

ivalue1:    .asciz      "  -1"
ivalue2:    .asciz      "-12 "
ivalue3:    .asciz      " -123 "
ivalue4:    .asciz      "-1234"
ivalue5:    .asciz      "-1234567890123456789"
ivalue6:    .asciz      "-18446744073709551615"
OFivalue:   .asciz      "18446744073709551616"
OFivalue2:  .asciz      "-18446744073709551616"

            .code
            .extern     printf

//////////////////////////////////////////////////////////////////////
//
// Return program title to C++ program:

            proc        getTitle, public
            lea         x0, ttlStr
            ret
            endp        getTitle
```

This program doesn't have any static, writable data; all variable data is kept in registers or in local variables.

The following code is the `strtou` function, which converts strings containing decimal digits to an unsigned integer:

```
// Listing9-14.S (cont.)
//
//////////////////////////////////////////////////////////////////////
//
// strtou
//
// Converts string data to a 64-bit unsigned integer
//
// Input:
//
//   X1-    Pointer to buffer containing string to convert
//
// Outputs:
//
//   X0-    Contains converted string (if success), error code
//          if an error occurs
//
//   X1-    Points at first char beyond end of numeric string
//          If error, X1's value is restored to original value.
```

```
//          Caller can check character at [X1] after a
//          successful result to see if the character following
//          the numeric digits is a legal numeric delimiter.
//
//   C-     (carry flag) Set if error occurs, clear if
//          conversion was successful. On error, X0 will
//          contain 0 (illegal initial character) or
//          0ffffffffffffffffh (overflow).

            proc    strtou

            str     x5, [sp, #-16]!
            stp     x3, x4, [sp, #-16]!
            stp     x1, x2, [sp, #-16]!

            mov     x3, xzr
            mov     x0, xzr
            mov     x4, #10     // Used to mul by 10

            // The following loop skips over any whitespace (spaces and
            // tabs) that appear at the beginning of the string:

          ❶ sub     x1, x1, #1      // Incremented below
skipWS:     ldrb    w2, [x1, #1]!   // Fetch next (first) char.
            cmp     w2, #' '
            beq     skipWS
            cmp     w2, #tab
            beq     skipWS

            // If you don't have a numeric digit at this
            // point, return an error.

          ❷ cmp     w2, #'0'  // Note: '0' < '1' < ... < '9'
            blo     badNumber
            cmp     w2, #'9'
            bhi     badNumber

// Okay, the first digit is good. Convert the string
// of digits to numeric form.
//
// Have to check for unsigned integer overflow here.
// Unfortunately, madd does not set the carry or
// overflow flag, so you have to use umulh to see if
// overflow occurs after a multiplication and do
// an explicit add (rather than madd) to add the
// digit into the accumulator (X0).

❸ convert:  umulh   x5, x0, x4      // Acc * 10
            cmp     x5, xzr
            bne     overflow
            and     x2, x2, #0xf    // Char -> numeric in X2
            mul     x0, x0, x4      // Can't use madd!
            adds    x0, x0, x2      // Add in digit.
            bcs     overflow
```

```
❹ ldrb    w2, [x1, #1]!   // Get next char.
  cmp     w2, #'0'        // Check for digit.
  blo     endOfNum
  cmp     w2, #'9'
  bls     convert

// If you get to this point, you've successfully converted
// the string to numeric form. Return without restoring
// the value in X1 (X1 points at end of digits).

❺ endOfNum:   ldp     x3, x4, [sp], #16   // Really X1, X2
              mov     x2, x4
              ldp     x3, x4, [sp], #16
              ldr     x5, [sp], #16

              // Because the conversion was successful, this
              // procedure leaves X1 pointing at the first
              // character beyond the converted digits.
              // Therefore, we don't restore X1 from the stack.

              msr     nzcv, xzr    // clr c = no error
              ret

// badNumber- Drop down here if the first character in
//            the string was not a valid digit.

❻ badNumber:  mov     x0, xzr
  errorRet:   mrs     x1, nzcv    // Return error in carry flag.
              orr     x1, x1, #(1 << 29)
              msr     nzcv, x1    // Set c = error.

              ldp     x1, x2, [sp], #16
              ldp     x3, x4, [sp], #16
              ldr     x5, [sp], #16
              ret

// overflow- Drop down here if the accumulator overflowed
//           while adding in the current character.

  overflow:   mov     x0, #-1  // 0xFFFFFFFFFFFFFFFF
              b.al    errorRet
              endp    strtou
```

On entry into `strtou`, the X1 register points at the first character of the string to convert. This function begins by skipping over any whitespace characters (spaces and tabs) in the string, leaving X1 pointing at the first non-space/non-tab character ❶.

After any whitespace characters, the first character must be a decimal digit, or `strtou` must return a conversion error. Therefore, after finding a non-whitespace character, the code checks to see that the character is in the range `'0'` to `'9'` ❷.

After verifying that the first character is a digit, the code enters the main conversion loop ❸. Normally, you'd just convert the character to an

integer (by ANDing with 0xF), multiply the accumulator in X0 by 10, and add in the character's value. This could be done using two instructions:

```
and  x2, x2, #0xf
madd x0, x0, x4, x2 // X4 contains 10.
```

The only problem is that you cannot detect overflow by using these two instructions (something that the `strtou` function must do). To detect an overflow due to the multiplication by 10, the code must use the `umulh` instruction and check the result for 0 (if it is not 0, overflow occurs) ❸. If the `umulh` result is 0, the code can multiply the accumulator (X0) by 10 without fear of overflow. Of course, overflow can still occur when adding the character's value to the product of X0 and 10, so you still cannot use `madd`; instead, you must multiply the accumulator by 10, then use the `adds` instruction to add in the character value and check the carry flag immediately thereafter.

The convert loop repeats this process until either an overflow occurs or it encounters a nondigit character. Once it encounters a nondigit character ❹, the converted integer value is in the X0 register, and the function returns with the carry clear. Note that if the conversion is successful, the `strtou` function does not restore the X1 register; instead, it returns with X1 pointing at the first nondigit character ❺. It is the caller's responsibility to check this character to see if it is legitimate.

In the event of an overflow or an illegal starting character, the function returns with the carry flag set and an error code in X0 ❻.

The following code is the `strtoi` procedure, which is the signed-integer version of the `strtou` procedure:

```
// Listing9-14.S (cont.)
//
// strtoi
//
// Converts string data to a 64-bit signed integer
//
// Input:
//
//   X1-    Pointer to buffer containing string to convert
//
// Outputs:
//
//   X0-    Contains converted string (if success), error code
//          if an error occurs
//
//   X1-    Points at first char beyond end of numeric string.
//          If error, X1's value is restored to original value.
//          Caller can check character at [X1] after a
//          successful result to see if the character following
//          the numeric digits is a legal numeric delimiter.
//
//   C-    (carry flag) Set if error occurs, clear if
//         conversion was successful. On error, X0 will
```

```
//         contain 0 (illegal initial character) or
//         -1 (overflow).

tooBig:     .dword  0x7fffffffffffffff

            proc    strtoi

            locals  si
            qword   si.saveX1X2
            endl    si

            enter   si.size

            // Preserve X1 in case you have to restore it;
            // X2 is the sign flag:

            stp     x1, x2, [fp, #si.saveX1X2]

            // Assume you have a nonnegative number:

            mov     x2, #false

// The following loop skips over any whitespace (spaces and
// tabs) that appear at the beginning of the string:

          ❶ sub     x1, x1, #1  // Adjust for +1 below.
skipWSi:    ldrb    w0, [x1, #1]!
            cmp     w0, #' '
            beq     skipWSi
            cmp     w0, #tab
            beq     skipWSi

            // If the first character you've encountered is
            // '-', then skip it, but remember that this is
            // a negative number:

          ❷ cmp     w0, #'-'
            bne     notNeg
            mov     w2, #true
            add     x1, x1, #1  // Skip '-'

❸ notNeg:   bl      strtou       // Convert string to integer.
            bcs     hadError

            // strtou returned success. Check the negative
            // flag and negate the input if the flag
            // contains true:

          ❹ cmp     w2, #true
            bne     itsPosOr0

            negs    x0, x0
            bvs     overflowi
            ldr     x2, [fp, #si.saveX1X2+8]
```

```
            msr     nzcv, xzr   // clr c = no error
            leave

// Success, so don't restore X1:

itsPosOr0:
            ldr     x2, tooBig
            cmp     x0, x2      // Number is too big.
            bhi     overflowi
            ldr     x2, [fp, #si.saveX1X2+8]
            msr     nzcv, xzr  // clr c = no error
            leave

// If you have an error, you need to restore RDI from the stack:

overflowi:  mov     x0, #-1     // Indicate overflow.
hadError:
            mrs     x2, nzcv    // Return error in carry flag.
            orr     x2, x2, #(1 << 29)
            msr     nzcv, x2    // Set c = error.
            ldp     x1, x2, [fp, #si.saveX1X2]
            leave
            endp    strtoi
```

The `strtoi` function converts a string containing a signed integer to the corresponding value in X0. The code begins by eliminating whitespace ❶, then checks for a `'-'` character ❷. The function maintains a "negative flag" in the X2 register (`0` = nonnegative, `1` = negative). After skipping the optional sign character, the code calls the `strtou` function to convert the following string to an unsigned value ❸.

Upon return from `strtou`, the `strtoi` function checks the sign flag in X2 and negates the number if it's supposed to be negative ❹. In both cases (negative or nonnegative), the code also checks for an overflow condition and returns an error if an overflow occurred.

As for `strtou`, the `strtoi` function does not restore X1 if the conversion was successful. However, it will restore X1 if an overflow occurred or if `strtou` reported an error.

When you call `strtou` to convert the string to an integer, `strtoi` will allow an arbitrary amount of whitespace between the minus sign and the first digit of a string representing a negative number. If this is a problem for you, modify `strtou` to skip whitespace and then call a subservient routine to do the conversion; next, have `strtoi` call that subservient routine (which will return an illegal initial character error, if appropriate) in place of `strtou`.

The `asmMain` function demonstrates calling the `strtou` and `strtoi` functions:

```
// Listing9-14.S (cont.)
//
////////////////////////////////////////////////////////////////////
//
```

```
// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            byte    am.shadow, 64
            endl    am

            enter   am.size

// Test unsigned conversions:

            lea     x1, value1
            bl      strtou
            bcs     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr1
            lea     x1, value1
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x1, value2
            bl      strtou
            bcs     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr1
            lea     x1, value2
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x1, value3
            bl      strtou
            bcs     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr1
            lea     x1, value3
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x1, value4
            bl      strtou
            bcs     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr1
            lea     x1, value4
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf
```

```
                lea     x1, value5
                bl      strtou
                bcs     UnexpectedError

                mov     x2, x0
                lea     x0, fmtStr1
                lea     x1, value5
                mstr    x1, [sp]
                mstr    x2, [sp, #8]
                bl      printf

                lea     x1, value6
                bl      strtou
                bcs     UnexpectedError

                mov     x2, x0
                lea     x0, fmtStr1
                lea     x1, value6
                mstr    x1, [sp]
                mstr    x2, [sp, #8]
                bl      printf

                lea     x1, OFvalue
                bl      strtou
                bcc     UnexpectedError
                cmp     x0, xzr         // Nonzero for overflow
                beq     UnexpectedError

                mov     x2, x0
                lea     x0, fmtStr2
                lea     x1, OFvalue
                mstr    x1, [sp]
                mstr    x2, [sp, #8]
                bl      printf

                lea     x1, OFvalue2
                bl      strtou
                bcc     UnexpectedError
                cmp     x0, xzr         // Nonzero for overflow
                beq     UnexpectedError

                mov     x2, x0
                lea     x0, fmtStr2
                lea     x1, OFvalue2
                mstr    x1, [sp]
                mstr    x2, [sp, #8]
                bl      printf

// Test signed conversions:

                lea     x1, ivalue1
                bl      strtoi
                bcs     UnexpectedError

                mov     x2, x0
```

```
lea     x0, fmtStr3
lea     x1, ivalue1
mstr    x1, [sp]
mstr    x2, [sp, #8]
bl      printf

lea     x1, ivalue2
bl      strtoi
bcs     UnexpectedError

mov     x2, x0
lea     x0, fmtStr3
lea     x1, ivalue2
mstr    x1, [sp]
mstr    x2, [sp, #8]
bl      printf

lea     x1, ivalue3
bl      strtoi
bcs     UnexpectedError

mov     x2, x0
lea     x0, fmtStr3
lea     x1, ivalue3
mstr    x1, [sp]
mstr    x2, [sp, #8]
bl      printf

lea     x1, ivalue4
bl      strtoi
bcs     UnexpectedError

mov     x2, x0
lea     x0, fmtStr3
lea     x1, ivalue4
mstr    x1, [sp]
mstr    x2, [sp, #8]
bl      printf

lea     x1, ivalue5
bl      strtoi
bcs     UnexpectedError

mov     x2, x0
lea     x0, fmtStr3
lea     x1, ivalue5
mstr    x1, [sp]
mstr    x2, [sp, #8]
bl      printf

lea     x1, ivalue6
bl      strtoi
bcs     UnexpectedError
```

```
            mov     x2, x0
            lea     x0, fmtStr3
            lea     x1, ivalue6
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x1, OFivalue
            bl      strtoi
            bcc     UnexpectedError
            cmp     x0, xzr         // Nonzero for overflow
            beq     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr2
            lea     x1, OFivalue
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x1, OFivalue2
            bl      strtoi
            bcc     UnexpectedError
            cmp     x0, xzr         // Nonzero for overflow
            beq     UnexpectedError

            mov     x2, x0
            lea     x0, fmtStr2
            lea     x1, OFivalue2
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            b.al    allDone

UnexpectedError:
            lea     x0, unexError
            bl      printf

allDone:    leave   // Returns to caller
            endp    asmMain
```

The `asmMain` function in Listing 9-14 is a typical test program; it converts various strings appearing in the read-only data section to their corresponding integer values and displays them. It also tests a couple of overflow conditions to verify that the routines properly handle overflow.

The following is the `build` command and sample output for the program in Listing 9-14:

```
% ./build Listing9-14
% ./Listing9-14
Calling Listing9-14:
strtou: String='  1'
    value=1
```

```
strtou: String='12 '
    value=12
strtou: String=' 123 '
    value=123
strtou: String='1234'
    value=1234
strtou: String='1234567890123456789'
    value=1234567890123456789
strtou: String='18446744073709551615'
    value=18446744073709551615
Overflow: String='18446744073709551616'
    value=ffffffffffffffff
Overflow: String='99999999999999999999'
    value=ffffffffffffffff
strtoi: String='  -1'
    value=-1
strtoi: String='-12 '
    value=-12
strtoi: String=' -123 '
    value=-123
strtoi: String='-1234'
    value=-1234
strtoi: String='-1234567890123456789'
    value=-1234567890123456789
strtoi: String='-18446744073709551615'
    value=1
Overflow: String='18446744073709551616'
    value=ffffffffffffffff
Overflow: String='-18446744073709551616'
    value=ffffffffffffffff
Listing9-14 terminated
```

For an extended-precision string-to-numeric conversion, simply modify the `strtou` function to include an extended-precision accumulator, then do an extended-precision multiplication by 10 (rather than a standard multiplication).

9.3.2 Hexadecimal Strings to Numeric Form

As was the case for numeric output, hexadecimal input is the easiest numeric input routine to write. The basic algorithm for converting hexadecimal strings to numeric form is the following:

1. Initialize an accumulator value to 0.
2. For each input character that is a valid hexadecimal digit, repeat steps 3 through 6; skip down to step 7 when the character is not a valid hexadecimal digit.
3. Convert the hexadecimal character to a value in the range 0 to 15 (0h to 0Fh).

4. If the HO 4 bits of the accumulator value are nonzero, raise an exception.
5. Multiply the current value by 16 (that is, shift left 4 bits).
6. Add the converted hexadecimal digit value to the accumulator.
7. Check the current input character to ensure that it is a valid delimiter. Raise an exception if it is not.

Listing 9-15 implements this hexadecimal input routine for 64-bit values.

```
// Listing9-15.S
//
// Hexadecimal-string-to-numeric conversion

            #include    "aoaa.inc"

false       =           0
true        =           1
tab         =           9

            .section    .rodata, ""
ttlStr:     .asciz      "Listing 9-15"
fmtStr1:    .ascii      "strtoh: String='%s' "
            .asciz      "value=%llx\n"

fmtStr2:    .asciz      "Error, str='%s', x0=%lld\n"

fmtStr3:    .ascii      "Error, expected overflow: x0=%llx, "
            .asciz      "str='%s'\n"

fmtStr4:    .ascii      "Error, expected bad char: x0=%llx, "
            .asciz      "str='%s'\n"

hexStr:     .asciz      "1234567890abcdef"
hexStrOVFL: .asciz      "1234567890abcdef0"
hexStrBAD:  .asciz      "x123"

            .code
            .extern     printf

////////////////////////////////////////////////////////////
//
// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

////////////////////////////////////////////////////////////
//
```

```
// strtoh:
//
// Converts string data to a 64-bit unsigned integer
//
// Input:
//
//   X1-    Pointer to buffer containing string to convert
//
// Outputs:
//
//   X0-    Contains converted string (if success), error code
//          if an error occurs
//
//   X1-    Points at first char beyond end of hexadecimal string.
//          If error, X1's value is restored to original value.
//          Caller can check character at [X1] after a
//          successful result to see if the character following
//          the hexadecimal digits is a legal delimiter.
//
//   C-     (carry flag) Set if error occurs, clear if
//          conversion was successful. On error, X0 will
//          contain 0 (illegal initial character) or
//          -1 = 0xffffffffffffffff (overflow).

            proc    strtoh

            stp     x3, x4, [sp, #-16]!
            stp     x1, x2, [sp, #-16]!

            // This code will use the value in X3 to test
            // whether overflow will occur in X0 when
            // shifting to the left 4 bits:

            mov     x3, 0xF000000000000000
            mov     x0, xzr // Zero out accumulator.

            // 0x5f is used to convert lowercase to
            // uppercase:

            mov     x4, 0x5f

// The following loop skips over any whitespace (spaces and
// tabs) that appear at the beginning of the string:

            sub     x1, x1, #1  // Because of inc below
skipWS:     ldrb    w2, [x1, #1]!
            cmp     w2, #' '
            beq     skipWS
            cmp     w2, #tab
            beq     skipWS

            // If you don't have a hexadecimal digit at this
            // point, return an error:
```

```
            ❶ cmp     w2, #'0'    // Note: '0' < '1' < ... < '9'
              blo     badNumber
              cmp     w2, #'9'
              bls     convert
              and     x2, x2, x4  // Cheesy LC -> UC conversion
              cmp     w2, #'A'
              blo     badNumber
              cmp     w2, #'F'
              bhi     badNumber
              sub     w2, w2, #7  // Maps 41h..46h -> 3ah..3fh

              // Okay, the first digit is good. Convert the
              // string of digits to numeric form:

❷ convert:    ands    xzr, x3, x0  // See if adding in the current
              bne     overflow     // digit will cause an overflow.

              and     x2, x2, #0xf // Convert to numeric in X2.

              // Multiply 64-bit accumulator by 16 and add in
              // new digit:

            ❸ lsl     x0, x0, #4
              add     x0, x0, x2  // Never overflows

              // Move on to next character:

              ldrb    w2, [x1, #1]!
              cmp     w2, #'0'
              blo     endOfNum
              cmp     w2, #'9'
              bls     convert

              and     x2, x2, x4  // Cheesy LC -> UC conversion
              cmp     x2, #'A'
              blo     endOfNum
              cmp     x2, #'F'
              bhi     endOfNum
              sub     x2, x2, #7  // Maps 41h..46h -> 3ah..3fh
              b.al    convert

  // If you get to this point, you've successfully converted
  // the string to numeric form:

  endOfNum:

              // Because the conversion was successful, this
              // procedure leaves X1 pointing at the first
              // character beyond the converted digits.
              // Therefore, don't restore X1 from the stack.

              ldp     x3, x2, [sp], #16   // X3 holds old X1
              ldp     x3, x4, [sp], #16
              msr     nzcv, xzr   // clr c = no error
              ret
```

```
// badNumber- Drop down here if the first character in
//            the string was not a valid digit.

badNumber:  mov     x0, xzr
            b.al    errorExit

overflow:   mov     x0, #-1     // Return -1 as error on overflow.
errorExit:
            mrs     x1, nzcv    // Return error in carry flag.
            orr     x1, x1, #(1 << 29)
            msr     nzcv, x1    // Set c = error.

            ldp     x1, x2, [sp], #16
            ldp     x3, x4, [sp], #16
            ret
            endp    strtoh

////////////////////////////////////////////////////////////
//
// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            byte    am.stack, 64
            endl    am

            enter   am.size

            // Test hexadecimal conversion:

            lea     x1, hexStr
            bl      strtoh
            bcs     error

            mov     x2, x0
            lea     x1, hexStr
            lea     x0, fmtStr1
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

 // Test overflow conversion:

            lea     x1, hexStrOVFL
            bl      strtoh
            bcc     unexpected

            mov     x2, x0
            lea     x0, fmtStr2
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf
```

```
// Test bad character:

            lea     x1, hexStrBAD
            bl      strtoh
            bcc     unexp2

            mov     x2, x0
            lea     x0, fmtStr2
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            b.al    allDone

unexpected: mov     x3, x0
            lea     x0, fmtStr3
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            mstr    x3, [sp, #16]
            bl      printf
            b.al    allDone

unexp2:     mov     x3, x0
            lea     x0, fmtStr4
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            mstr    x3, [sp, #16]
            bl      printf
            b.al    allDone

error:      mov     x2, x0
            lea     x0, fmtStr2
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

allDone:    leave
            endp    asmMain
```

The `strtoh` function is similar to `strtou`, except that it tests for hexadecimal digits ❶ (rather than just decimal digits), tests the HO 4 bits to determine whether an overflow occurs ❷ (much easier than the decimal case), and multiplies by the hexadecimal radix (16) rather than by 10 ❸.

Here's the `build` command and sample output for the program in Listing 9-15:

```
% ./build Listing9-15
% ./Listing9-15
Calling Listing9-15:
strtoh: String='1234567890abcdef' value=1234567890abcdef
Error, str='1234567890abcdef0', x0=-1
Error, str='x123', x0=0
Listing9-15 terminated
```

For hexadecimal string conversions that handle numbers greater than 64 bits, you have to use an extended-precision shift left by 4 bits. Listing 9-16 demonstrates the necessary modifications to the `strtoh` function for a 128-bit conversion.

```
// Listing9-16.S
//
// 128-bit Hexadecimal-string-to-numeric conversion

            #include    "aoaa.inc"

false       =           0
true        =           1
tab         =           9

            .section    .rodata, ""
 tlStr:     .asciz      "Listing 9-16"

fmtStr1:    .asciz      "strtoh128: value=%llx%llx, String='%s'\n"

hexStr:     .asciz      "1234567890abcdeffedcba0987654321"


            .code
            .extern     printf

////////////////////////////////////////////////////////////
//
// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

////////////////////////////////////////////////////////////
//
// strtoh128
//
// Converts string data to a 128-bit unsigned integer
//
// Input:
//
//   X2-    Pointer to buffer containing string to convert
//
// Outputs:
//
//   X1:X0- Contains converted string (if success), error code
//          if an error occurs
//
//   X2-    Points at first char beyond end of hexadecimal
//          string. If error, X2's value is restored to
//          original value.
//          Caller can check character at [X2] after a
```

```
//          successful result to see if the character following
//          the hexadecimal digits is a legal delimiter.
//
//   C-     (carry flag) Set if error occurs, clear if
//          conversion was successful. On error, X0 will
//          contain 0 (illegal initial character) or
//          -1 = 0xffffffffffffffff (overflow).

            proc    strtoh128

            stp     x4, x5, [sp, #-16]!
            stp     x2, x3, [sp, #-16]!

            // This code will use the value in X4 to test
            // whether overflow will occur in X1 when
            // shifting to the left 4 bits:

            mov     x4, 0xF000000000000000
            mov     x0, xzr // Zero out L0 accumulator.
            mov     x1, xzr // Zero out H0 accumulator.

            // 0x5f is used to convert lowercase to
            // uppercase:

            mov     x5, 0x5f

// The following loop skips over any whitespace (spaces and
// tabs) that appear at the beginning of the string:

            sub     x2, x2, #1 // Because of inc below
skipWS:     ldrb    w3, [x2, #1]!
            cmp     w3, #' '
            beq     skipWS
            cmp     w3, #tab
            beq     skipWS

            // If you don't have a hexadecimal digit at this
            // point, return an error:

            cmp     w3, #'0'   // Note: '0' < '1' < ... < '9'
            blo     badNumber
            cmp     w3, #'9'
            bls     convert
            and     x3, x3, x5 // Cheesy LC -> UC conversion
            cmp     w3, #'A'
            blo     badNumber
            cmp     w3, #'F'
            bhi     badNumber
            sub     w3, w3, #7 // Maps 41h..46h -> 3ah..3fh

            // Okay, the first digit is good. Convert the
            // string of digits to numeric form:

convert:    ands    xzr, x4, x1  // See whether adding in the current
            bne     overflow     // digit will cause an overflow.
```

```
            and     x3, x3, #0xf // Convert to numeric in X3.

            // Multiply 128-bit accumulator by 16 and add in
            // new digit (128-bit extended-precision shift
            // by 4 bits):

❶           lsl     x1, x1, #4  // 128 bits shifted left 4 bits
            orr     x1, x1, x0, lsr #60
            lsl     x0, x0, #4
            add     x0, x0, x3  // Never overflows

            // Move on to next character:

            ldrb    w3, [x2, #1]!
            cmp     w3, #'0'
            blo     endOfNum
            cmp     w3, #'9'
            bls     convert

            and     x3, x3, x5  // Cheesy LC -> UC conversion
            cmp     x3, #'A'
            blo     endOfNum
            cmp     x3, #'F'
            bhi     endOfNum
            sub     x3, x3, #7 // Maps 41h..46h -> 3ah..3fh
            b.al    convert

// If you get to this point, you've successfully converted
// the string to numeric form:

endOfNum:

            // Because the conversion was successful, this
            // procedure leaves X2 pointing at the first
            // character beyond the converted digits.
            // Therefore, we don't restore X2 from the stack.

            ldp     x4, x3, [sp], #16   // X4 holds old X2.
            ldp     x4, x5, [sp], #16
            msr     nzcv, xzr   // clr c = no error

            ret

// badNumber- Drop down here if the first character in
//            the string was not a valid digit.

badNumber:  mov     x0, xzr
            b.al    errorExit

overflow:   mov     x0, #-1     // Return -1 as error on overflow.
errorExit:
```

```
            mrs     x1, nzcv    // Return error in carry flag.
            orr     x1, x1, #(1 << 29)
            msr     nzcv, x1    // Set c = error.
            ldp     x2, x3, [sp], #16
            ldp     x4, x5, [sp], #16
            ret
            endp    strtoh128

////////////////////////////////////////////////////////
//
// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            byte    am.stack, 64
            endl    am

            enter   am.size

// Test hexadecimal conversion:

            lea     x2, hexStr
            bl      strtoh128

            lea     x3, hexStr
            mov     x2, x0
            lea     x0, fmtStr1
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            mstr    x3, [sp, #16]
            bl      printf

allDone:    leave
            endp    asmMain
```

This code works similarly to that in Listing 9-15. The main difference is the 128-bit shift left by 4 bits ❶ in Listing 9-16. The code shifts X0 to the right 60 bits, then ORs this into X1 after shifting it to the left 4 bits, which shifts 4 bits from X0 into X1.

Here's the `build` command and sample output for Listing 9-16:

```
% ./build Listing9-16
% ./Listing9-16
Calling Listing9-16:
strtoh128: value=1234567890abcdeffedcba0987654321, String='1234567890abcdeffedcba0987654321'
Listing9-16 terminated
```

The hexadecimal-string-to-numeric function worked as expected.

9.3.3 String to Floating-Point

Converting a string of characters representing a floating-point number to the 64-bit `double` format is slightly easier than the double-to-string conversion that appeared earlier in this chapter. Because decimal conversion (with no exponent) is a subset of the more general scientific notation conversion, if you can handle scientific notation, you get decimal conversion for free. Beyond that, the basic algorithm is to convert the mantissa characters to an integer form in order to convert to floating-point, then read the (optional) exponent and adjust the `double` exponent accordingly. The algorithm for the conversion is the following:

1. Begin by stripping away any leading space or tab characters (and any other delimiters).
2. Check for a leading plus (+) or minus (-) sign character. Skip it if one is present. Set a sign flag to true if the number is negative (false if nonnegative).
3. Initialize an exponent value to –16. The algorithm will create an integer value from the mantissa digits in the string. As double-precision floats support a maximum of 16 significant digits, initializing the exponent to –16 accounts for this.
4. Initialize a significant-digit-counter variable that counts the number of significant digits processed thus far to 16.
5. If the number begins with any leading 0s, skip over them (do not change the exponent or significant digit counters for leading 0s to the left of the decimal point).
6. If the scan encounters a decimal point after processing any leading 0s, go to step 11; otherwise, fall through to step 7.
7. For each nonzero digit to the left of the decimal point, if the significant digit counter is not 0, multiply the integer accumulator by 10 and add in the numeric equivalent of the digit. This is the standard integer conversion. (If the significant digit counter is 0, the algorithm has already processed 16 significant digits and will ignore any additional digits, since the double format cannot represent more than 16 significant digits.)
8. For each digit to the left of the decimal point, increment the exponent value (originally initialized to –16) by 1.
9. If the significant digit counter is not 0, decrement the significant digit counter (which will also provide the index into the digit string array).
10. If the first nondigit encountered is not a decimal point, skip to step 14.
11. Skip over the decimal point character.
12. For each digit encountered to the right of the decimal point, continue adding the digits to the integer accumulator as long as the significant digit counter is not 0. If the significant digit counter is greater than 0, decrement it. Also decrement the exponent value.

13. If the algorithm hasn't encountered at least one decimal digit by this point, report an illegal character exception and return.
14. If the current character is not `e` or `E`, go to step 20. Otherwise, skip over the `e` or `E` character and continue with step 15. (Note that some string formats also allow `d` or `D` to denote a double-precision value. You can also choose to allow this, and possibly check the range of the value if the algorithm encounters `e` or `E` versus `d` or `D`.)
15. If the next character is `+` or `-`, skip over it. Set a flag to true if the sign character is `-`; set it to false otherwise (note that this exponent sign flag is different from the mantissa sign flag set earlier in this algorithm).
16. If the next character is not a decimal digit, report an error.
17. Convert the string of digits starting with the current decimal digit character to an integer.
18. Add the converted integer to the exponent value that was initialized to –16 at the start of this algorithm.
19. If the exponent value is outside the range –324 to +308, report an out-of-range exception.
20. Convert the mantissa, which is currently an integer, to a floating-point value.
21. Take the absolute value of the exponent, preserving the exponent's sign. This value will be 9 bits or less.
22. If the exponent was positive, then for each set bit in the exponent, multiply the current mantissa value by 10 raised to the power specified by that bit's position. For example, if bits 4, 2, and 1 are set, multiply the mantissa value by 10^{16}, 10^{4}, and 10^{2}.
23. If the exponent was negative, then for each set bit in the exponent, divide the current mantissa value by 10 raised to the power specified by that bit's position. For example, if bits 4, 3, and 2 are set, divide the mantissa value by 10^{16}, 10^{8}, and 10^{4} (starting with the larger values and working your way down).
24. If the mantissa is negative (the first sign flag set at the beginning of the algorithm), negate the floating-point number.

Listing 9-17 provides an implementation of this algorithm, explained section by section. The first part is typical for the sample programs in this book, containing some constant declarations, static data, and the `getTitle` function.

```
// Listing9-17.S
//
// Real string to floating-point conversion

            #include    "aoaa.inc"

false       =           0
true        =           1
tab         =           9
```

```
                .section    .rodata, ""
ttlStr:         .asciz      "Listing 9-17"
fmtStr1:        .asciz      "strToR64: str='%s', value=%e\n"
errFmtStr:      .asciz      "strToR64 error, code=%ld\n"

❶ fStr1a:       .asciz      " 1.234e56"
fStr1b:         .asciz      "\t-1.234e+56"
fStr1c:         .asciz      "1.234e-56"
fStr1d:         .asciz      "-1.234e-56"
fStr2a:         .asciz      "1.23"
fStr2b:         .asciz      "-1.23"
fStr2c:         .asciz      "001.23"
fStr2d:         .asciz      "-001.23"
fStr3a:         .asciz      "1"
fStr3b:         .asciz      "-1"
fStr4a:         .asciz      "0.1"
fStr4b:         .asciz      "-0.1"
fStr4c:         .asciz      "0000000.1"
fStr4d:         .asciz      "-0000000.1"
fStr4e:         .asciz      "0.1000000"
fStr4f:         .asciz      "-0.1000000"
fStr4g:         .asciz      "0.0000001"
fStr4h:         .asciz      "-0.0000001"
fStr4i:         .asciz      ".1"
fStr4j:         .asciz      "-.1"
fStr5a:         .asciz      "123456"
fStr5b:         .asciz      "12345678901234567890"
fStr5c:         .asciz      "0"
fStr5d:         .asciz      "1."
fStr6a:         .asciz      "0.000000000000000000001"

              ❷ .align      3
values:         .dword      fStr1a, fStr1b, fStr1c, fStr1d
                .dword      fStr2a, fStr2b, fStr2c, fStr2d
                .dword      fStr3a, fStr3b
                .dword      fStr4a, fStr4b, fStr4c, fStr4d
                .dword      fStr4e, fStr4f, fStr4g, fStr4h
                .dword      fStr4i, fStr4j
                .dword      fStr5a, fStr5b, fStr5c, fStr5d
                .dword      fStr6a
                .dword      0

❸ PotTbl:       .double     1.0e+256
                .double     1.0e+128
                .double     1.0e+64
                .double     1.0e+32
                .double     1.0e+16
                .double     1.0e+8
                .double     1.0e+4
                .double     1.0e+2
                .double     1.0e+1
                .double     1.0e+0

                .data
r8Val:          .double     0.0
```

```
            .code
            .extern     printf

//////////////////////////////////////////////////////////
//
// Return program title to C++ program:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle
```

The read-only section contains various test strings that this program will convert into floating-point values ❶. These test strings were carefully chosen to test most of the (successful) paths through the `strToR64` function. To reduce the size of the main program, Listing 9-17 processes these strings in a loop. The array of pointers ❷ points at each of the test strings, with a NULL pointer (`0`) marking the end of the list. The main program will iterate through these pointers in a loop to test the input strings.

The `PotTbl` (powers-of-10 table) array ❸ contains various powers of 10. The `strToR64` function uses this table to convert a decimal exponent (in integer format) to an appropriate power of 10:

```
// Listing9-17.S (cont.)
//
// strToR64
//
// On entry:
//
//  X0- Points at a string of characters that represent a
//      floating-point value
//
// On return:
//
//  D0- Converted result
//  X0- On return, X0 points at the first character this
//      routine couldn't convert (if no error).
//
//  C-  Carry flag is clear if no error, set if error.
//      X7 is preserved if an error, X1 contains an
//      error code if an error occurs (else X1 is
//      preserved).

            proc    strToR64

            locals  sr
            qword   sr.x1x2
            qword   sr.x3x4
            qword   sr.x5x6
            qword   sr.x7x0
            dword   sr.d1
            byte    sr.stack, 64    // Not really needed, but ...
```

```
            endl    sr

            enter   sr.size

// Defines to give registers more
// meaningful names:

❶ #define mant    x1      // Mantissa value
  #define sigDig  x2      // Mantissa significant digits
  #define expAcc  x2      // Exponent accumulator
  #define sign    w3      // Mantissa sign
  #define fpExp   x4      // Exponent
  #define expSign w5      // Exponent sign
  #define ch      w6      // Current character
  #define xch     x6      // Current character (64 bits)
  #define ten     x7      // The value 10

            // Preserve the registers this
            // code modifies:

          ❷ stp     x1, x2, [fp, #sr.x1x2]
            stp     x3, x4, [fp, #sr.x3x4]
            stp     x5, x6, [fp, #sr.x5x6]
            stp     x7, x0, [fp, #sr.x7x0]
            str     d1,     [fp, #sr.d1  ]

            // Useful initialization:

            mov     fpExp, xzr      // X3 Decimal exponent value
            mov     mant, xzr       // X0 Mantissa value
            mov     sign, wzr       // W2 Assume nonnegative.

            // Initialize sigDig with 16, the number of
            // significant digits left to process.

            mov     sigDig, #16     // X1

            // Verify that X0 is not NULL.

            cmp     x0, xzr
            beq     refNULL
```

The `strToR64` function uses `#define` statements ❶ to create meaningful, more readable names for the local variables it maintains in various registers.

Although this function uses only registers X0 through X7 and D1 (which are all volatile in the ARM ABI), this function preserves all the registers it modifies ❷. In assembly language, it's always good programming style to preserve modified registers. This code does not preserve X0 (assuming a successful conversion) because it returns X0 pointing at the end of the (successfully) converted string as a function result. Note that this code returns the main function result in D0.

After function initialization, the `strToR64` function begins by skipping all whitespace (spaces and tabs) at the beginning of the string:

```
// Listing9-17.S (cont.)

            sub     x0, x0, #1      // Will inc'd in loop
whileWSLoop:
            ldrb    ch, [x0, #1]!   // W5
            cmp     ch, #' '
            beq     whileWSLoop
            cmp     ch, #tab
            beq     whileWSLoop
```

This code exits with `ch` (W6) containing the first non-whitespace character and X0 pointing at that character in memory.

Immediately after any whitespace characters, the string may optionally contain a single + or -character. This code skips either of these characters (if present) and sets the mantissa sign flag (`sign`) to 1 if a - character is present:

```
// Listing9-17.S (cont.)

            // Check for + or -

            cmp     ch, #'+'
            beq     skipSign

            cmp     ch, #'-'
            cinc    sign, sign, eq  // W2
            bne     noSign

skipSign:   ldrb    ch, [x0, #1]!   // Skip '-'
noSign:
```

Immediately after a sign character (or if there isn't an optional sign character), the string must contain a decimal digit character or a decimal point. This code tests for one of these two conditions and reports a conversion error if the condition fails:

```
// Listing9-17.S (cont.)

          ❶ sub     ch, ch, #'0'    // Quick test for '0' to '9'
            cmp     ch, #9
            bls     scanDigits      // Branch if '0' to '9'

          ❷ cmp     ch, #'.'-'0'    // Check for '.'
            bne     convError

            // If the first character is a decimal point,
            // the second character needs to be a
            // decimal digit.

          ❸ ldrb    ch, [x0, #1]!   // W5 Skip period.
            cmp     ch, #'0'
            blo     convError
```

```
            cmp     ch, #'9'
            bhi     convError
            b.al    whileDigit2
```

This code uses a common trick to compare for a character in the range `'0'` through `'9'`. It subtracts the ASCII code for `'0'` from the character ❶. If the character was in the range `'0'` to `'9'`, this translates its value to the range 0 to 9. A single *unsigned* comparison against the value 9 tells us whether the character value was in the range `'0'` to `'9'`. If so, this code transfers control to the code that will process digits to the left of the decimal point.

Because the code has subtracted `'0'` from the character's ASCII code, it cannot simply compare the character against a period. The `cmp ch, #'.'-'0'` instruction correctly compares the character against a period by subtracting the character code for `'0'` from `'.'` ❷. If the character was a period, the code will verify that the following character is also a digit ❸.

Next, the code starting at `scanDigits` processes the mantissa digits to the left of the decimal point (if present):

```
// Listing9-17.S (cont.)
//
// Scan for digits at the beginning of the number:

scanDigits: mov     ten, #10        // X7 used to multiply by 10
            add     ch, ch, #'0'    // Restore character.
 whileADigit:
            sub     ch, ch, #'0'    // Quick way to test for
            cmp     ch, #10         // a range and convert
            bhs     notDigit        // to an integer

            // Ignore any leading 0s in the number.
            // You have a leading '0' if the mantissa is 0
            // and the current character is '0'.

          ❶ cmp     mant, xzr       // Ignore leading 0s.
            ccmp    ch, #0, #0, eq
            beq     Beyond16

            // Each digit to the left of the decimal
            // point increases the number by an
            // additional power of 10. Deal with that
            // here.

          ❷ add     fpExp, fpExp, #1

            // Save all the significant digits but ignore
            // any digits beyond the 16th digit.

          ❸ cmp     sigDig, xzr     // X1
            beq     Beyond16

            // Count down the number of significant digits.

            sub     sigDig, sigDig, #1
```

```
            // Multiply the accumulator (mant) by 10 and
            // add in the current digit. Note that ch
            // has already been converted to an integer.

          ❹ madd    mant, mant, ten, xch    // X0, X6, X5

            // Because you multiplied the exponent by 10,
            // you need to undo the increment of fpExp.

          ❺ sub     fpExp, fpExp, #1

Beyond16:   ldrb    ch, [x0, #1]!   // Get next char.
            b.al    whileADigit
```

This code skips over leading 0s by noting that if the mantissa value is 0 and the current character is `'0'`, it's a leading 0 ❶. For each mantissa digit the code fetches, it adjusts the mantissa value by multiplying the mantissa by 10 and adding in the numeric equivalent of the digit ❹. However, if the loop processes more than 16 significant digits ❸, it does not add in the character to the `mant` accumulator (because double-precision objects support a maximum of 16 significant digits). If the input string exceeds 16 significant digits, the code increments the `fpExp` variable ❷ to track the eventual exponent of the number. The code undoes this increment ❺ if the mantissa was multiplied by 10 (in which case the exponent does not need to be incremented).

The next section of code handles the digits after a decimal point:

```
// Listing9-17.S (cont.)
//
// If you encountered a nondigit character,
// check for a decimal point:

notDigit:
            cmp     ch, #'.'-'0'    // See if a decimal point.
            bne     whileDigit2

// Okay, process any digits to the right of the decimal point.
// If this code falls through from the above, it skips the
// decimal point.

getNextChar:
            ldrb    ch, [x0, #1]!   // Get the next character.
whileDigit2:
            sub     ch, ch, #'0'
            cmp     ch, #10
            bhs     noDigit2

            // Ignore digits after the 16th significant
            // digit but don't count leading 0s
            // as significant digits:

          ❶ cmp     mant, xzr           // Ignore leading 0s.
            ccmp    ch, wzr, #0, eq
```

```
            ccmp    sigDig, xzr, #0, eq  // X2
            beq     getNextChar

            // Each digit to the right of the decimal point decreases
            // the number by an additional power of 10. Deal with
            // that here.

❷           sub     fpExp, fpExp, #1

            // Count down the number of significant digits:

            sub     sigDig, sigDig, #1

            // Multiply the accumulator (mant) by 10 and
            // add in the current digit. Note that ch
            // has already been converted to an integer:

            Madd    mant, mant, ten, xch    // X1, X7, X6
            b.al    getNextChar
```

The code is similar to the digits to the left, except that it decrements the running exponent value for each digit ❷. This is because the mantissa is being maintained as an integer, and the code continues to insert the fractional digits into the mantissa by multiplying by 10 and adding in the digit's value. Should the total number of significant digits exceed 16 (not including leading 0s ❶), this function ignores any further digits.

Next up is processing the string's optional exponent:

```
// Listing9-17.S (cont.)

❶ noDigit2:
            mov     expSign, wzr    // W5 Initialize exp sign.
            mov     expAcc, xzr     // X2 Initialize exponent.
            cmp     ch, #'e'-'0'
            beq     hasExponent
            cmp     ch, #'E'-'0'
            bne     noExponent

❷ hasExponent:
            ldrb    ch, [x0, #1]!           // Skip the "E".
            cmp     ch, #'-'                // W6
            cinc    expSign, expSign, eq    // W5
            beq     doNextChar_2
            cmp     ch, #'+'
            bne     getExponent

doNextChar_2:
            ldrb    ch, [x0, #1]!   // Skip '+' or '-'.

// Okay, you're past the "E" and the optional sign at this
// point. You must have at least one decimal digit.
```

```
❸ getExponent:
            sub     ch, ch, #'0'    // W5
            cmp     ch, #10
            bhs     convError

            mov     expAcc, xzr     // Compute exponent value in X2.
ExpLoop:    ldrb    ch, [x0], #1
            sub     ch, ch, #'0'
            cmp     ch, #10
            bhs     ExpDone

            madd    expAcc, expAcc, ten, xch    // X2, X7, X6
            b.al    ExpLoop

// If the exponent was negative, negate your computed result:

❹ ExpDone:
            cmp     expSign, #false // W5
            beq     noNegExp

            neg     expAcc, expAcc  // X2

noNegExp:

// Add in the computed decimal exponent with the exponent
// accumulator:

          ❺ add     fpExp, fpExp, expAcc    // X4, X2

noExponent:

// Verify that the exponent is from -324 to +308 (which
// is the maximum dynamic range for a 64-bit FP value):

          ❻ mov     x5, #308        // Reuse expSign here.
            cmp     fpExp, x5
            bgt     voor            // Value out of range
            mov     x5, #-324
            cmp     fpExp, x5
            blt     voor
          ❼ ucvtf   d0, mant        // X1
```

This code first checks for an `e` or `E` character denoting the start of an exponent ❶. If the string has an exponent, the code checks for an optional sign character ❷. If a - character is present, the code sets `expSign` to 1 (default is 0) to specify a negative exponent.

After processing the exponent sign, the code expects decimal digits ❸ and converts these digits to an integer (held in the `expAcc` variable). If `expSign` is true (nonzero), the code negates the value in `expAcc` ❹. The exponent code then adds `expAcc` to the exponent value obtained when processing the mantissa digits to obtain the actual exponent value ❺.

Finally, the code checks the exponent to verify it's in the range –324 to +308 ❻. This is the maximum dynamic range of a 64-bit double-precision

floating-point value. If the exponent is out of this range, the code returns a value-out-of-range error.

At this point, the code has completely processed the string data, and the X0 register points at the first byte in memory that is not part of the floating-point value. To convert the mantissa and exponent values from integers into a double-precision value, first convert the mantissa value (in `mant`) to a floating-point value by using the `ucvtf` instruction ❼.

Next, processing the exponent is somewhat tricky. The `fpExp` variable contains the decimal exponent, but this is an integer value representing a power of 10. You must multiply the value in D0 (the mantissa) by 10^{fpExp}, but unfortunately, the ARM instruction set does not provide an instruction that computes 10 raised to some integer power. You'll have to write your own code to do this:

```
// Listing9-17.S (cont.)
//
// Okay, you have the mantissa into D0. Now multiply
// D0 by 10 raised to the value of the computed exponent
// (currently in fpExp).
//
// This code uses power-of-10 tables to help make the
// computation a little more accurate.
//
// You want to determine which power of 10 is just less than the
// value of our exponent. The powers of 10 you are checking are
// 10**256, 10**128, 10**64, 10**32, and so on. A slick way to
// check is by shifting the bits in the exponent
// to the left. Bit #8 is the 256 bit, so if this bit is set,
// your exponent is >= 10**256. If not, check the next bit down
// to see if your exponent >= 10**128, and so on.

            mov     x1, -8      // Initial index into power-of-10 table
            cmp     fpExp, xzr  // X4
            bpl     positiveExponent

          ❶ // Handle negative exponents here:

            neg     fpExp, fpExp
            lsl     fpExp, fpExp, #55   // Bits 0..8 -> 55..63
            lea     x6, PotTbl
❷ whileExpNE0:
            add     x1, x1, #8          // Next index into PotTbl.
            adds    fpExp, fpExp, fpExp // (LSL) Need current POT?
            bcc     testExp0

            ldr     d1, [x6, x1]
            fdiv    d0, d0, d1

 testExp0:  cmp     fpExp, xzr
            bne     whileExpNE0
            b.al    doMantissaSign
```

```
    // Handle positive exponents here.

❸ positiveExponent:
            lea     x6, PotTbl
            lsl     fpExp, fpExp, #55       // Bits 0..8 -> 55..63
            b.al    testExpis0_2

    whileExpNE0_2:
            add     x1, x1, #8
            adds    fpExp, fpExp, fpExp     // (LSL)
            bcc     testExpis0_2

            ldr     d1, [x6, x1]
            fmul    d0, d0, d1

    testExpis0_2:
            cmp     fpExp, xzr
            bne     whileExpNE0_2
```

This code uses two nearly identical sections of code to handle negative ❶ and positive ❸ exponents. The difference between the two pieces of code is the choice of an `fdiv` instruction (for negative exponents) or an `fmul` instruction (for positive exponents). Each section contains a loop ❷ that steps through each entry of the `PotTbl` (powers-of-10) table. The exponent is a 9-bit value, as the maximum unsigned exponent value is 324, which fits in 9 bits or fewer.

For each set bit in this integer, the code must multiply the floating-point result by the corresponding power of 10 from `PotTbl`. For example, if bit 9 is set, multiply or divide the mantissa by 10^{256} (the first entry in `PotTbl`); if bit 8 is set, multiply or divide the mantissa by 10^{128} (the second entry in `PotTbl`), . . . ; if bit 0 is set, multiply or divide the mantissa by 10^{0} (the last entry in `PotTbl`). The two loops in the code accomplish this by moving the 9 bits into the HO positions of `fpExp`, then shifting the bits out one at a time and doing the multiplication (for positive exponents) or division (for negative exponents) if the carry flag is set, using successive entries from `PotTbl`.

Next, the code negates the value if it was negative (the flag is held in the sign variable) and returns the floating-point value to the caller in the D0 register:

```
    // Listing9-17.S (cont.)

    doMantissaSign:
            cmp     sign, #false            // W3
            beq     mantNotNegative

            fneg    d0, d0

    // Successful return here. Note: does not restore X0
    // on successful conversion.
```

```
mantNotNegative:
            msr     nzcv, xzr   // clr c = no error
            ldp     x1, x2, [fp, #sr.x1x2]
            ldp     x3, x4, [fp, #sr.x3x4]
            ldp     x5, x6, [fp, #sr.x5x6]
            ldr     x7,     [fp, #sr.x7x0]
            ldr     d1,     [fp, #sr.d1  ]
            leave
```

On a successful conversion, this function returns X0 pointing at the first character beyond the floating-point string. This code does not restore X0 to its original value on a successful conversion.

The last part of the `strToR64` function is the error-handling code:

```
// Listing9-17.S (cont.)
//
// Error returns down here. Returns error code in X0:

refNULL:    mov     x1, #-3
            b.al    ErrorExit

convError:  mov     x1, #-2
            b.al    ErrorExit

voor:       mov     x1, #-1 // Value out of range
            b.al    ErrorExit

illChar:    mov     x1, #-4

// Note: on error, this code restores X0.

ErrorExit:
            str     x1, [fp, #sr.x1x2]  // Return error code in X1.
            mrs     x1, nzcv            // Return error in carry flag.
            orr     x1, x1, #(1 << 29)
            msr     nzcv, x1            // Set c = error.
            ldp     x1, x2, [fp, #sr.x1x2]
            ldp     x3, x4, [fp, #sr.x3x4]
            ldp     x5, x6, [fp, #sr.x5x6]
            ldp     x7, x0, [fp, #sr.x7x0]
            ldr     d1,     [fp, #sr.d1  ]
            leave

            endp    strToR64
```

Each error returns a special error code in X1. So this code does not restore X1 upon return. Unlike the successful return, the error return code will restore X0 to its original value.

Finally, the `asmMain` function consists of a loop that processes each of the strings by using the pointers found in the `values` array. It simply steps through each pointer, passing it along to `strToR64`, until it encounters a NULL (0) value:

```
// Listing9-17.S (cont.)

// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            dword   am.x20
            byte    stack, 64
            endl    am

            enter   am.size
            str     x20, [fp, #am.x20]

// Test floating-point conversion:

            lea     x20, values
ValuesLp:   ldr     x0, [x20]
            cmp     x0, xzr
            beq     allDone
            bl      strToR64

            lea     x0, fmtStr1
            ldr     x1, [x20]
            mstr    x1, [sp]
            mstr    d0, [sp, #8]
            bl      printf
            add     x20, x20, #8
            b.al    ValuesLp

allDone:    ldr     x20, [fp, #am.x20]
            leave
            endp    asmMain
```

Here's the `build` command and sample output for Listing 9-17:

```
% ./build Listing9-17
% ./Listing9-17
Calling Listing9-17:
strToR64: str=' 1.234e56', value=1.234000e+56
strToR64: str='    -1.234e+56', value=-1.234000e+56
strToR64: str='1.234e-56', value=1.234000e-56
strToR64: str='-1.234e-56', value=-1.234000e-56
strToR64: str='1.23', value=1.230000e+00
strToR64: str='-1.23', value=-1.230000e+00
strToR64: str='001.23', value=1.230000e+00
strToR64: str='-001.23', value=-1.230000e+00
strToR64: str='1', value=1.000000e+00
strToR64: str='-1', value=-1.000000e+00
strToR64: str='0.1', value=1.000000e-01
strToR64: str='-0.1', value=-1.000000e-01
strToR64: str='0000000.1', value=1.000000e-01
strToR64: str='-0000000.1', value=-1.000000e-01
strToR64: str='0.1000000', value=1.000000e-01
```

```
strToR64: str='-0.1000000', value=-1.000000e-01
strToR64: str='0.0000001', value=1.000000e-07
strToR64: str='-0.0000001', value=-1.000000e-07
strToR64: str='.1', value=1.000000e-01
strToR64: str='-.1', value=-1.000000e-01
strToR64: str='123456', value=1.234560e+05
strToR64: str='12345678901234567890', value=1.234568e+19
strToR64: str='0', value=0.000000e+00
strToR64: str='1.', value=1.000000e+00
strToR64: str='0.000000000000000000001', value=1.000000e-17
Listing9-17 terminated
```

It would be interesting to modify the real-to-string and string-to-real programs to perform a "round-trip" conversion from real to string to real, to see whether you get roughly the same result back that you put in. (Because of rounding and truncation errors, you won't always get the same exact value back, but it should be close.) I will leave it up to you to try this out.

9.4 Other Numeric Conversions

This chapter has presented the more common numeric conversion algorithms: decimal integer, hexadecimal integer, and floating-point. Other conversions are sometimes useful. For example, some applications might need octal (base-8) conversions or conversions in an arbitrary base. For bases 2 through 9, the algorithm is virtually the same as for decimal integer conversions, except that rather than dividing by 10 (and taking the remainder), you divide by the desired base. Indeed, it would be fairly simple to write a generic function to which you pass the radix (base) to get the appropriate conversion.

Of course, base-2 output is nearly trivial because the ARM CPU stores values internally in binary. All you need do is shift bits out of the number (into the carry flag) and output a `0` or `1` based on the state of the carry. Base-4 and base-8 conversions are also fairly simple, working with groups of 2 or 3 bits (respectively).

Some floating-point formats do not follow the IEEE standard. To handle these cases, write a function that converts such formats to the IEEE form, if possible, then use the examples from this chapter to convert between floating-point and string. If you need to work with such formats directly, the algorithms in this chapter should prove sufficiently general and easy to modify for your use.

9.5 Moving On

This long chapter covered two main topics: converting numeric values to strings and converting strings to numeric values. For the former, this chapter covered numeric-to-hexadecimal conversion (bytes, hwords, words, dwords, and qwords), numeric-to-unsigned decimal conversion (64- and 128-bit), and numeric-to-signed decimal conversion (64- and 128-bit). It

also discussed formatted conversion for controlling the output format when doing numeric-to-string conversions, and formatted floating-point-to-string conversions for decimal and exponential formats, as well as computing the number of print positions a conversion requires.

While discussing string-to-numeric conversions, this chapter covered converting unsigned decimal strings to numeric forms, signed decimal strings to numeric forms, hexadecimal strings to numeric forms, and floating-point strings to double-precision numeric forms. Finally, the chapter briefly discussed other possible numeric output formats.

Although this book will continue to use the C `printf()` function for formatted output, you can use the procedures in this chapter to avoid relying on C when writing your own assembly code. These procedures also form the basis for an assembly language library you can use to simplify writing assembly code.

9.6 For More Information

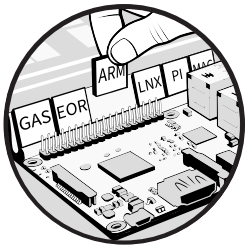
- Donald Knuth's *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd edition (Addison-Wesley Professional, 1997) contains lots of useful information about decimal arithmetic and extended-precision arithmetic, though the text is generic and doesn't describe how to do this in ARM assembly language.
- For more information on division via multiplication by a reciprocal, see the University of Iowa tutorial at *http://homepage.cs.uiowa.edu/~jones/bcd/divide.html*.

TEST YOURSELF

1. How many hexadecimal digits will `hwtoStr` produce?
2. Explain how to use `qToStr` to write a 128-bit hexadecimal output routine.
3. How do you write a signed decimal-to-string conversion if you're given a function that does an unsigned decimal-to-string conversion?
4. What are the parameters for the `u64toSizeStr` function?
5. What string will `u64toSizeStr` produce if the number requires more print positions than specified by the minimum field-width parameter?
6. What are the parameters for the `r64ToStr` function?
7. What string will `r64ToStr` produce if the output won't fit in the string size specified by the `fWidth` argument?
8. What are the arguments to the `e64ToStr` function?
9. What is a delimiter character?
10. What are two possible errors that could occur during a string-to-numeric conversion?

10

TABLE LOOKUPS



In the early days of assembly language programming, replacing expensive computations with table lookups was a common way to improve program performance. Today, memory speeds in modern systems limit the performance gains that can be obtained by using table lookups. However, for very complex calculations, this is still a viable technique for writing high-performance code.

This chapter discusses how to use table lookups to speed up or reduce the complexity of computations, demonstrating the space and speed tradeoffs involved.

10.1 Using Tables in Assembly Language

To an assembly language programmer, a *table* is an array containing initialized values that do not change after they're created. In assembly language,

you can use tables for a variety of purposes: computing functions, controlling program flow, or simply looking up data. In general, tables provide a fast mechanism for performing an operation, at the expense of space in your program (the extra space holds the tabular data).

In this section, we'll explore some of the many possible uses of tables in an assembly language program. Keep in mind that because tables typically contain initialized data that does not change during program execution, the `.section .rodata, ""` section is a good place to put your table objects.

10.1.1 Function Computation via Table Lookup

A simple-looking HLL arithmetic expression can be equivalent to a considerable amount of ARM assembly language code and may therefore be expensive to compute. Assembly language programmers often precompute many values and use a table lookup of those values to speed up their programs, which is easier and often more efficient.

Consider the following Pascal statement:

```
if (character >= 'a') and (character <= 'z') then
    character := chr(ord(character) - 32);
```

This `if` statement converts the character variable's value from lowercase to uppercase if the character is in the range `a` to `z`. Comparable assembly code requires a total of seven machine instructions, as follows:

```
    mov  w1, #'z'
    ldrb w0, [fp, #character]  // Assume "character" is local.
    cmp  w0, #'a'
  ❶ ccmp w0, w1, #0b0010, hs
    bhi  notLower
  ❷ eor  w0, w0, #0x20
notLower:
    strb w0, [fp, #character]
```

The NZCV constant 0b0010 sets the carry flag and clears the 0 so that the branch will be taken if W0 is less than `'a'` (if W0 is less than `'a'`, the carry is set and the zero flag is clear, which is "higher or same" without the same component, so just higher) ❶. Note that the conditional compare instruction allows only 5-bit immediate constants; this is why the code loads the character constant `'z'` into W1 and conditionally compares against W1.

The usual method for converting lowercase to uppercase is to clear bit 5 of the ASCII character code. However, `and w0, w0, #0x5F` is not a legal instruction because 0x5F is not a legal logical constant. This code uses the `eor` (exclusive-OR) instruction to invert bit 5 ❷. Because this bit is guaranteed to be set at this point (bit 5 is set for all lowercase characters), the `eor` instruction will clear this bit.

The lookup table solution uses only four instructions:

```
lea  x1, xlatTbl
ldrb w0, [fp, #character]
```

```
ldrb w0, [x1, w0, uxt2 #0]
strb w0, [fp, #character]
```

The conversion logic is completely buried in the lookup table (`xlatTbl`). This is a 256-byte array; each index contains the index value (element 0 contains the value 0, element 1 contains the value 1, and so on) except for the indices corresponding to the ASCII codes for the lowercase characters (indices 97 through 122). Those particular array elements contain the ASCII codes for the uppercase characters (values 65 through 90).

Note that if you can guarantee that you'll load only 7-bit ASCII characters into this code, you can get by with a 128-byte (rather than a 256-byte) array.

Here's a typical (128-byte) lookup table that converts lowercase characters to uppercase:

```
xlatTbl:    .byte       0,1,2,3,4,5,6,7
            .byte       8,9,10,11,12,13,14,15
            .byte       16,17,18,19,20,21,22,23
            .byte       24,25,26,27,28,29,30,31
            .byte       32,33,34,35,36,37,38,39
            .byte       40,41,42,43,44,45,46,47
            .byte       48,49,50,51,52,53,54,55
            .byte       56,57,58,59,60,61,62,63
            .byte       64
            .ascii      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            .byte       91,92,93,94,95,96
            .ascii      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            .byte       123,124,125,126,127
```

If you want a full 256-byte table, elements 128 through 255 would contain the values 128 through 255.

The `ldrb w0, [x1, w0, uxtw #0]` instruction loads W0 with the byte at the index specified by the (original) value held in W0, assuming X1 holds the address of `xlatTbl`. If W0 holds a non-lowercase character code, that index into the table will load the same value into W0 (so this instruction does not change W0's value if it is not a lowercase letter). If W0 contains a lowercase letter, the index into this table fetches the ASCII code of the corresponding uppercase character.

Listing 10-1 demonstrates these two forms of case conversion: `if...eor` and table lookup.

```
// Listing10-1.S
//
// Lowercase-to-uppercase conversion

            #include    "aoaa.inc"

            .section    .rodata, ""

ttlStr:     .asciz      "Listing 10-1"
```

```
textStr:    .ascii      "abcdefghijklmnopqrstuvwxyz\n"
            .ascii      "ABCDEFGHIJKLMNOPQRSTUVWXYZ\n"
            .asciz      "0123456789\n"

// Translation table to convert lowercase to uppercase:

xlatTbl:    .byte       0, 1, 2, 3, 4, 5, 6, 7
            .byte       8, 9, 10, 11, 12, 13, 14, 15
            .byte       16, 17, 18, 19, 20, 21, 22, 23
            .byte       24, 25, 26, 27, 28, 29, 30, 31
            .byte       32, 33, 34, 35, 36, 37, 38, 39
            .byte       40, 41, 42, 43, 44, 45, 46, 47
            .byte       48, 49, 50, 51, 52, 53, 54, 55
            .byte       56, 57, 58, 59, 60, 61, 62, 63
            .byte       64
            .ascii      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            .byte       91, 92, 93, 94, 95, 96
            .ascii      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            .byte       123, 124, 125, 126, 127

// Various printf format strings this program uses:

fmtStr1:    .asciz      "Standard conversion:\n"
fmtStr2:    .asciz      "\nConversion via lookup table:\n"
fmtStr:     .asciz      "%c"

            .code
            .extern     printf


////////////////////////////////////////////////////////////////////
//
// Return program title to C++ program:

            proc        getTitle, public
            lea         x0, ttlStr
            ret
            endp        getTitle

////////////////////////////////////////////////////////////////////
//
// Here is the asmMain function:

            proc    asmMain, public

            locals  am
            dword   am.x20
            dword   am.x21
            byte    am.shadow, 64
            endl    am

            enter   am.size
            str     x20, [fp, #am.x20]
            str     x21, [fp, #am.x21]
```

```
// Print first title string:

            lea     x0, fmtStr1
            bl      printf

// Convert textStr to uppercase using
// standard "if and EOR" operation:

            lea     x20, textStr    // String to convert
            mov     x21, #'z'       // CCMP doesn't like #'z'.
            b.al    testNot0

// Check to see if W1 is in the range 'a'..'z'. If so,
// invert bit 5 to convert it to uppercase:

stdLoop:    cmp     w1, #'a'
            ccmp    w1, w21, #0b0010, hs
            bhi     notLower
            eor     w1, w1, #0x20
notLower:

// Print the converted character:

            lea     x0, fmtStr
            mstr    x1, [sp]
            bl      printf

// Fetch the next character from the string:

testNot0:   ldrb    w1, [x20], #1
            cmp     w1, #0
            bne     stdLoop

// Convert textStr to uppercase by using
// a lookup table. Begin by printing
// an explanatory string before the
// output:

            lea x0, fmtStr2
            bl      printf

// textStr is the string to convert.
// xlatTbl is the lookup table that will convert
// lowercase characters to uppercase:

            lea     x20, textStr
            lea     x21, xlatTbl
            b.al    testNot0a

// Convert the character from lowercase to
// uppercase via a lookup table:

xlatLoop:   ldrb    w1, [x21, w1, uxtw #0]
```

```
// Print the character:

            lea     x0, fmtStr
            mstr    x1, [sp]
            bl      printf

// Fetch the next character from the string:

testNot0a:  ldrb    w1, [x20], #1
            cmp     w1, #0
            bne     xlatLoop


allDone:    ldr     x20, [fp, #am.x20]
            ldr     x21, [fp, #am.x21]
            leave   // Returns to caller
            endp    asmMain
```

Here's the `build` command and sample output for Listing 10-1:

```
% ./build Listing10-1
% ./Listing10-1
Calling Listing10-1:
Standard conversion:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789

Conversion via lookup table:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
Listing10-1 terminated
```

I didn't attempt to time the two versions, because the call to `printf()` dominates the execution time of the two algorithms. However, because the table-lookup algorithm accesses memory on each character (to fetch a byte from the lookup table), the process is no shorter even though it uses fewer instructions. The lookup table adds 128 bytes (or 256 bytes) to the size of the program's code.

Using a lookup table for a simple computation such as lowercase-to-uppercase conversion carries little benefit. But as the complexity of the computation increases, the table lookup algorithm could become faster. Consider the following code that swaps cases (converts lowercase to uppercase and uppercase to lowercase):

```
// If it's lowercase, convert it to uppercase:

    mov  w1, #'z'
    ldrb w0, [fp, #character]  // Assume "character" is local.
    cmp  w0, #'a'
    ccmp w0, w1, #0b0010, hs
```

```
    bhi  notLower
    eor  w0, w0, #0x20
    b.al allDone

// If it's uppercase, convert it to lowercase:

notLower:
    mov  w1, #'Z'
    cmp  w0, #'A'
    ccmp w0, w1, #0b0010, hs
    bhi  allDone
    eor  w0, w0, #0x20

allDone:
    strb w0, [fp, #character]
```

The lookup-table version is almost identical to Listing 10-1. Only the values in the lookup table change:

```
    lea  x1, xlatTbl2
    ldrb w0, [fp, #character]
    ldrb w0, [x1, w0, uxtw #0]
    strb w0, [fp, #character]
```

The `xlatTbl2` array will contain the lowercase ASCII codes at the indices corresponding to the uppercase characters, in addition to having the uppercase ASCII codes at the indices corresponding to the lowercase ASCII codes.

This case-conversion algorithm still might not be complex enough to justify using a lookup table to improve performance. However, it demonstrates that as the complexity of the algorithm increases (taking longer to execute without a lookup table), the lookup table algorithm's execution time remains constant.

10.1.2 Function Domains and Ranges

Functions computed via table lookup have a limited *domain*, the set of possible input values they accept. This is because each element in the domain of a function requires an entry in the lookup table. For example, the previous uppercase/lowercase conversion functions have the 256-character extended ASCII character set as their domain. A function such as `sin()` or `cos()` accepts the (infinite) set of real numbers as possible input values. You won't find it very practical to implement a function via table lookup whose domain is the set of real numbers, because you must limit the domain to a small set.

Most lookup tables are quite small, usually 10 to 256 entries. Rarely do they grow beyond 1,000 entries. Most programmers don't have the patience to create and verify the correctness of a 1,000-entry table (but see section 10.1.4, "Table Generation," on page 615 for a discussion of generating tables programmatically).

Another limitation of functions based on lookup tables is that the elements in the domain must be fairly contiguous. Table lookups use the input value to a function as an index into the table and return the value at that entry in the table. A function that accepts values 0, 100, 1,000, and 10,000 would require 10,001 elements in the lookup table because of the range of input values. You cannot, therefore, efficiently create such a function via a table lookup. This section on tables assumes throughout that the domain of the function is a fairly contiguous set of values.

The *range* of a function is the set of possible output values it produces. From the perspective of a table lookup, a function's range determines the size of each table entry. For example, if a function's range is the integer values 0 through 255, each table entry requires a single byte; if the range is 0 through 65,535, each table entry requires 2 bytes, and so on.

The best functions you can implement via table lookups are those whose domain and range are always 0 to 255 (or a subset of this range). Any such function can be computed using the same two instructions:

```
lea x1, table
ldrb w0, [x1, w0, uxtw #0]
```

The only thing that changes is the lookup table. The uppercase/lowercase conversion routines presented earlier are good examples of such functions.

Lookup tables become slightly less efficient if the domain or range is not 0 to 255. If the domain of a function is outside 0 to 255 but the range of the function falls within this set of values, your lookup table will require more than 256 entries, but you can represent each entry with a single byte. Therefore, the lookup table can be an array of bytes. The C/C++ function invocation

```
B = Func(X);
```

where `Func` is

```
byte Func( word parm ) { ... }
```

which is easily converted to the following ARM code:

```
lea  x1, FuncTbl
ldr  w0, X       // Using appropriate addressing mode
ldrb w0, [x1, w0, uxtw #0]
strb w0, B       // Using appropriate addressing mode
```

This code loads the function parameter into W0, uses this value (in the range 0 to *maxParmValue*) as an index into the `FuncTbl` table, fetches the byte at that location, and stores the result into `B`. Obviously, the table must contain a valid entry for each possible value of `X` (up to *maxParmValue*). For example, suppose you want to map a cursor position on an 80 × 25 text-based video display in the range 0 to 1,999 (an 80 × 25 video display has 2,000 character positions) to its `X` (0 to 79) or `Y` (0 to 24) coordinate on the screen.

You could compute the X coordinate via this function

```
X = Posn % 80;
```

and the Y coordinate with this formula:

```
Y = Posn / 25;
```

The following code, which realizes these two functions via table lookup, may improve the performance of your code, particularly if you access the table frequently and it is sitting in the processor's cache:

```
lea  x2, xTbl
lea  x3, yTbl
ldr  w4, Posn   // Using an appropriate addressing mode
ldrb w0, [x2, w4, uxtw #0] // Get X.
ldrb w1, [x3, w4, uxtw #0] // Get Y.
```

Given appropriate values in xTbl and yTbl, this will leave the x-coordinate in W0 and the y-coordinate in W1.

If the domain of a function is within 0 to 255 but the range is outside this set, the lookup table will contain 256 or fewer entries, but each entry will require 2 or more bytes. If both the range and domains of the function are outside 0 to 255, each entry will require 2 or more bytes, and the table will contain more than 256 entries.

Recall from Chapter 4 that the formula for indexing into a single-dimensional array (of which a table is a special case) is as follows:

```
Element_Address = Base + Index × Element_Size
```

If elements in the range of the function require 2 bytes, you must multiply the index by 2 before indexing into the table. Likewise, if each entry requires 3, 4, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function F(x), defined by the following C/C++ declaration:

```
short F( word x ) { ... } // short is a half word (16 bits).
```

You can create this function by using the following ARM code (and, of course, the appropriate table named F):

```
lea  x1, F
ldrh w0, x    // Using an appropriate addressing mode
ldrh w0, [x1, w0, uxtw #1] // Shift left does multiply by 2.
```

Any function whose domain is small and mostly contiguous is a good candidate for computation via table lookup. In some cases, noncontiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values (a previously discussed example is processing

switch statement expressions). Such operations are called *conditioning* and are the subject of the next section.

10.1.3 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

```
sin x = sin x|(x∈[-2π,2π])
```

This says that the (computer) function sin(x) is equivalent to the (mathematical) function sin *x* where:

```
-2π <= x <= 2π
```

As you know, sine is a circular function, which will accept any real-value input. The formula used to compute sine, however, accepts only a small set of these values. This range limitation doesn't present any real problems; by simply computing sin(y mod (2π)), you can compute the sine of any input value. Modifying an input value so that you can easily compute a function is called *conditioning the input*. The preceding example computed (x % 2) * pi and used the result as the input to the sin() function. This truncates x to the domain sin() needs without affecting the result.

You can apply input conditioning to table lookups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following C/C++ function:

```
short val( short x )
{
    switch ( x )
    {
        case 0: return 1;
        case 1: return 1;
        case 2: return 4;
        case 3: return 27;
        case 4: return 256;
    }
    return 0;
}
```

This function computes a value for x in the range 0 to 4 and returns 0 if x is outside this range. Since x can take on 65,536 values (being a 16-bit hword), creating a table containing 65,536 hwords where only the first five entries are nonzero seems to be quite wasteful. However, you can still compute this function by using a table lookup if you use input conditioning. The following assembly language code presents this principle:

```
    mov  w0, #0       // Result = 0, assume x > 4
    ldrh w1, [fp, #x] // Assume x is local.
```

```
    cmp  w1, #4       // See if in the range 0 to 4.
    bhi  outOfRange
    lea  x2, valTbl   // Address of lookup table
    ldrh w0, [x2, w1, uxtw #1] // index * 2 (half-word table)
outOfRange:
```

This code checks whether x is outside the range 0 to 4. If so, it manually sets W0 to 0; otherwise, it looks up the function value through the `valTbl` table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table lookup.

10.1.4 Table Generation

One big problem with using table lookups is creating the table in the first place. This is particularly true if the table has many entries. Figuring out the data to place in the table, then laboriously entering the data, and finally checking that data to make sure it is valid is a time-consuming and boring process.

For many tables, there is no way around this. For other tables, however, you can use the computer to generate the table for you. I'll explain this by example. Consider the following modification to the sine function:

$$\sin(x) \times r = \left\langle \frac{(r \times (1000 \times \sin x))}{1000} \;\middle|\; [x \in 0{,}359] \right\rangle$$

This states that x is an integer in the range 0 to 359 (degrees) and that r must be an integer. The computer can easily compute this with the following code:

```
lea   x1, Sines     // Table of 16-bit values
ldr   w0, [fp, #x]  // Assume x is local.
ldrh  w0, [x1, w0, uxtw #1]  // index * 2 for half words
ldrh  w2, [fp, #r]  // Assume r is local.
sxth  x0, w0
sxth  x2, w2
smul  w0, w0, w2    // r *(1000 * sin(x))
mov   w2, #1000
sdiv  x0, x0, x2    // r *(1000 * sin(x))/ 1000
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1,000 and the division by 1,000 because they appear to cancel each other out. Furthermore, this code must compute this function in exactly this order.

All you need to complete this function is `Sines`, a table containing 360 values corresponding to the sine of the angle (in degrees) times 1,000. The C/C++ program in Listing 10-2 generates this table.

```
// Listing10-2.cpp
//
// g++ -o Listing10-2 Listing10-2.c -lm
//
```

```
// GenerateSines
//
// A C program that generates a table of sine values for
// an assembly language lookup table

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char **argv)
{
    FILE *outFile;
    int angle;
    int r;

    // Open the file:

    outFile = fopen("sines.inc", "w");

    // Emit the initial part of the declaration to
    // the output file:

    fprintf
    (
        outFile,
        "Sines:"  // sin(0) = 0
    );


    // Emit the Sines table:

    for(angle = 0; angle <= 359; ++angle)
    {
        // Convert angle in degrees to an angle in
        // radians using:
        //
        // radians = angle * 2.0 * pi / 360.0;
        //
        // Multiply by 1000 and store the rounded
        // result into the integer variable r.

        double theSine =
            sin
            (
                angle * 2.0 *
                3.14159265358979323846 /
                360.0
            );
        r = (int) (theSine * 1000.0);

        // Write out the integers eight per line to the
        // source file.
        // Note: If (angle AND %111) is 0, then angle
        // is divisible by 8 and you should output a
        // newline first.
```

```
        if((angle & 7) == 0)
        {
            fprintf(outFile, "\n\t.hword\t");
        }
        fprintf(outFile, "%5d", r);
        if ((angle & 7) != 7)
        {
            fprintf(outFile, ",");
        }

    } // endfor
    fprintf(outFile, "\n");

    fclose(outFile);
    return 0;

} // end main
```

Compiling and running the program in Listing 10-2 produces the file *sines.inc* containing the following text (truncated for brevity):

```
Sines:
     .hword      0,   17,   34,   52,   69,   87,  104,  121
     .hword    139,  156,  173,  190,  207,  224,  241,  258
     .hword    275,  292,  309,  325,  342,  358,  374,  390
     .hword    406,  422,  438,  453,  469,  484,  499,  515
     .hword    529,  544,  559,  573,  587,  601,  615,  629
     .hword    642,  656,  669,  681,  694,  707,  719,  731
     .hword    743,  754,  766,  777,  788,  798,  809,  819
        .
        .
        .
     .hword   -529, -515, -500, -484, -469, -453, -438, -422
     .hword   -406, -390, -374, -358, -342, -325, -309, -292
     .hword   -275, -258, -241, -224, -207, -190, -173, -156
     .hword   -139, -121, -104,  -87,  -69,  -52,  -34,  -17
```

Obviously, writing the C program that generated this data is much easier than entering and verifying this data by hand. You can also use Pascal/Delphi, Java, C#, Swift, or another HLL to write the table-generation program. Because the program will execute only once, its performance is not an issue.

Once you run the table-generation program, the only step left is to cut and paste the table from the file (*sines.inc* in this example) into the program that will actually use the table (or, alternatively, use the `#include "sines.inc"` directive to include the text in your source file).

10.2 Table-Lookup Performance

In the early days of PCs, table lookups were a preferred way to do high-performance computations. Today, it is common for a CPU to be 10 to

100 times faster than main memory. As a result, using a table lookup may not be faster than doing the same calculation with machine instructions. However, the on-chip CPU cache memory subsystems operate at near-CPU speeds. Therefore, table lookups can be cost-effective if your table resides in cache memory on the CPU. This means that the way to get good performance from table lookups is to use small tables (because the cache has only so much room) and to use tables whose entries you reference frequently (so the tables stay in the cache).

Ultimately, the best way to determine whether a table lookup is faster than a calculation is to write both versions of the code and time them. Although the "10 million loop and time" approach is probably good enough for coarse measurements, you might also want to find and use a decent profiler program that will produce much better timing results. See "For More Information" for additional details.

10.3 Moving On

Using table lookups to optimize applications has grown out of favor as CPU speeds have increased and memory access times have not kept pace. Nevertheless, this short chapter covered the instances when table lookups are still useful. It began with a discussion of basic table lookup operations, then covered domain conditioning and using software to automatically generate tables. It concluded with a few notes on how to decide whether table lookups are the right choice for a particular project.

On modern CPUs, multiple cores and SIMD instruction sets are the common way of improving application performance. The next chapter discusses the ARM Neon/SIMD instruction set and how you can use it to improve program performance.

10.4 For More Information

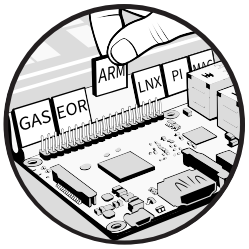
- Donald Knuth's *The Art of Computer Programming, Volume 3: Searching and Sorting*, 2nd edition (Addison-Wesley Professional, 1998), contains a lot of useful information about searching for data in tables.
- See my book *Write Great Code*, Volume 1, 2nd edition (No Starch Press, 2020) or the electronic version of *The Art of Assembly Language* at *https://www.randallhyde.com* for details concerning the operation of cache memory and how you can optimize its use.
- For information on profiler programs, see "Getting Started with dotTrace on macOS and Linux" by Maarten Balliauw at *https://blog.jetbrains.com/dotnet/2023/02/22/getting-started-with-dottrace-on-macos-and-linux*. You can also check out "13 Profiling Software to Debug Application Performance Issue" by Amrita Pathak at *https://geekflare.com/application-profiling-software/*.

TEST YOURSELF

1. What is the domain of a function?
2. What is the range of a function?
3. Provide the code that implements the following functions (using pseudo-C prototypes and `f` as the table name):
   a. `byte f(byte input)`
   b. `halfword f(byte input)`
   c. `byte f(word input)`
   d. `word f(word input)`
4. What is domain conditioning?
5. Why might table lookups not be effective on modern processors?

11

NEON AND SIMD PROGRAMMING



This chapter discusses the vector instructions on the ARM. This special class of instructions provides parallel processing, traditionally known as *single-instruction, multiple-data (SIMD)* instructions because, quite literally, a single instruction operates on several pieces of data concurrently. As a result of this concurrency, SIMD instructions can often execute several times faster (in theory, as much as 32 to 64 times faster) than the comparable *single-instruction, single-data (SISD)* instructions that compose the standard ARM instruction set.

Vector instructions, also known as the *Neon instruction set* or *ARM Advanced SIMD*, provide an extension to the standard scalar instructions. While a *scalar instruction* operates on a single piece of data at a time, the Neon instructions simultaneously operate on a *vector* (a fancy name for an array) of data objects.

This chapter covers a brief history of SIMD instructions, then discusses the ARM Neon architecture (including the vector registers) and Neon data types. The majority of this chapter then covers the Neon instruction set. A complete treatise on SIMD programming is beyond the scope of this book; however, it wouldn't do to write this chapter without at least a few SIMD programming examples in order to demonstrate the benefits of SIMD programming, so this chapter concludes with examples that show a bitonic sort and a numeric-to-hex-string conversion.

11.1 The History of SIMD Instruction Extensions

The Neon instruction set extensions were added to the ARM instruction set long after the ARM was created. Arm created Neon to counter competition from the Intel x86 CPU family. To understand why the Neon instruction set is so radically different from the standard instruction set, you have to understand the history of SIMD (vector) instruction sets.

The first vector computers were supercomputers such as the CDC Star-100, Texas Instruments Advanced Scientific Computer (ASC), and Cray computers, which could operate on a vector of data with a single instruction. These vector computers were the precursor to the early SIMD computers such as the Thinking Machines CM-1 and CM-2. Ultimately, supercomputers moved away from the SIMD approach when Intel introduced SIMD features on its low-cost i860 (and, later, Pentium processors).

The Intel Multimedia Extensions (MMX) instruction set was the first widely adopted desktop SIMD architecture. Intel added parallel integer arithmetic instructions to the venerable x86 instruction set to accelerate digital audio processing and other digital signal processing applications. The PowerPC followed this with the much more capable AltiVec architecture (which included support for single-precision floating-point values). Intel then produced the SSE2 and SSE3, AVX, AVX2, and AVX-512 SIMD instruction architectures (which now include full double-precision floating-point support).

Intel's approach to adding vector instructions to its x86 series CPUs was a bit hackneyed. Given the limited transistor budgets on CPUs in the middle 1990s, Intel added a few vector instructions (MMX) in its early Pentium processors and then extended the SIMD instruction set as its CPUs became larger and had more transistors available to implement advanced features. This evolution produced a bit of a kludge, with new sets of instructions replicating and obsoleting older instructions (with the newer instruction set's ability to handle more data or handle data differently).

By the time ARM added SIMD instructions via its *Neon Advanced SIMD* instructions, Intel had gone through multiple generations of SIMD instructions; Arm was able to cherry-pick the more interesting and useful instructions from Intel's set, leaving behind all the kruft and legacy instructions. For this reason, the Neon instruction set is considerably more compact and much easier to understand than Intel's MMX/SSE/AVX instruction sets.

11.2 Vector Registers

The ARM provides 32 main FP/Neon registers that are 128 bits each, broken into five groups based on their size:

- V0 to V31, the 128-bit vector registers (for Neon instructions), also referenced as Q0 to Q31, the qword registers; the V*n* names support special syntax for vector operations
- D0 to D31, the 64-bit double-precision floating-point registers
- S0 to S31, the 32-bit single-precision floating-point registers
- H0 to H31, the 16-bit half-precision floating-point registers
- B0 to B31, the 8-bit byte registers

Figure 11-1 shows the vector register layout.

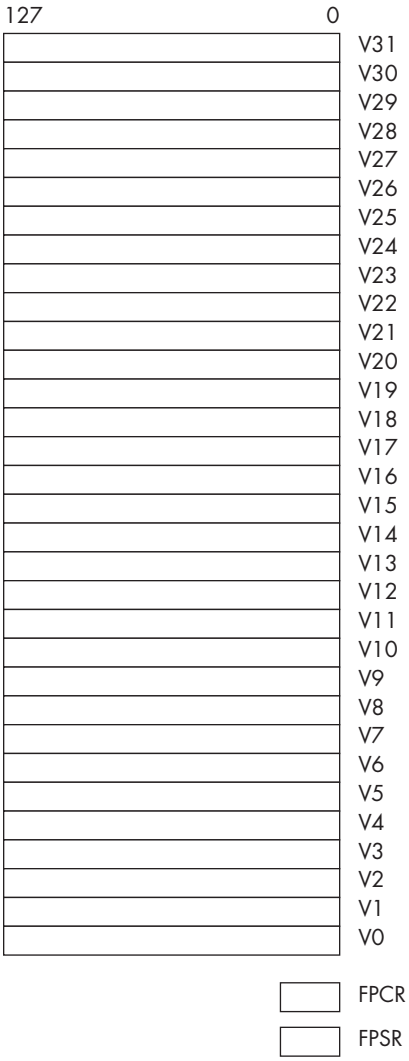


Figure 11-1: The FP/Neon registers

The B*n*, H*n*, S*n*, D*n*, and V*n* registers overlay one another, as shown in Figure 11-2.

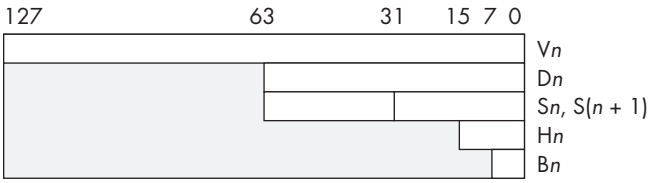


Figure 11-2: Byte, half-word, single, double, and vector register overlays

See Chapter 6 for more information about the scalar floating-point D*n*, S*n*, and H*n* registers. Keep in mind, however, that if you mix vector and floating-point operations in your code, the instructions share the same register set.

Figures 11-1 and 11-2 give the impression that the V*n* registers are 128-bit registers (which, presumably, you can manipulate as a single 128-bit value). In fact, the V*n* registers are vectors containing sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit, or (single) 128-bit values, as Figure 11-3 shows.

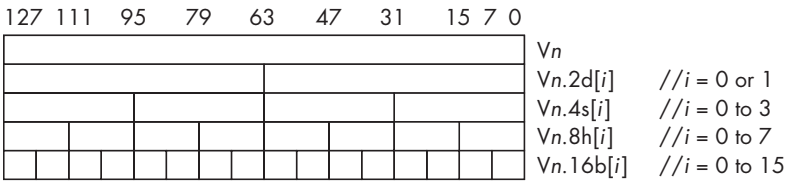


Figure 11-3: Vector register overlays

When an instruction operates on a particular element of a vector register, you reference that element by using one of the following register names (in all cases, *n* represents a vector register number in the range 0 to 31):

- V*n* or Q*n* when referencing the whole 128-bit register
- V*n*.B when treating the whole register as an array of 16 bytes
- V*n*.H when treating the whole register as an array of eight half words
- V*n*.S when treating the whole register as an array of four words (single-precision values)
- V*n*.D when treating the whole register as an array of two dwords (double-precision values)
- V*n*.2D[0] or V*n*.2D[1] when referencing 64-bit double-precision in bit positions 0 to 63 or 64 to 127, respectively
- V*n*.4S[0], V*n*.4S[1], V*n*.4S[2], V*n*.4S[3] when accessing a 32-bit single-precision value in bit positions 0 to 31, 32 to 63, 64 to 95, or 96 to 127, respectively

- V*n*.8H[0], V*n*.8H[1], . . . , V*n*.8H[7] when accessing a 16-bit half-word value in bit positions 0 to 15, 16 to 31, . . . , 112 to 127, respectively
- V*n*.16B[0], V*n*.16B[1], . . . , V*n*.16B[15] when accessing an 8-bit byte in bit positions 0 to 7, 8 to 16, . . . , 120 to 127

The exact name to choose will depend on the instruction and situation. You'll see examples of these registers in use in the next section, "Vector Data Movement Instructions," particularly section 11.3.4, "Vector Load and Store," on page 632.

Figures 11-2 and 11-3 showed the five basic types associated with the data in a vector register: bytes, half words, single-precision values, double-precision values, and 128-bit qwords. In fact, the 32-bit (single) and 64-bit (double) fields support both floating-point (single and double) and integer (word and dword) types, bringing the total number of types to seven.

Except for the special 128-bit case, the vector registers contain arrays of bytes, half words, words, and qwords. For reasons you'll learn when this chapter discusses vector operations, each element of the array is known as a *lane*. When performing operations using two vector registers, the CPU generally computes results by using the operands in corresponding lanes in the two source registers and stores the result in the corresponding lane in a destination register. For example, suppose that V1 contains 2.0 in the HO 64 bits (lane 1) and 1.0 in the LO 64 bits (lane 0), and that V2 contains 20.0 in lane 1 and 10.0 in lane 0. Summing these two vector registers and storing the result in V3 produces 22.0 in lane 1 and 11.0 in lane 0.

Although the vector registers generally contain arrays of data (when performing SIMD operations), don't forget that the floating-point registers (D*n* and S*n*) overlay the vector registers as well. When doing normal floating-point operations (see Chapter 6), these registers contain a single value rather than an array of values. These single values are known as *scalars*.

Very few operations treat an entire 128-bit Neon register as a scalar value. Those that do (mainly load and store instructions) use the name Q*n* to denote a scalar value rather than V*n* (a vector value).

11.3 Vector Data Movement Instructions

Move instructions are the most common integer and floating-point instructions you'll use in the Neon instruction set. In this section, you'll learn how to use these instructions to move data between registers, load constants into Neon registers, and load and store vector registers to and from memory.

11.3.1 Data Movement Between Registers

You can use the `mov` instruction to move data between vector registers. Unfortunately, the obvious syntax won't work:

```
mov v0, v1  // Generates a syntax error
```

The mov instruction copies elements of a vector into a vector register. It can copy data between two vector registers or data between a general-purpose (X*n* or W*n*) register and a vector register. The exact syntax depends on how much data you're copying and the location of the source and destination registers (vector or general-purpose).

Moving data from a 32-bit general-purpose register (W*m*) into a vector register (V*n*) uses one of the following syntaxes:

```
mov Vn.B[i], Wm  // Inserts LO byte of Wm into Vn[i] (i = 0 to 15)
mov Vn.H[i], Wm  // Inserts LO hword of Wm into Vn[i] (i = 0 to 7)
mov Vn.S[i], Wm  // Inserts Wm into Vn[i] (i = 0 to 3)
```

The index *i* must be a literal integer constant, as demonstrated in the following examples:

```
mov v0.b[15], w0  // Copy LO byte of W0 into lane 15 of V0.
mov v1.h[0], w2   // Copy LO hword of W2 into lane 0 of W1.
mov v2.s[2], w1   // Copy W1 into lane 2 of V2.
```

Moving data from a 64-bit general-purpose register (X*m*) into a vector register (V*n*) uses the following syntax:

```
mov Vn.D[i], Xm   // Inserts Xm into Vn[i] (i = 0 to 1)
```

I've used the word *inserts* in these examples because the mov instruction copies only the byte, hword, word, or dword into the vector register at the index that *i* specifies. It does not affect the other data in V*n*. For example

```
mov v0.b[4], w0
```

inserts only the LO byte from W0 into lane 4 in the V0 register; it leaves all other bytes in V0 unchanged. Moving bytes, hwords, and words is possible only when using the W*m* register; if you use X*m* in the instruction, you can move only 64 bits. The type specification for the vector register is S (single-precision) for 32 bits and D (double-precision) for 64 bits. You use this designation even when copying 32-bit and 64-bit integers.

NOTE *The ARM instruction* ins *(insert) is a synonym for* mov *when copying data from a general-purpose register to a vector register—yet another reason for saying these instructions* insert *data rather than saying they* copy *data.*

The previous examples copy the value from a 32- or 64-bit general-purpose register into a vector register. You can also copy data from one vector register (V*n*) to another (V*m*) by using the following syntax:

```
mov Vm.8B, Vn.8B    // Copy 64 bits.
mov Vm.16B, Vn.16B  // Copy 128 bits.
```

These instructions copy 64 bits (8 bytes, four half words, two words, or a single dword) or 128 bits (16 bytes, eight half words, four words / single-precision values, or two dwords / double-precision values) from one vector register to another. In theory, you should be able to enter something like `mov v1, v0` or `mov q1, q0` to move the contents of the 128-bit vector register V0 (Q0) into V1 (Q1). Sadly, Gas does not accept this syntax, so you'll have to use one of the previous four instructions, as in the following example:

```
mov v0.16b, v1.16b  // Copies V1 to V2
```

You can also extract a single byte from one vector register and insert it in an arbitrary lane in another vector register, using the following syntax

```
mov Vm.B[i1], Vn.B[i2]
```

where *i2* is the index of a byte in the source vector and *i1* is the destination index. Both indices must be in the range 0 to 15.

You can also extract a half word from one vector and insert that into another:

```
mov Vm.H[i1], Vn.H[i2]
```

The rules are the same for bytes, except that the two index values must be in the range 0 to 7.

You can copy words (single-precision values) and dwords (double-precision values) by using the following syntax:

```
mov Vm.S[i1], Vn.S[i2]  // i1 and i2 must be in range 0 to 3.
mov Vm.D[i3], Vn.D[i4]  // i3 and i4 must be in range 0 to 1.
```

Here's an example that copies the LO dwords of V0 and V1 merges them into the two dwords in V2:

```
mov v2.d[0], v0.d[0]
mov v2.d[1], v1.d[0]
```

Thus far, I've described how to move data from a general-purpose register to a vector register and between two vector registers. The only missing combination is moving data from a vector register to a general-purpose register, handled by the following `mov`, `umov`, and `smov` instructions:

```
mov  Wn, Vm.S[i0]  // Copies 32 bits
mov  Xn, Vm.D[i1]  // Copies 64 bits

umov Wn, Vm.B[i1]  // Zero-extends byte to 32 bits
umov Wn, Vm.H[i2]  // Zero-extends hword to 32 bits
umov Xn, Vm.D[i3]  // Copies 64 bits

smov Wn, Vm.B[i5]  // Sign-extends byte to 32 bits
smov Wn, Vm.H[i6]  // Sign-extends hword to 32 bits
```

```
smov Xn, Vm.B[i5]  // Sign-extends byte to 64 bits
smov Xn, Vm.H[i6]  // Sign-extends hword to 64 bits
smov Xn, Vm.S[i7]  // Sign-extends word to 64 bits
```

There are no 8- or 16-bit zero extensions to 64 bits. Zero-extending into W*n* automatically zero-extends all the way through the upper 32 bits of X*n*. Here are some examples of these instructions:

```
mov  w0, v0.s[0]  // Copy lane 0 (word) of V0 to W0.
mov  x1, v7.d[1]  // Copy lane 1 (dword) of V7 to X1.
umov w0, v1.b[2]  // Copy and zero-extend V1[2] byte to W0.
smov x1, v0.s[3]  // Copy and sign-extend V0[3] (word) to X1.
```

Remember that `smov x1, v0.s[3]` is moving an integer value, even though the specified type is S (single-precision).

11.3.2 Vector Load Immediate Instructions

The ARM CPU provides a limited set of instructions that allow you to load certain immediate constants into a vector register. The integer versions of these instructions allow only an unsigned 8-bit immediate operand that can be used as is or shifted to the left 1, 2, or 3 bytes (filling vacated positions with 0s or 1s). Furthermore, these immediate instructions copy the data into every lane of a byte array, half-word array, or word array. The floating-point versions of these instructions allow a limited set of floating-point constants (the same limitations as for scalar floating-point constants; see section 6.9.1.4, "`fmov` with Immediate Operand," on page 334).

The standard *move immediate* instruction is `movi`

```
movi Vn.size, #uimm8
movi Vn.size, #uimm8, lsl #c  // size = 4H, 8H, 2S, or 4S
movi Vn.size, #uimm8, msl #c  // size = 2S or 4S
movi Vn.2D, #uimm64
movi Dn, #uimm64
```

where *size* is 16B, 8B, 4H, 8H, 2S, or 4S; *uimm8* is an 8-bit constant; and *uimm64* is either `0` or `0xFFFFFFFFFFFFFFFF`. The `lsl #c` component is optional for instructions with 4H, 8H, 2S, and 4S sizes. The `msl #c` option is optional for 2S and 4S sizes. The `movi` instructions initialize all lanes in the vector register, or just the lanes in the LO 64 bits, with the specified immediate constant. The following paragraphs describe the specific variants of each of these instructions.

The `movi Vn.8B, #uimm8` instruction fills each of the LO 8 bytes of V*n* with the specified constant and the HO 64 bits of the register with 0s. For example

```
movi v0.8b, #0x80
```

loads 0x80808080 into V0.

The `movi Vn.16B, #uimm8` instruction fills all 16 bytes of V*n* with the specified constant. Each lane receives a copy of the *uimm8* value.

The `movi Vn.4H, #uimm8` instruction fills the four hword lanes in the LO 64 bits of V*n* with a copy of the *uimm8* constant, and fills the HO 64 bits of V*n* with 0. Because this instruction accepts only 8-bit immediate constants, the HO 8 bits of each half-word lane will contain 0s. For example

```
movi v1.4h, #1
```

loads 0x0001000100010001 into V1.

The `movi Vn.4H, #uimm8, lsl #0` instruction is identical to `movi Vn.4H, #uimm8`. If the shift constant is #8, this instruction shifts the immediate constant to the left eight positions before storing it into the four half-word lanes (in the LO 64 bits of V*n*). In this case, the LO 8 bits of each of these lanes will contain 0s. For example

```
movi v1.4h, #1, lsl #8
```

loads 0x0100010001000100 into V1.

The `movi Vn.8H, #uimm8` and `movi Vn.8H, #uimm8, lsl #c` instructions do the same thing as the 4H instructions, except that they store the immediate constant (shifted by 0 or 8 bits) into all eight lanes of the V*n* register.

The `movi Vn.2S, #uimm8` instruction fills the two word (single-precision) lanes in the LO 64 bits of V*n* with a copy of the *uimm8* constant, and fills the HO 64 bits of V*n* with 0. Because this instruction accepts only 8-bit immediate constants, the HO 24 bits of each word lane will contain 0s. Although the type specification is S, this instruction assigns integer constants, not floating-point constants, to the lanes. If the optional shift clause is present (`movi Vn.2S, #uimm8, lsl #c`, where *c* is 0, 8, 16, or 24), this instruction will shift the 8-bit constant by the specified number of bits before storing the constant into the two lanes. Here are a few examples:

```
movi v3.2s, #1          // Loads 0x0000000100000001 into V3
movi v4.2s, #1, lsl #8  // Loads 0x0000010000000100 into V4
movi v5.2s, #1, lsl #16 // Loads 0x0001000000010000 into V5
movi v6.2s, #1, lsl #24 // Loads 0x0100000001000000 into V6
```

The `movi Vn.2S, #uimm8, msl #c` instruction is almost identical to its `lsl` counterpart, except it shifts 1 bits rather than 0 bits into the vacated positions during the shift-left operation. The shift count is limited to 8 or 16 rather than 0, 8, 16, and 24 (an annoying inconsistency). For example

```
movi v5.2s, #1, msl #16
```

loads 0x0001FFFF0001FFFF into V5.

The `movi Vn.4S, #uimm8` instruction fills the four word (single-precision) lanes in V*n* with a copy of the *uimm8* constant. Otherwise, this instruction (and the variant with shifting) behaves identically to the 2S version.

The `movi Vn.2D, #uimm64` instruction loads one of two constants (0 or –1) into the two dword lanes of the V*n* register. Once again, keep in mind that

these are integer constants, not floating-point constants, despite the use of the 2D type specifier.

The second move immediate instruction is `mvni` (move and *not* immediate). It supports the following syntax

```
mvni Vn.size, #uimm8 {, (lsl | msl) #c}
```

where *`size`* and *`uimm8`* have the same meanings as given for `movi`.

The operations are the same as for `movi`, except that `mvni` inverts all the bits before storing them into the lanes of the V*n* destination register. The HO 64 bits of V*n* still receive 0s for the 4H and 2S type specifiers, as shown in the following examples:

```
mvni v2.4h, #1, lsl #8  // Loads 0xFEFFfeffFEFFfeff into V2
mvni v4.2s, #2, msl #8  // Loads 0xFFFFFD00fffffd00 into V4
```

Note the absence of the 2D types for the `mvni` instruction. These instructions are unnecessary because the two allowable `movi` *`uimm64`* constants are already the inverse of each other. If you want inverted bits, just use the other *`uimm64`* constant (0 versus –1) with the `movi` instruction.

The third form of the move immediate instruction, `fmov`, allows you to load certain floating-point constants into the lanes of a vector register. The allowable syntax is the following:

```
fmov Vn.2S, #fimm
fmov Vn.4S, #fimm
fmov Vn.2D, #fimm
```

The floating-point immediate constant (*`fimm`*) must be a value defined by

$$\pm n \div 16 \times 2r$$

where $16 \le n \le 31$ and $-3 \le r \le 4$. You cannot represent 0.0 with this formula; if you need to load 0.0 into the lanes of a vector register, just load the integer constant 0 into those lanes by using the `movi` instruction (all 0 bits is 0.0):

```
fmov v0.2s, #1.0  // Loads [0.0, 0.0, 1.0, 1.0] into V0
fmov v0.2d, #2.0  // Loads [2.0, 2.0] into V0
```

The move immediate instructions load only certain constant values into the vector registers. The following are the exact values you can load as immediate floating-point constants (Gas will accept only these values):

```
0.1250000  0.1328125  0.1406250  0.1484375
0.1562500  0.1640625  0.1718750  0.1796875
0.1875000  0.1953125  0.2031250  0.2109375
0.2187500  0.2265625  0.2343750  0.2421875
0.2500000  0.2656250  0.2812500  0.2968750
```

```
0.3125000  0.3281250  0.3437500  0.3593750
0.3750000  0.3906250  0.4062500  0.4218750
0.4375000  0.4531250  0.4687500  0.4843750
0.5000000  0.5312500  0.5625000  0.5937500
0.6250000  0.6562500  0.6875000  0.7187500
0.7500000  0.7812500  0.8125000  0.8437500
0.8750000  0.9062500  0.9375000  0.9687500
1.00       1.0625     1.125      1.1875
1.25       1.3125     1.375      1.4375
1.50       1.5625     1.625      1.6875
1.75       1.8125     1.875      1.9375
2.00       2.1250     2.250      2.3750
2.50       2.6250     2.750      2.8750
3.00       3.1250     3.250      3.3750
3.50       3.6250     3.750      3.8750
4.00       4.2500     4.500      4.7500
5.00       5.2500     5.500      5.7500
6.00       6.2500     6.500      6.7500
7.00       7.2500     7.500      7.7500
8.0        8.5        9.0        9.5
10.0       10.5       11.0       11.5
12.0       12.5       13.0       13.5
14.0       14.5       15.0       15.5
16.0       17.0       18.0       19.0
20.0       21.0       22.0       23.0
24.0       25.0       26.0       27.0
28.0       29.0       30.0       31.0
```

Based on the way programs typically use the Neon registers, this is a reasonable set of values, which can be encoded into a 32-bit instruction opcode. To load larger or different constants, see section 11.3.4, "Vector Load and Store," on the next page.

11.3.3 Register or Lane Value Duplication

The `dup` instruction allows you to duplicate a value held in a general-purpose register or in a single lane of a vector register, throughout all the lanes in a vector register. This instruction supports the following forms:

```
dup Vn.2D, Xm        // Copy Xm into lanes 0-1 (64 bits each) in Vn.

dup Vn.8B, Wm        // LO 8 bits of Wm to lanes 0-7 in Vn
dup Vn.16B, Wm       // LO 8 bits of Wm to lanes 0-15 in Vn

dup Vn.4H, Wm        // LO 16 bits of Wm to lanes 0-3 in Vn
dup Vn.8H, Wm        // LO 16 bits of Wm to lanes 0-7 in Vn

dup Vn.2S, Wm        // Wm to lanes 0-1 in Vn
dup Vn.4S, Wm        // Wm to lanes 0-3 in Vn

dup Vn.8B, Vm.B[i1]  // Dup Vm lane i1 through lanes 0-7 in Vn.
dup Vn.16B, Vm.B[i2] // Dup Vm lane i2 through lanes 0-15 in Vn.
```

```
dup Vn.4H, Vm.H[i3]  // Dup Vm lane i3 through lanes 0-3 in Vn.
dup Vn.8H, Vm.H[i4]  // Dup Vm lane i4 through lanes 0-7 in Vn.

dup Vn.2S, Vm.S[i5]  // Dup Vm lane i5 through lanes 0-1 in Vn.
dup Vn.4S, Vm.S[i6]  // Dup Vm lane i6 through lanes 0-3 in Vn.

dup Vn.2D, Vm.D[i7]  // Dup Vm lane i7 through lanes 0-1 in Vn.
```

The first instruction in each pair duplicates only data in the LO 64 bits of V*n*; the second instruction of each pair copies a full 128 bits. The two single instructions copy 128 bits.

11.3.4 Vector Load and Store

The `mov`, `movi`, `mvni`, `fmov`, and `dup` instructions can move data between vector registers and between general-purpose and vector registers, and can load constants into vector registers. However, they don't allow you to load a register from memory or store the value held in a vector register to memory. The Neon instruction set provides several load and store instructions to handle these tasks.

Because the load and store instructions are the most fundamental, this section considers them first. To load or store an entire 128-bit vector register, use the following syntax

```
ldr Qn, memory
str Qn, memory
```

where *memory* is one of the usual ARM memory addressing modes (same as for the scalar `ldr` and `str` instructions). Note the use of Q*n* to denote the register (rather than V*n*). This is one of the few places the Q*n* register is legal (one wonders why they didn't just use V*n*). These instructions will load or store a full 16 bytes, that is, 128 bits.

The `stp` instructions also allow vector register (Q*n*) operands:

```
ldp Qn, Qm, memory
stp Qn, Qm, memory
```

Note that *n* and *m* in these instructions don't have to be consecutive numbers but can be any arbitrary value in the range 0 to 31.

11.3.5 Interleaved Load and Store

The Neon instruction set provides load and store instructions that load data into a single lane across multiple vector registers. These instructions load interleaved data from memory into one, two, three, or four vector registers. The load (`ld1`, `ld2`, `ld3`, and `ld4`) and store (`st1`, `st2`, `st3`, and `st4`) instructions support non-interleaved data, pairs of interleaved data, triplets of interleaved data, and quad-interleaved data, respectively. The following subsections describe these types of interleaved load and store instructions.

11.3.5.1 Interleaved Load and Store Addressing Modes

The interleaved load and store instructions access memory, but they do not support the full set of ARM memory addressing modes, just three

```
instr  {register_list}, [Xn]
instr  {register_list}, [Xn], Xm
instr  {register_list}, [Xn], #imm
```

where *instr* is one of ld*n*/st*n* and *register_list* is a comma-separated set of Q*n* registers that the load and store instructions will use when loading data from, or storing data to, memory. (The following sections discuss *register_list* at greater length.)

The standard register-indirect addressing mode is[Xn]. The ld*n*/st*n* instructions will access the data at the memory address held in general-purpose register X*n*.

The [Xn], Xm addressing mode computes its effective address as the sum of the values in X*n* and X*m*. This is a post-increment addressing mode; immediately after accessing the specified memory address, this mode adds the value of X*m* to X*n*.

The [Xn], #imm addressing mode is also a post-increment addressing mode, which computes its effective address as the sum of Xn + *imm*, then adds the immediate constant to X*n* after referencing the address. The immediate value is limited to the constants 1, 2, 4, 8, 16, 32, 48, or 64, where the *register_list* operand(s) determines the value you must use. The following sections describe the allowable immediate constants for each version of the instruction.

11.3.5.2 ld1/st1

The ld1 instruction loads one to four registers with data from sequential (non-interleaved) memory locations. With a single vector register, the syntax for this instruction is the following

```
ld1 {Vn.8B}, memory
ld1 {Vn.16B}, memory
ld1 {Vn.B}[index], memory

ld1 {Vn.4H}, memory
ld1 {Vn.8H}, memory
ld1 {Vn.H}[index], memory

ld1 {Vn.2S}, memory
ld1 {Vn.4S}, memory
ld1 {Vn.S}[index], memory

ld1 {Vn.2D}, memory
ld1 {Vn.D}[index], memory
```

where *memory* is one of the following:

```
[Xn]
[Xn], Xm
[Xn], #imm
```

The *imm* operand, if present, must match the size of the register operand. That is, for B it must be 1; for 8B, 8; for 16B, 16; for H, 2; and so on.

The `ld1` instruction with the `{Vn.8B}` register list operand loads 8 bytes into the LO 64 bits of V*n*, while the `{Vn.16B}` register list operand loads 16 bytes.

With a 4H or 2S type specification, the `ld1` register also loads 64 bits (four hwords or two words) into the LO 64 bits of V*n*. With an 8H or 4S type, the `ld1` instruction loads 128 bits into V*n*. Although the 8B, 4H, and 2S types and the 16B, 8H, 4S, and 2D types seem to be interchangeable (they load the same amount of data into V*n*), you should aim to pick the most appropriate type for the data you're manipulating. Not only does this improve your documentation, but also the internal microarchitecture of the ARM CPU might be able to optimize its operations better based on the type of data you are using.

With the bare B, H, S, or D type specification, the `ld1` instruction loads a single lane in V*n* with data from memory. This operation does not affect the data in the other lanes in V*n*. This is the most important variant of the `ld1` instruction because it allows you to build up data in a vector register one lane at a time from different locations in memory.

Why does the `ld1` instruction require braces around the vector register specification? The destination operand of this instruction is actually a register *list*. You can specify one to four registers in this list, as shown in the following examples:

```
ld1 {v1.8b}, [x0]
ld1 {v1.8b, v2.8b}, [x0]
ld1 {v1.8b, v2.8b, v3.8b}, [x0]
ld1 {v1.8b, v2.8b, v3.8b, v4.8b}, [x0]
```

The registers that may appear in this list have two restrictions:

- They must be consecutively numbered registers (with V0 being the successor to V31).
- The type specifications must be identical for all registers in the list.

If you have two or more consecutively numbered registers in a list, you can use the shorthand

```
{Vn.t - V(n + m).t}
```

where *m* is 1, 2, or 3, and *t* is one of the usual vector types, as shown in the following examples:

```
ld1 {v1.8b}, [x0]
ld1 {v1.8b - v2.8b}, [x0]
ld1 {v1.8b - v3.8b}, [x0]
ld1 {v1.8b - v4.8b}, [x0]
```

When you specify more than one register in the list, the `ld1` instruction will load values from consecutive locations into the register. For example, the following code will load V0 from the 16 bytes at the address held in X0, V1 from the 16 bytes at X0 + 16, and V2 from the 16 bytes at X0 + 32:

```
ld1 {v0.16b, v1.16b, v2.16b}, [x0]
```

The `st1` instruction supports an identical instruction syntax (except, of course, you substitute the `st1` mnemonic for `ld1`). It stores the contents of the register(s) or lanes from those registers into the specified memory location. Here is an example that demonstrates storing the values in V0 and V1 to the location specified by X0:

```
st1 {v0.16b, v1.16b}, [x0]
```

This instruction stores the value in V0 at the address held in X0, and the value in V1 to address X0 + 16.

11.3.5.3 ld2/st2

The `ld2` and `st2` instructions load and store interleaved data. These two instructions use the following syntax

```
ld2 {Vn.t1, V(n + 1).t1}, memory
ld2 {Vn.t2, V(n + 1).t2}[index], memory
st2 {Vn.t1, V(n + 1).t1}, memory
st2 {Vn.t2, V(n + 1).t2}[index], memory
```

where the register list must contain exactly two registers, and their register numbers must be consecutive. The *t1* size is 8B, 16B, 4H, 8H, 2S, 4S, or 2D, while *t2* is B, H, S, or D. The literal constant *index* is an appropriate lane number for the type's size (0 to 15 for B, 0 to 7 for H, 0 to 3 for S, and 0 to 1 for D). Finally, *memory* is one of the addressing modes described in section 11.3.5.1, "Interleaved Load and Store Addressing Modes," on page 633.

The variants with *index* (which load a single lane into the two registers) load the first register's lane from the specified memory address and load the second register's lane *n* bytes later (where *n* is the size of the lane, in bytes).

The `ld2` instruction with the *t1* type specification (8B, 16B, 4H, 8H, and so on), meanwhile, loads the two registers one value at a time (of the

specified type: B, H, S, or D), alternating destination lanes between the two registers. For example

```
ld2 {v0.8b, v1.8b}, [x0]
```

loads the LO 8 bytes of V0 from memory locations X0, X0 + 2, X0 + 4, X0 + 6, X0 + 8, X0 + 10, X0 + 12, and X0 + 14. It loads the LO 8 bytes of V1 from locations X0 + 1, X0 + 3, X0 + 5, X0 + 7, X0 + 9, X0 + 11, X0 + 13, and X0 + 15. This deinterleaves the data in memory, loading the even bytes into V0 and the odd bytes into V1. Figure 11-4 shows how `ld2` extracts interleaved data from X0 and stores the deinterleaved results in V0 and V1.

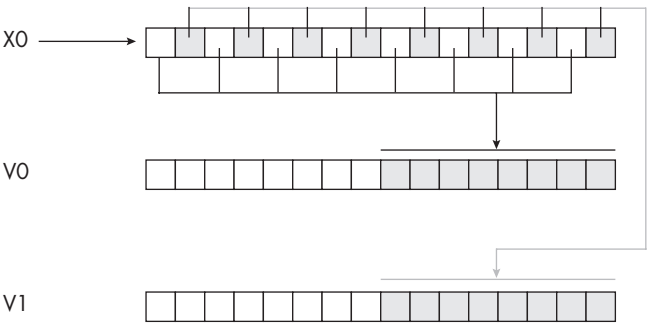


Figure 11-4: The `ld2` deinterleaving operation

If you specify the half-word type (4H or 8H), the `ld2` instruction deinterleaves 16-bit values (even and odd half words). This is particularly useful for deinterleaving digital audio tracks that interleave left and right channels (16 bits per sample).

If you specify 2S/4S or 2D, this instruction will deinterleave words or dwords. For example, if you have an array of floating-point complex numbers, the `ld2` instruction can deinterleave the real and imaginary components.

Because `ld2` deinterleaves pairs of objects, the register list must contain exactly two registers. The assembler will reject any other number of registers in the list.

The `st2` instruction uses the same syntax (except, of course, substituting `st2` for `ld2`). This instruction stores data lanes of the specified type from two registers into memory, interleaving the data between the two registers. The store operation is basically reversing the arrows in Figure 11-4 (that is, copying the data from V0 and V1 into X0, interleaving the two data sets).

11.3.5.4 ld3/st3

The `ld3` and `st3` instructions behave in a similar fashion to `ld2`/`st2`, except that they (de)interleave three objects in memory rather than two, and the register list must contain exactly three registers.

A common example of using the `ld3`/`st3` instructions is to (de)interleave red, green, blue (RGB) values consisting of 3 bytes—an 8-bit red, 8-bit green, and 8-bit blue value—in memory. Using the `ld3` instruction, you can

deinterleave an array of 3-byte RGB values into separate red, green, and blue byte arrays. You can use the `st3` instruction to interleave red, green, and blue values into an RGB array.

11.3.5.5 ld4/st4

Finally, as you've probably figured out by now, the `ld4` instruction copies four consecutive values from memory and stores those values into the same lane of the four registers specified by the four-element register list:

```
ld4 {v4.d, v5.d, v6.d, v7.d}[0], [x0]
```

This instruction copies the four dwords starting at the address held in X0 into lane 0 of V4, V5, V6, and V7, respectively. Figure 11-5 diagrams how this `ld4` instruction operates.

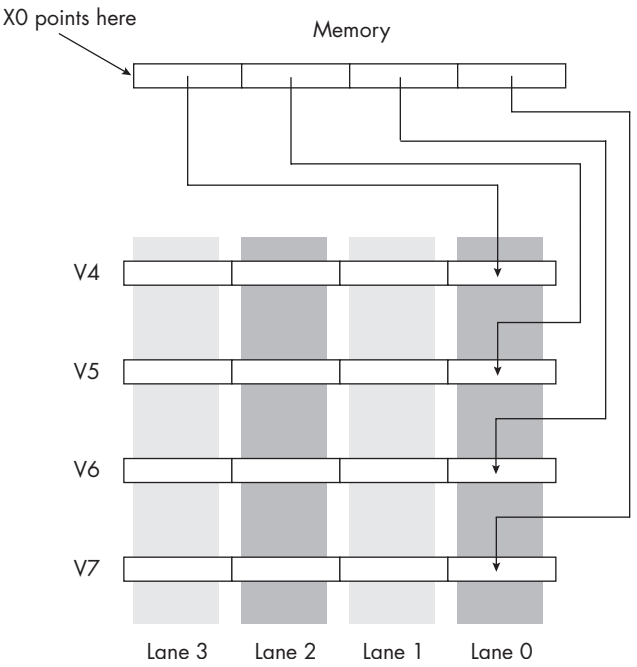


Figure 11-5: The `ld4` instruction operation

The `ld4/st4` instructions are useful for (de)interleaving data in memory that consists of an array of four objects. For example, suppose that you have an array of CMYK (cyan-magenta-yellow-black) color pixels in memory, arranged as shown in Figure 11-6.

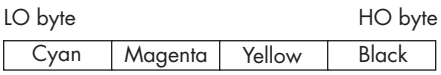


Figure 11-6: CMYK pixel layout in memory

When submitting an image to a printing service, you generally need to provide color separations—that is, four separate images consisting only of the cyan pixels, magenta pixels, yellow pixels, and black pixels. Therefore, you'll need to extract all the cyan pixels from the full-color image and create a separate image for that; likewise for the magenta, yellow, and black pixels.

You can use the `ld4` instruction to extract the cyan, magenta, yellow, and black values from the original image and place those pixels in four separate vector registers. For example, assuming X0 points at the first CMYK pixel (32 bits) in memory

```
ld4 {v0.b - v3.b}[0], [x0]
```

will extract the 4 bytes pointed at by X0 and distribute them into lane 0 of V0 (cyan), V1 (magenta), V2 (yellow), and V3 (black). If you add 4 to X0 and repeat this instruction, specifying lane 1 instead of lane 0, this will separate the second pixel into lane 1 of V0–V4. Repeat this 14 more times and you'll have 16 cyan pixels in V0, 16 magenta pixels in V1, 16 yellow pixels in V2, and 16 black pixels in V3. You can then store away these four registers into the graphic image area that will hold the four-color separations. Repeat this process for all the pixels in the four-color image and you'll have your color separations.

Of course, you can use the 8B and 16B types to process 8 or 16 pixels concurrently:

```
ld4 {v0.16b - v3.16b}, [x0]
```

This instruction copies 64 bytes into V0, V1, V2, and V3, with every fourth byte going into successive lanes in the four registers: V0 gets bytes at offsets $i \% 4$, V1 gets bytes at offsets $(i \% 4) + 1$, and so on, where i is the byte index into memory.

11.3.5.6 ldnr

The `ld1`, `ld2`, `ld3`, and `ld4` instructions load the lanes of one to four registers with successive values in memory, deinterleaving an array of interleaved objects (bytes, hwords, words, or dwords). The `ld1r`, `ld2r`, `ld3r`, and `ld4r` instructions also deinterleave an interleaved object, but the memory object is a single object that the instruction replicates through all lanes in the vector register(s).

The syntax for these instructions is the same as for the `ld`*n* instructions with the addition of the `r` suffix on the mnemonic:

```
ld1r {Vn.t}, memory
ld2r {Vn.t, V(n + 1).t}, memory
ld3r {Vn.t, V(n + 1).t, V(n + 2).t}, memory
ld4r {Vn.t, V(n + 1).t, V(n + 2).t, V(n + 3).t}, memory
```

The .*t* represents a lane type (more on this in a moment), and memory is the usual ld*n* addressing modes. You can also use the range syntax

```
Vn.t - V(n + m).t
```

when specifying two or more registers in the list.

For these instructions, allowable types are 8B, 16B, 4H, 8H, 2S, 4S, and 2D. These type specifications do the following when used with the ld1r instruction:

- 8B loads the first 8 lanes of V*n* with a copy of the byte found at *memory*, replicating that byte in each lane.
- 16B loads all 16 lanes of V*n* with a copy of the byte found at *memory*, replicating that byte in each lane.
- 4H loads the first 4 lanes of V*n*, replicating the hword found at *memory*.
- 8H loads all 8 lanes of V*n*, replicating the hword found at *memory*.
- 2S loads the first 2 lanes of V*n*, replicating the word found at *memory*.
- 4S loads all 4 lanes of V*n*, replicating the word found at *memory*.
- 2D loads the 2 dword lanes of V*n*, replicating the dword found at *memory*.

The ld1r instruction fetches only a single lane value from memory and writes it to all the lanes of the destination register. The ld2r instruction fetches two lane objects from successive memory locations and replicates the first value throughout the first register and the second value throughout the second. The ld3r instruction fetches three lane objects from memory and replicates them through the first, second, and third registers, respectively. Finally, the ld4r instruction fetches four lane objects from memory and uses them to initialize the lanes of the four registers.

11.3.6 Register Interleaving and Deinterleaving

The ld*n*/st*n* and ld*n*r instructions operate between memory and the vector registers. When you want the ability to interleave and deinterleave data appearing in vector registers, leaving the result in a vector register, use the trn1, trn2, zip1, zip2, uzip1, uzip2, and ext instructions.

11.3.6.1 trn1 and trn2

The trn1 and trn2 (transpose) instructions—so called because you can use them to transpose the elements of a 2 × 2 matrix (or larger arrays with a little effort)—extract data from two source registers and interleave that data into a destination register. These instructions use the following syntax

```
trn1 Vd.t, Va.t, Vb.t
trn2 Vd.t, Va.t, Vb.t
```

where *t* can be 8B, 16B, 4H, 8H, 2S, 4S, or 2D. The *d* (destination), *a*, and *b* items are register numbers in the range 0 to 31. These register numbers are

arbitrary (they don't have to be consecutive values, as is the case for the `ldn/stn` and `ldnr` instructions).

The `trn1` instruction copies the data from even-numbered lanes in `Va.t` into the corresponding lanes in `Vd.t`, and data from even-numbered lanes in `Vb.t` into the odd lanes in `Vd.t`, while ignoring the odd-numbered lanes in `Va.t` and `Vb.t`. For example, consider the following instruction:

```
trn1 v0.4s, v2.4s, v4.4s
```

This instruction interleaves the alternate bytes in V2 and V4, leaving the result in V0, as shown in Figure 11-7.

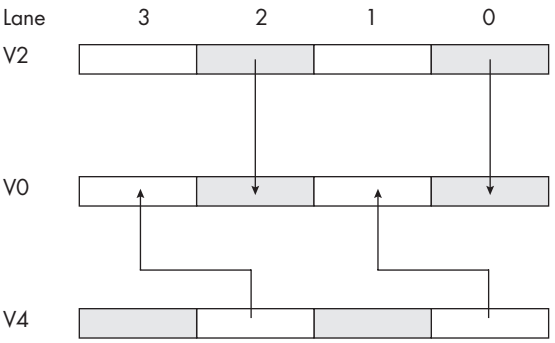


Figure 11-7: The `trn1 v0.4s, v2.4s, v4.4s` operation

The `trn2` instruction copies the values in the odd lanes in `Va.t` and `Vb.t` into alternating lanes in `Vd.t`, as shown in Figure 11-8 (similar to `trn1` except that it swaps the source locations).

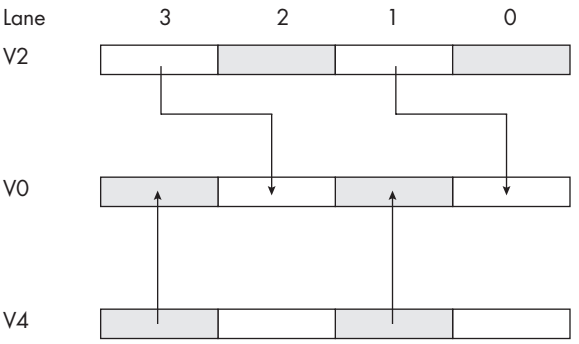


Figure 11-8: The `trn2 v0.4s, v2.4s, v4.4s` operation

Consider the 2×2 matrix of double-precision values held in V2 and V3 as shown in Figure 11-9 (note the positions of the array elements, which is different from what you would normally expect).

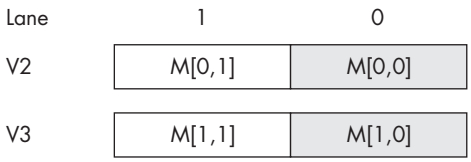


Figure 11-9: A 2×2 matrix held in V2 and V3

The following two instructions will transpose this matrix, leaving the result in V0 and V1:

```
trn1 v0.2d, v2.2d, v3.2d
trn2 v1.2d, v2.2d, v3.2d
```

Of course, `trn1` and `trn2` are generally useful for rearranging and interleaving values in the vector registers, even if you aren't transposing 2×2 matrices.

11.3.6.2 zip1 and zip2

The `zip1` and `zip2` instructions are similar to `trn1` and `trn2` insofar as they produce an interleaved result from data taken from two source registers. The name *zip* comes from *zipper*: the instruction interleaves lanes just like a zipper interleaves the two halves of the connector. Except for the mnemonics, the syntax is identical to `trn1` and `trn2`

```
zip1 Vd.t, Va.t, Vb.t
zip2 Vd.t, Va.t, Vb.t
```

where *t* can be 8B, 16B, 4H, 8H, 2S, 4S, or 2D (all types must be the same in the instruction).

The `zip`*n* and `trn`*n* instructions differ in the way they select the source lanes to interleave. The `zip1` instruction interleaves lane values taken from the beginning of the source registers (consuming half the lanes of each source register and ignoring the remaining lanes). See Figure 11-10 for an example.

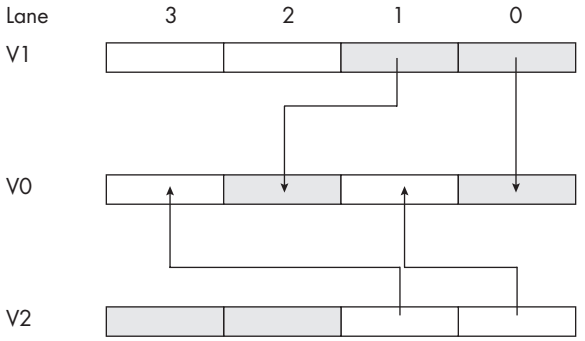


Figure 11-10: The `zip1 v0.4s, v1.4s, v2.4s` operation

The zip2 instruction works similarly except that it processes the second half of the lanes in the source registers. Figure 11-11 shows an example.

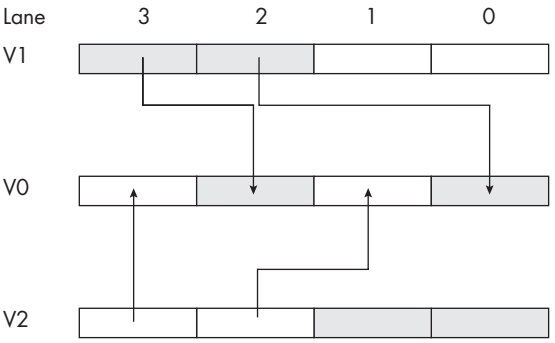


Figure 11-11: The zip2 v0.4s, v1.4s, v2.4s operation

As you can see from these figures, the zip1 and zip2 instructions are typically what you would use to create interleaved data using only registers.

11.3.6.3 uzp1 and uzp2

The uzp1 and uzp2 (unzip1 and unzip2) instructions are the inverse of zip1 and zip2. They take interleaved data in two source registers and produce deinterleaved data in the destination register. Their syntax is the same as that of the trn*n* and zip*n* instructions:

```
uzp1 Vd.t, Va.t, Vb.t
uzp2 Vd.t, Va.t, Vb.t
```

As usual, *t* can be 8B, 16B, 4H, 8H, 2S, 4S, or 2D.

The uzp1 instruction copies the even lanes from V*a*.*t* into the first half of V*d*.*t*, then appends the even lanes of V*b*.*t* to the end of V*d*.*t*. See Figure 11-12 for an example.

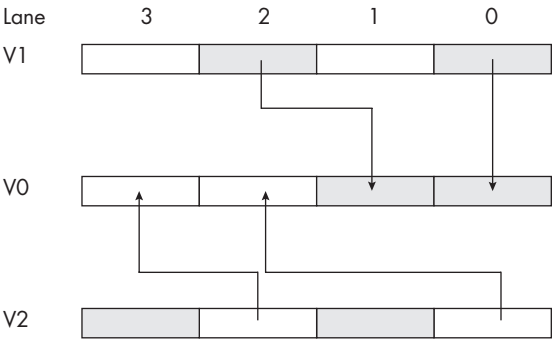


Figure 11-12: The uzp1 v0.4s, v1.4s, v2.4s operation

The uzp2 instruction copies the odd lanes from the source registers. Figure 11-13 shows an example of the uzp2 instruction in action.

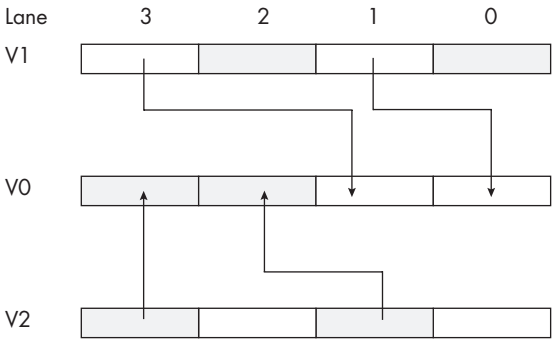


Figure 11-13: The uzp2 v0.4s, v1.4s, v2.4s operation

If the type specifier is 64 bits (8B, 4H, or 2S), the uzp1 and uzp2 instructions leave 0s in the HO lanes of the destination register.

11.3.6.4 ext

The ext (extract) instruction creates an 8- or 16-byte vector from *n* bytes in one vector and 8-*n* (or 16-*n*) bytes from a second vector. This instruction allows you to extract an 8- or 16-byte vector from across two vectors. The syntax for this instruction is as follows

```
ext Vd.8B, Vs1.8B, Vs2.8B, #n
ext Vd.16B, Vs1.16B, Vs2.16B, #n
```

where *n* is a starting index, Vd is the destination register, and Vs_1 and Vs_2 are the source registers.

The ext Vd.8B, Vs_1.8B, Vs_2.8B, #*n* instruction fetches the LO *n* bytes from Vs_2 and copies them to the HO *n* bytes of the LO 64 bits in Vd. It also extracts the LO 8-*n* bytes from Vs_1 and copies them to the LO 8-*n* bytes of Vd. For an example of ext, see Figure 11-14.

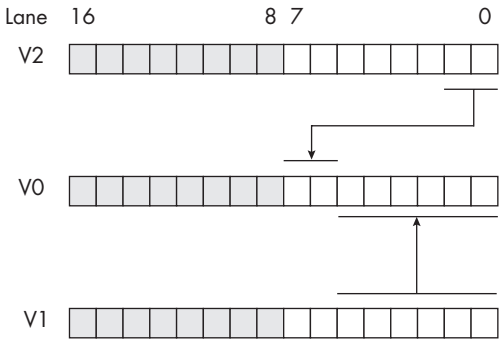


Figure 11-14: The ext v0.8B, v1.8B, v2.8B, #2 instruction

The `ext Vd.16B, Vs1.16B, Vs2.16B, #n` instruction fetches the LO *n* bytes from Vs_2 and copies them to the HO *n* bytes of `Vd`. It also extracts the LO 16-*n* bytes from Vs_1 and copies them to the LO 16-*n* bytes of `Vd` (see Figure 11-15 for an example).

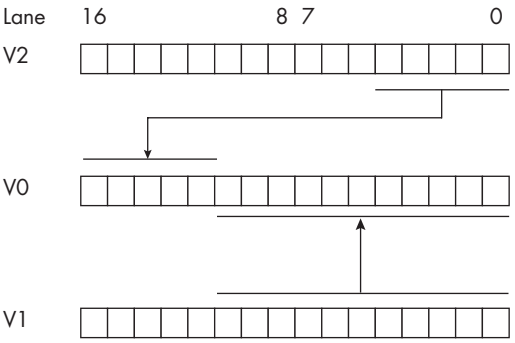


Figure 11-15: The `ext V0.16B, V1.16B, V2.16B, #5` instruction

This instruction supports only the 8B and 16B types. You can easily extract hwords, words, or dwords by choosing an appropriate index value (*n*) that includes all the objects you want to extract.

11.3.7 Table Lookups with tbl and tbx

The `tbl` and `tbx` (table lookup) instructions allow you to exchange all the byte values in one register with values taken from a lookup table containing up to 64 entries. The syntax for these instructions is

```
tbl Vd.8B, {table_list}, Vs.8B
tbl Vd.16B, {table_list}, Vs.16B
tbx Vd.8B, {table_list}, Vs.8B
tbl Vd.16B, {table_list}, Vs.16B
```

where *table_list* is a list of one to four (consecutively numbered) registers, all of which must have a 16B type attached to them. (You can also use the `Vn.t - Vm.t` syntax, where $m > n$ and $m < (n + 4)$.) This list of registers provides a lookup table that contains 16, 32, 48, or 64 entries. The LO byte of the first register is index 0 in the table; the HO byte of the last register is index 15, 31, 47, or 63 into the table.

The `tbl` instruction fetches each byte from the source register (`Vs.t`) and uses its value as an index into the lookup table. It fetches the byte at that index from the table and copies it to the corresponding location in the destination register—that is, the same byte index from which the source byte was taken; so this is equivalent to `Vd[i] = table[ Vs[i] ]`. If the value is out of range (greater than 15, 31, 47, or 63, depending on the size of the table), the `tbl` instruction stores a 0 into the corresponding location in the destination register. The `tbx` instruction works similarly to `tbl`, except that it

leaves the destination location unchanged if the source value is out of range.

For very small tables (64 entries or fewer), you can use `tbl` and `tbx` to implement lookup tables as described in Chapter 10. However, the main purpose of these two instructions is to provide arbitrary vector permutations like the `trn1/trn2`, `zip1/zip2`, `uzp1/uzp2`, and `ext` instructions. Suppose, for example, that you want to reverse the positions of all 16 bytes in a vector register (swapping indices 0 and 15, 1 and 14, 2 and 13, 3 and 12, and so on). Figure 11-16 shows a 16-byte endian swap operation, where the double-ended arrows point to the two locations where the bytes are exchanged.

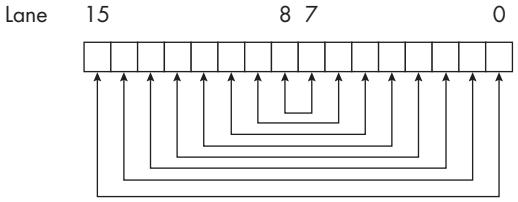


Figure 11-16: A 16-byte endian swap

If you load a vector register with the following 16-byte value

```
0x000102030405060708090a0b0c0d0e0f
```

and then use this value in the source register for the `tbl` (or `tbx`) instruction, `tbl` (or `tbx`) will swap the bytes in a single 16-byte register supplied as the *`table_list`*, storing the reversed bytes in the destination register. Assuming you've loaded this value into V0, the following instruction will swap the bytes in `{V1}`, placing the results in V2:

```
tbl v2.16b, {v1.16b}, v0.16b
```

After you load V1 with the bytes to be swapped and execute this instruction, V2 will contain the swapped values.

To use `tbl` or `tbx` as a vector permutation instruction, load the permutation indexes into the source register (V0 in this example). The indices will always be values in the range 0 to 15, to select specific entries in *`table_list`*. For a true permutation, each of the values (0 to 15) will appear exactly once in the *`source`* register, and there will always be a single register in the *`table_list`*. Because you're limiting the values in the source register to the range 0 to 15, the table index values are always in range, so you can use either `tbl` or `tbx`. Both work exactly the same when the values are not out of range.

Of course, you can use any permutation you like by specifying different values in the source register. As with the `ext` instruction, `tbl` and `tbx` support only the 8B and 16B lane types. However, it's easy enough to synthesize other types (for permutations, anyway) by choosing the positions of the source register lane values to permute hwords, words, and dwords. Obviously, for table lookup operations (rather than permutations), you're

limited to 8-bit values, so hword, word, and dword types don't make any sense.

11.3.8 Endian Swaps with rev16, rev32, and rev64

The `rev16`, `rev32`, and `rev64` instructions are similar to their scalar counterparts `rev16`, `rev32`, and `rev` (see section 3.3, "Little-Endian and Big-Endian Data Organization," on page 133), except, of course, they operate on the lanes in a vector source register rather than on a general-purpose integer register. Here is their syntax:

```
rev16 Vd.t1, Vs.t1  // Swap the bytes in the half-word lanes.
rev32 Vd.t2, Vs.t2  // Swap the bytes in the word lanes.
rev64 Vd.t1, Vs.t3  // Swap the bytes in the double-word lanes.
```

The legal types and lane counts for these instructions appear in Table 11-1.

Table 11-1: Legal Types and Lane Counts for `rev*` Instructions

t	Type and lane count
t1	8B, 16B
t2	8B, 16B, 4H, or 8H
t3	8B, 16B, 4H, 8H, 2S, or 4S

If the lane count and type is 8B, 4H, or 2S, the instruction operates only on the LO 64 bits of the source register (and clears the HO 64 bits of the destination register). If the lane count and type is 16B, 8H, or 4S, these instructions operate on the full 128 bits of the source register.

11.4 Vertical and Horizontal Operations

Up to this point, vector operations have been *vertical*, meaning they've operated on the same lane across multiple registers (which, when stacked as appearing in most figures thus far, show a vertical operational direction). Consider the following vector addition instruction:

```
add v0.16b, v1.16b, v2.16b
```

As for the scalar addition operation (for example, `add w0, w1, w2`), this instruction adds the values of two source registers (V1.16B and V2.16B), producing a sum in the destination register. However, this is not a 128-bit addition operation, but rather an 8-bit operation repeated 16 times. Vector operations typically operate on a lane-by-lane basis, performing multiple small operations in parallel. For this particular instruction, the CPU adds together 16 byte values, producing 16 independent byte results. This is the

magic behind SIMD programming: the ability to do 16 times as much work with a single instruction (so it should run about 16 times faster than running these 16-byte additions individually).

Figure 11-17 shows the lane-by-lane operation of the `add` instruction with the lane-by-lane addition following the arrow directions.

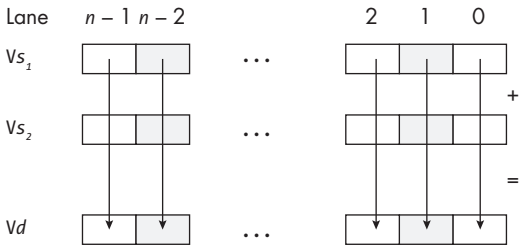


Figure 11-17: Lane-by-lane operations

Lane-by-lane operations are independent of one another, meaning that if any carries, overflows, or other exceptional conditions occur, such anomalies are limited in scope to a single lane. Because there is only a single set of NVZC condition code flags, vector instructions cannot (and do not) affect these flags. If an unsigned carry out of one lane occurs (such as when adding 255 + 1 in a byte lane), the sum wraps around with no indication of overflow or underflow. In general, you must handle overflows completely differently from the way you'd handle them when doing scalar arithmetic. This chapter covers some strategies for doing so when discussing saturation in later sections.

Certain vector instructions provide *horizontal operations*, also known as *reducing operations*. Rather than operating lane by lane between two registers, these operations operate on all the lanes within a single vector register, producing a scalar result. For example, the `addv` instruction will produce the sum of all the lanes in a single vector register.

11.5 SIMD Logical Operations

Because logical (Boolean) operations are computed on a bitwise basis, vector logical operations are unique insofar as you can use them to perform 128 individual bit operations. Whether you treat the source operands as sixteen 1-byte values or as one 128-byte value, the result is the same. For that reason, the vector logical operations support only two types: 8B (for 64-bit operands) and 16B (for 128-bit operands). If you really want to operate on 4H or 2S operands, just specify 8B; you'll get the same result. Likewise, for 8H, 4S, or 2D operands, specifying 16B produces the same result.

The Neon instruction set supports eight logical instructions, as shown in Table 11-2. Here, *t* is 8B or 16B, `Vd` is the destination register, Vs_1 is the left source register, and Vs_2 is the right source register (`Vs` is the only source register for the `not` instruction).

Table 11-2: Neon Logical Instructions

Mnemonic	Syntax	Description
and	and Vd.t, Vs_1.t, Vs_2.t	Vd = Vs_1 & Vs_2
orr	orr Vd.t, Vs_1.t, Vs_2.t	Vd = Vs_1 | Vs_2
orn	orn Vd.t, Vs_1.t, Vs_2.t	Vd = Vs_1 | ~(Vs_2)
eor	eor Vd.t, Vs_1.t, Vs_2.t	Vd = Vs_1 ^ Vs_2
bic	bic Vd.t, Vs_1.t, Vs_2.t	Vd = Vs_1 & ~(Vs_2) (bit clear)
bif	bif Vd.t, Vs_1.t, Vs_2.t	Bit insert if false
bit	bit Vd.t, Vs_1.t, Vs_2.t	Bit insert if true
bsl	bsl Vd.t, Vs_1.t, Vs_2.t	Bitwise select
not	not Vd.t, Vs.t	Vd = ~Vs

The and, orr, and eor instructions do the usual logical operations (same as scalar) and require no further explanation. The orn instruction is similar to bic insofar as it inverts the second source operand prior to the OR operation.

The bic (bit clear) instruction clears all the bits in the value of Vs_1 in the positions containing 1s in Vs_2. It stores the result in Vd. Note that there is no need for a bis (bit set) instruction, because orr will set bits in Vd.

The bif (bit insert if false) and bit (bit insert if true) instructions are unusual insofar as they use three operands in their computation (rather than using a function of two inputs and storing the result in a third operand). The bif instruction copies the bits from Vs_1 to Vd wherever the corresponding bit in Vs_2 contains a 0. In the bit positions where Vs_2 contains a 1, this instruction leaves the corresponding bit in Vd unchanged. The bit instruction works similarly, except it copies the bits when the corresponding bit in Vs_2 contains a 1 (rather than 0).

The bsl (bit select) instruction selects bits from Vs_1 or Vs_2 (and copies them to Vd) based on the original contents of Vd. If Vd originally contained a 1 in a particular bit position, bsl selects the corresponding bit from Vs_1. Otherwise, it selects the bit from Vs_2.

The not instruction inverts all the bits in the source register and stores the result into the destination register. This instruction is different from the other logical instructions, having only a single source operand.

The Neon instruction set supports a few special immediate versions of the orr and bic instructions

```
orr Vd.t, #imm
orr Vd.t, #imm, lsl #shift
bic Vd.t, #imm
bic Vd.t, #imm, lsl #shift
```

where *imm* is an unsigned 8-bit immediate value; the type (*t*) is 2S, 4S, 4H, or 8H; and *shift* is 0 or 8 if *t* is 4H/8H and 0, 8, 16, or 24 if *t* is 2S or 4S. If *shift* is not specified, it is assumed to be 0. These instructions require the H and S types rather than the B types, since they replicate the immediate value through the bytes in the lanes in Vd.t.

11.6 SIMD Shift Operations

Shift instructions are generally considered to be logical operations. However, from a vector point of view, they are more correctly thought of as arithmetic operations because shift operations can produce overflows. Vector shift operations handle overflows in one of four ways:

- Ignoring any carry out of the shift operation (truncation)
- Saturating the shift result
- Rounding the result
- Providing an extended shift operation whose destination operand is larger than the source register

This section describes these various shift operations.

NOTE *The Neon instruction set uses mnemonics based on* `shr` *and* `shl` *for shift left and shift right. This is in contrast to the* `lsl`*,* `lsr`*, and* `asr` *instructions that the scalar integer instruction set uses. I cannot think of a good reason they did it this way; it would have made the instruction set easier to learn had they stuck to a consistent naming convention.*

11.6.1 Shift-Left Instruction

The `shl` instruction shifts each lane of a vector register to the left the specified number of bits. This instruction shifts 0s into the (vacated) LO bits. Any carry out of the HO bit of the lane is lost. The syntax is as follows

```
shl Vd.8B, Vs.8B, #imm
shl Vd.16B, Vs.16B, #imm
shl Vd.4H, Vs.4H, #imm
shl Vd.8H, Vs.8H, #imm
shl Vd.2S, Vs.2S, #imm
shl Vd.4S, Vs.4S, #imm
shl Vd.2D, Vs.2D, #imm
```

where `Vd` is the destination register and `Vs` is the source register. The immediate count value must be in the ranges appearing in Table 11-3 (based on the specified type). The assembler will report an error if an immediate shift value is outside these ranges.

Table 11-3: Valid `shl` Shift Values

Type	Shift range
8B/16B	0 to 7
4H/8H	0 to 15
2S/4S	0 to 31
2D	0 to 63

There is also a scalar shl instruction that operates on the LO dword of a vector register, with the following syntax

```
shl Dd, Ds, #imm
```

where Dd is the destination scalar register and Ds is the source register (corresponding to the LO 64 bits of Vd and Vs). The imm shift count must be in the range 0 to 63. Note that this instruction will zero out the HO 64 bits of Dd.

To shift the lanes by a variable number of bits, see section 11.6.9, "Shift by a Variable Number of Bits," on page 657.

11.6.2 Saturating Shift Left

The saturating shift-left instructions uqshl, sqshl, and sqshlu shift the lanes in a vector to the left the specified number of bit positions. If an overflow (whether signed or unsigned) occurs, these instructions saturate the result to the largest (signed or unsigned) value depending on the instruction. The syntax for these instructions is as follows

```
uqshl  Vd.t, Vs.t, #imm
uqshl  Vd.t, Vs.t, Vc.t
sqshl  Vd.t, Vs.t, #imm
sqshl  Vd.t, Vs.t, Vc.t
sqshlu Vd.t, Vs.t, #imm
sqshlu Vd.t, Vs.t, Vc.t
```

where Vd is the destination register, Vs is the source register, imm is an appropriate immediate shift constant or Vc contains a shift count in the LO byte, and t is type 8B, 16B, 4H, 8H, 2S, 4S, or 2D. The t specification must be the same for Vd and Vs.

The shift value's range depends on the lane type; see Table 11-3 in the previous section for the legal immediate values. For immediate values, the assembler will report an error if the shift constant is out of range. For the register shift count variants, if the LO byte contains an out-of-range value, then the instruction will always saturate the result if a lane contains a nonzero value (see the discussion of saturation that follows). The uqshl instruction shifts values to the left one bit position, storing the result in the corresponding lane in the destination register. If the HO bit is set (before the shift), this instruction stores all 1 bits (the maximum unsigned value) in the destination lane. For example, if a lane contains 0x7F in V1, the corresponding lane will contain 0xFE (0x7F shifted left one position) after the execution of the following:

```
uqshl v0.16b, v1.16b, #1
```

However, if a source lane contains the value 0x80 through 0xFF, then shifting it to the left one position produces 0xFF in the destination lane. In general, if anything other than 0 bits are shifted out of a source lane, the corresponding destination lane will contain 0xFF.

The sqshl instruction is a signed saturation shift-left operation. For signed values, an overflow will occur during a left shift if the H two bits of a

lane contain different values. For negative source values (the HO bit is set), overflow saturates to a result with the HO bit set and all other bits containing 0s (for example, with hword types, 0xa000 will saturate to 0x8000).

The `sqshlu` instruction is similar to `sqshl`, except that it treats the destination as an unsigned value. Positive (and 0) source values will shift to the left exactly like the `uqshl` instruction, while negative source values (with the HO bit set) will saturate to 0.

There are also scalar versions of the `uqshl`, `sqshl`, and `sqshlu` instructions

```
uqshl  Rd, Rs, #imm
sqshl  Rd, Rs, #imm
sqshlu Rd, Rs, #imm
```

where `Rn` (*n* = *d* or *s*) is one of the registers B*n*, H*n*, S*n*, or D*n*, and *d*, *s*, and *imm* have the usual meanings and limitations. Unlike the plain `shl` instruction, these instructions allow byte, hword, and word registers, as well as dword registers.

As for the vector instructions, the `uqshl` instructions do an unsigned saturation. If any bits are shifted out of the HO bit of the source register, these instructions set the destination (B*n*, H*n*, S*n*, or D*n*) to all 1 bits. These instructions zero-extend the result through the rest of the vector register containing R*d*.

The `sqshl` instruction does a signed saturation, leaving the result in the destination (scalar) register. This instruction zeros out the remaining HO bits of the corresponding vector register (that is, all the HO bits beyond the size of the scalar register).

The `sqshlu` instruction does a shift on a signed source value but saturates it to an unsigned value (negative results saturate to 0, just as with the vector register versions of this instruction).

11.6.3 Shift-Left Long

The *shift-left long* instructions `sshll`, `sshll2`, `ushll`, and `ushll2` provide a mechanism to handle overflow during a shift operation. These instructions sign- or zero-extend the value in a lane to twice its size and then perform the left shift on the double-sized source, storing the result into the (double-sized) destination lane. The syntax for these instructions is

```
ushll  Vd.t2, Vs.t, #imm
sshll  Vd.t2, Vs.t, #imm
```

where *t2* is the double-sized type and can be 8H, 4S, or 2D; *t* is the original type and can be 8B, 4H, or 2S. *imm* is the shift count and should be in the range 0 to *n* – 1, where *n* is the number of bits in the *t* type.

The `ushll` instruction zero-extends the values in the source lanes to twice their size, shifts the zero-extended result by the specified number of bits, and stores the result into the corresponding (double-sized) destination lanes. The `sshll` instruction sign-extends the source lane values to twice their size, then shifts the results and stores them in the double-sized destination lanes.

Because these instructions double the size of their values, they operate only on the LO 64 bits of the source register (lanes 0 to 7 for bytes, 0 to 3 for hwords, and 0 to 1 for words). These instructions ignore the HO 64 bits of the source register.

To handle the upper 64 bits of the source register, the ARM provides the `ushll2` and `sshll2` instructions:

```
ushll2 Vd.t4, Vs.t3, #imm
sshll2 Vd.t4, Vs.t3, #imm
```

These accomplish the same operations as the `ushll` and `sshll` instructions, except that they take their source operands from the HO 64 bits rather than the LO 64 bits. To indicate this, the *t4/t3* type pairs must be 8H/16B, 4S/8H, or 2D/4S. The *imm* shift values must match the source lane size in bits (0 to 15 for 8H/16B, 0 to 31 for 4S/8H, and 0 to 63 for 2D/4S).

The `ushll`, `ushll2`, `sshll`, and `sshll2` instructions have no scalar versions. Just use the vector versions and zero out the HO bits yourself if you need this operation.

11.6.4 Shift and Insert

The `sli` and `sri` instructions allow you to shift a source operand a certain number of bits and then (using other instructions) insert other bits into the locations (0 bits) vacated by the shift operation. Here's the syntax for these instructions

```
sli Vd.t, Vs.t, #imm
sri Vd.t, Vs.t, #imm
```

where *t* is the usual set of types: 8B, 16B, 4H, 8H, 2S, 4S, or 2D. For `sli`, *imm* is the shift count, which must be in the range 0 to *n* – 1, where *n* is bit size of a lane. For `sri`, the immediate value is a count in the range 1 to *n*.

The `sli` instruction shifts each lane in `Vs.t` to the left the specified number of bits. It then logically ORs the *n* - *imm* LO bits of `Vd.t` into the result (replacing the 0s that were shifted in) and stores the result back into `Vd.t`, as shown in Figure 11-18.

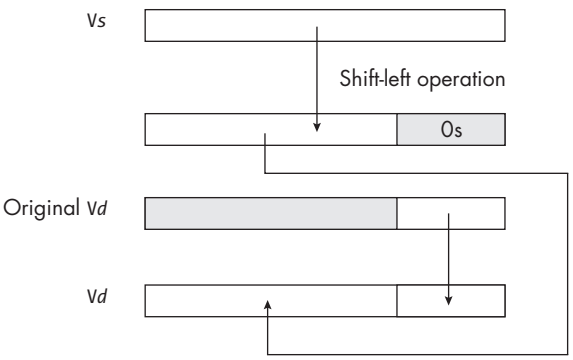


Figure 11-18: The `sli` instruction operation

For example, to shift in 1 bits rather than 0 bits, you could load the destination register with all 1 bits, then execute the `sli` instruction, as shown in the following code:

```
movi    v0.16b, #0xff
movi    v1.4s, #0x1
sli     v0.4s, v1.4s, #4
```

This produces 0x0000001f0000001f0000001f0000001f in V0.

The `sri` instruction shifts each lane in `Vs.t` to the right the specified number of bits, then logically ORs the *`n - imm`* HO bits of `Vd.t` into the result (replacing the 0s that were shifted in), then stores the result back into `Vd.t`, as shown in Figure 11-19.

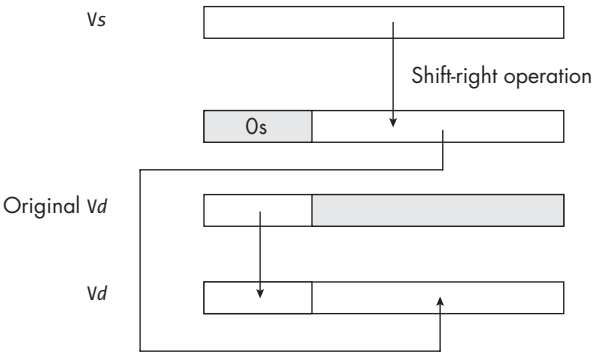


Figure 11-19: The `sri` instruction operation

The scalar versions of the `sli` and `sri` instructions have the following syntax:

```
sli Dd, Ds, #imm  // imm = 0 to 63
sri Dd, Ds, #imm  // imm = 1 to 64
```

These instructions operate on the LO 64 bits of the specified vector register (`Dn`) and zero out the HO 64 bits of the destination register.

11.6.5 Signed and Unsigned Shift Right

Because an arithmetic shift left and a logical shift left are essentially the same operation, the ARM uses a single instruction for both operations: `shl`. However, the logical and arithmetic shifts are different for right shifts. Therefore, the Neon instruction set provides two instructions, `sshr` and `ushr`, for signed and unsigned shift right (respectively, arithmetic shift right and logical shift right).

As noted in Chapter 2, a shift-left operation is the same as a multiplication by 2. Shift-right operations are approximately the same as a division by 2. I say *approximately* because the behaviors of signed and unsigned numbers are somewhat different. For example, when you shift the value 1

to the right one position, you get a 0 result. If you shift the signed value –1 (all 1 bits) to the right by using an arithmetic shift right, however, the result is –1. In one case, the shift rounds toward 0, while in the other it rounds away from 0. Neither case is particularly correct or incorrect, but not being able to choose the rounding direction can be a problem.

With scalar instructions, you can reverse this rounding effect by adding the carry flag to the result after the shift:

```
asr  x0, x0, #1
adc  x0, x0, xzr // -1 -> 0 and 1 -> 1
```

Because the vector operations don't track carries out of a shift in the carry flag, you don't have the option of correcting for this. The Neon instruction set therefore provides the rounding shift instructions `srshr` and `urshr`, which will add in the carry for you.

The syntax for the Neon shift-right instructions is shown here:

```
ushr  Vd.t, Vs.t, #imm  // Unsigned (logical) shift right
urshr Vd.t, Vs.t, #imm  // Unsigned rounding shift right
sshr  Vd.t, Vs.t, #imm  // Signed (arithmetic) shift right
srshr Vd.t, Vs.t, #imm  // Signed rounding shift right
```

The allowable types for the vector registers are the usual 8B, 16B, 4H, 8H, 2S, 4S, or 2D. The rounding variants (with the `r` as the second character in the mnemonic) add the carry flag back into the destination lane after the shift operation.

The `sshr`, `srshr`, `ushr`, and `urshr` instructions also have scalar versions:

```
sshr  Dd, Ds, #imm
srshr Dd, Ds, #imm
ushr  Dd, Ds, #imm
urshr Dd, Ds, #imm
```

These instructions operate on the LO 64 bits of the vector registers specified by `Dd` (destination) and `Ds` (source). The *`imm`* shift operand must be a value in the range 1 to 64. They will zero out the HO 64 bits of the corresponding `Vd` register. Otherwise, they are identical to their vector components.

11.6.6 Accumulating Shift Right

The accumulating shift-right instructions have the following syntax:

```
usra  Vd.t, Vs.t, #imm
ursra Vd.t, Vs.t, #imm
ssra  Vd.t, Vs.t, #imm
srsra Vd.t, Vs.t, #imm
```

These instructions are largely the same as the shift-right instructions, but they add their shifted values to the corresponding destination lanes (rather than just storing the shift lane values).

11.6.7 Narrowing Shift Right

The shrn, shrn2, rshrn, and rshrn2 instructions provide the converse operations to the shll and shll2 instructions. Rather than double the size of the operands when shifting, they halve ("narrow") the size. The syntax for these instructions is as follows

```
shrn   Vd.t1, Vs.t2, #imm
shrn2  Vd.t3, Vs.t4, #imm
rshrn  Vd.t1, Vs.t2, #imm
rshrn2 Vd.t3, Vs.t4, #imm
```

where:

t1 is 8B, 4H, or 2S

t2 is 8H, 4S, or 2D

t3 is 8B, 16B, 4H, 8H, 2S, or 4S

t4 is 8H, 4S, or 2D

The shrn instruction shifts each lane right the specified number of bits (shifting 0s in from the left); extracts the LO 8, 16, or 32 bits (depending on size of *t1*); and stores the result into the same lane number in the destination register. The shrn instruction ignores (truncates) any HO bits left in the shift operation that don't fit in the destination lane (which, recall, is half the size of the source lane). This instruction zeros out the HO 64 bits of the destination register.

The shrn2 instruction performs the exact same operation but stores the results in the HO 64 bits.

The rshrn and rshrn2 instructions do the same thing as shrn and shrn2, respectively, but round the shifted result before narrowing it. The rshrn instruction also clears the upper half of the destination register.

Because the narrowing shift-right instructions throw away all but the LO bits that fit in the destination lane, you might think a separate set of instructions must extract the HO bits after the shift operation. There's no need for such instructions, though; just add 8, 16, or 32 to your shrn, shrn2, rshurn, or rshrn2 shift count to extract the HO bits.

11.6.8 Saturating Shift Right with Narrowing

The standard narrowing shift instructions truncate any HO bits when narrowing the result to half the source lane size. The saturating shift-right instructions will saturate the shifted value if it does not fit in the destination lane. Table 11-4 gives the syntax for these instructions.

Table 11-4: Lane-by-Lane Saturating Shift Right with Narrowing Instructions

Mnemonic	Syntax	Description
uqshrn	uqshrn Vd.t1, Vs.t2, #imm	Unsigned shift right by *imm* bits with narrowing. Stores data into LO 64 bits of Vd.
uqrshrn	uqrshrn Vd.t1, Vs.t2, #imm	Unsigned shift right by *imm* bits with narrowing and rounding. Stores data into LO 64 bits of Vd.
sqshrn	sqshrn Vd.t1, Vs.t2, #imm	Signed shift right by *imm* bits with narrowing. Stores data into LO 64 bits of Vd.
sqrshrn	sqrshrn Vd.t1, Vs.t2, #imm	Signed shift right by *imm* bits with narrowing and rounding. Stores data into LO 64 bits of Vd.
sqshrun	sqshrun Vd.t1, Vs.t2, #imm	Signed shift right by *imm* bits with narrowing and saturation to an unsigned number. Stores data into LO 64 bits of Vd.
sqrshrun	sqrshrun Vd.t1, Vs.t2, #imm	Signed shift right by *imm* bits with narrowing, rounding, and saturation to an unsigned number. Stores data into LO 64 bits of Vd.
uqshrn2	uqshrn2 Vd.t3, Vs.t4, #imm	Unsigned shift right by *imm* bits with narrowing. Stores data into HO 64 bits of Vd.
uqrshrn2	uqrshrn2 Vd.t3, Vs.t4, #imm	Unsigned shift right by *imm* bits with narrowing and rounding. Stores data into HO 64 bits of Vd.
sqshrn2	sqshrn2 Vd.t3, Vs.t4, #imm	Signed shift right by *imm* bits with narrowing. Stores data into HO 64 bits of Vd.
sqrshrn2	sqrshrn2 Vd.t3, Vs.t4, #imm	Signed shift right by *imm* bits with narrowing and rounding. Stores data into HO 64 bits of Vd.
sqshrun2	sqshrun2 Vd.t3, Vs.t4, #imm	Signed shift right by *imm* bits with narrowing and saturation to an unsigned number. Stores data into HO 64 bits of Vd.
sqrshrun2	sqrshrun2 Vd.t3, Vs.t4, #imm	Signed shift right by *imm* bits with narrowing, rounding, and saturation to an unsigned number. Stores data into HO 64 bits of Vd.

Table 11-5 lists the legal types and lane counts for the saturating shift-right instructions appearing in Table 11-4.

Table 11-5: Saturating Shift-Right Types and Lane Counts

t	Legal types and lane counts
t1/t2	8B/8H, 4H/4S, or 2S/2D
t3/t4	16B/8H, 8H/4S, or 4S/2D

The instructions with the 2 suffix store their narrowed results into the HO 64 bits of the destination register. Those without this suffix will zero out the HO 64 bits of the destination register.

The uqrshrn, sqrshrn, uqrshrn2, and sqrshrn2 instructions round the shifted result before saturating the value (if saturation is necessary). Rounding consists of adding the last bit shifted out of the source lane back into the value.

The instructions with the s prefix operate on signed values, while those with the u prefix operate on unsigned values. Unsigned values saturate to all 1 bits (if the unsigned value will not fit in the destination lane size), but signed values will saturate to either a HO bit of 1 with other bits 0s, or a HO bit of 0 with all other bits containing 1s.

The sqrshrun and sqrshrun2 instructions do the following:

- Perform an arithmetic shift-right operation by the specified number of bits
- Round the result by adding the last bit shifted out back into the result
- Saturate the result to the maximum unsigned value (all 1 bits) if the result will not fit into the destination lane; negative values saturate to 0
- Store the saturated result into the destination lane

The sqrshrun instruction stores the results in the LO 64 bits of the destination register; sqrshrun2 stores the results in the HO 64 bits of the destination register.

These instructions also have scalar versions:

```
sqshrn Bd, Hs, #imm
sqshrn Hd, Ss, #imm
sqshrn Sd, Ds, #imm

uqshrn Bd, Hs, #imm
uqshrn Hd, Ss, #imm
uqshrn Sd, Ds, #imm

sqrshrn Bd, Hs, #imm
sqrshrn Hd, Ss, #imm
sqrshrn Sd, Ds, #imm

uqrshrn Bd, Hs, #imm
uqrshrn Hd, Ss, #imm
uqrshrn Sd, Ds, #imm

sqshrun Bd, Hs, #imm
sqshrun Hd, Ss, #imm
sqshrun Sd, Ds, #imm

sqrshrun Bd, Hs, #imm
sqrshrun Hd, Ss, #imm
sqrshrun Sd, Ds, #imm
```

Note that these instructions clear the upper bits (beyond the specified scalar register) of the underlying vector register.

11.6.9 Shift by a Variable Number of Bits

To shift a lane by a variable number of bits, use one of the following instructions

```
sshl   Vd.t, Vs.t, Vc.t
ushl   Vd.t, Vs.t, Vc.t
```

```
sqshl  Vd.t, Vs.t, Vc.t
uqshl  Vd.t, Vs.t, Vc.t
srshl  Vd.t, Vs.t, Vc.t
urshl  Vd.t, Vs.t, Vc.t
sqrshl Vd.t, Vs.t, Vc.t
uqrshl Vd.t, Vs.t, Vc.t
```

where *t* is the usual 8B, 16B, 4H, 8H, 2S, 4S, or 2D.

`Vc.t` holds the signed shift count in the LO byte. For positive values (in the range 0 to 0x7F), the instruction shifts the bits in a lane the number of bit positions to the left. For negative values (0xFF to 0x80; –1 to –128), the instruction shifts the bits to the right, despite using a `shl` mnemonic. See Table 11-6 for legal ranges when specifying the shift count by using a register.

Table 11-6: Legal `Vc.t` Shift Ranges

Type	Unsigned (SHL)	Signed (SHR)
8B/16B	0 to 7	–1 to –7
4H/8H	0 to 15	–1 to –15
2S/4S	0 to 31	–1 to –31
2D	0 to 63	–1 to –63

Values outside the ranges listed in Table 11-6 will produce the results shown in Table 11-7.

Table 11-7: Result of Shift If Count Exceeds Allowable Range

Shift instruction	Positive count, positive overflow	Positive count, negative overflow	Negative count, positive value	Negative count, negative value
`sshl`	0	0	0	–1 (all 1 bits)
`ushl`	0	0	0	0
`sqshl`	HO bit 0, all others 1 (for example, 0x7F)	HO bit 1, all others 0 (for example, 0x80)	0	–1 (all 1 bits)
`uqshl`	All 1 bits (for example, 0xff)	All 1 bits (for example, 0xff)	0	0
`srshl`	0	0	0	0 (–1 + carry)
`urshl`	0	0	0	1 (0 + carry)
`sqshl`	HO bit 0, all others 1 (for example, 0x7F)	HO bit 1, all others 0 (for example, 0x80)	0	–1 (all 1 bits)
`uqshl`	–1 (all 1 bits)	–1 (all 1 bits)	0	0
`sqrshl`	HO bit 0, all others 1 (for example, 0x7F)	HO bit 1, all others 0 (for example, 0x80)	0	0 (–1 + carry)
`uqrshl`	–1 (all 1 bits)	–1 (all 1 bits)	0	1 (0 + carry)

Using `shf` (for shift) in these instructions would probably have been a better choice then `shl`, since that name better matches the operation. Just keep in mind that the value in the LO byte of `Vc.t` is a signed integer and negative values indicate a right shift.

The Neon `shl` instruction also has some scalar saturating versions

```
sqshl    Rd, Rs, Rc
uqshl    Rd, Rs, Rc
sqrshl   Rd, Rs, Rc
uqrshl   Rd, Rs, Rc
```

where *R* represents one of the scalar register names (B, H, S, or D). These instructions shift the value in the scalar register `Rs` the number of bit positions specified by the LO byte of `Rc` and store the shifted result in `Rd`. `Rc` is treated as a signed number; positive values shift `Rs` left, while negative values shift `Rs` right. If an overflow (signed or unsigned, as appropriate) occurs during the shift, these instructions set `Rd` to the maximum positive signed or unsigned value.

If the shift count is negative for the `sqshl` instruction, the CPU performs an arithmetic shift-right operation, which will replicate the HO bit when shifting to the right. Positive (and 0) source values will saturate to 0, and negative source values will saturate to –1 (all 1 bits).

The `sqrshl` and `uqrshl` instructions are special rounding versions of the saturating shift instructions. During a shift-right operation (that is, when `Rc` is negative), these instructions round the result by adding 1 if the last bit shifted out was a 1 bit.

11.7 SIMD Arithmetic Operations

The Neon instruction set includes several common arithmetic operations, including addition, subtraction, and multiplication. The only surprise is that there is no division operation; instead, you'll have to compute the reciprocal and multiply by that value (using the instructions provided to estimate reciprocals).

11.7.1 SIMD Addition

Neon provides a wide set of instructions that add lanes (ignoring overflow), add and saturate (when overflow occurs), or perform horizontal additions.

11.7.1.1 Vector Addition

The Neon instruction set provides several instructions you can use to add integer and floating-point values in lanes within the vector registers, as listed in Table 11-8. These instructions compute `Vd = Vl + Vr`, where `Vd` is the destination, `Vl` is the left operand, and `Vr` is the right operand.

Table 11-8: Neon Addition Instructions

Instruction mnemonic	Syntax	Description
add	`add Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane integer sum
fadd	`fadd Vd.t2, Vl.t2, Vr.t2`	Computes lane-by-lane floating-point sum
sqadd	`sqadd Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane signed integer sum, with saturation
uqadd	`uqadd Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane unsigned integer sum, with saturation
saddl	`saddl Vd.t3, Vl.t4, Vr.t4`	Computes lane-by-lane signed integer sum, with long extension
uaddl	`uaddl Vd.t3, Vl.t4, Vr.t4`	Computes lane-by-lane unsigned integer sum, with long extension
saddl2	`saddl2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane signed integer sum, with long extension
uaddl2	`uaddl2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane unsigned integer sum, with long extension
saddw	`saddw Vd.t3, Vl.t3, Vr.t4`	Computes lane-by-lane signed integer sum, with wide extension
uaddw	`uaddw Vd.t3, Vl.t3, Vr.t4`	Computes lane-by-lane unsigned integer sum, with wide extension
saddw2	`saddw2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane signed integer sum, with wide extension
uaddw2	`uaddw2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane unsigned integer sum, with wide extension
addhn	`addhn Vd.t4, Vl.t3, Vr.t3`	Computes lane-by-lane addition with narrowing
raddhn	`raddhn Vd.t4, Vl.t3, Vr.t3`	Computes lane-by-lane addition with rounding and narrowing
addhn2	`addhn2 Vd.t6, Vl.t5, Vr.t5`	Computes lane-by-lane addition with narrowing (uses HO bits)
raddhn2	`raddhn2 Vd.t6, Vl.t5, Vr.t5`	Computes lane-by-lane addition with rounding and narrowing (uses HO bits)
shadd	`shadd Vd.t7, Vl.t7, Vr.t7`	Computes lane-by-lane signed addition with halving
uhadd	`uhadd Vd.t7, Vl.t7, Vr.t7`	Computes lane-by-lane unsigned addition with halving
srhadd	`srhadd Vd.t7, Vl.t7, Vr.t7`	Computes lane-by-lane signed addition with rounding and halving
urhadd	`urhadd Vd.t7, Vl.t7, Vr.t7`	Computes lane-by-lane unsigned addition with rounding and halving
addp	`addp Vd.t1, Vl.t1, Vr.t1`	Adds vector pairwise
faddp	`faddp Vd.t2, Vl.t2, Vr.t2`	Adds vector floating-point pairwise
saddlp	`saddlp Vd.t8, Vl.t9`	Adds vector pairwise, signed long integer
uaddlp	`uaddlp Vd.t8, Vl.t9`	Adds vector pairwise, unsigned long integer
saddalp	`saddalp Vd.t8, Vl.t9`	Adds vector pairwise and accumulates, signed long integer
uaddalp	`uaddalp Vd.t8, Vl.t9`	Adds vector pairwise and accumulates, unsigned long integer

Table 11-9 lists the legal types for the addition and subtraction instructions.

Table 11-9: Legal Types for Vector Addition and Subtraction

t	**Legal types**
t1	8B, 16B, 4H, 8H, 2S, 4S, or 2D
t2	2S, 4S, or 2D
t3/t4	8H/8B, 4S/4H, or 2D/2S
t5/t6	8H/16B, 4S/8H, or 2D/4S
t7	8B, 16B, 4H, 8H, 2S, or 4S
t8/t9	4H/8B, 8H/16B, 2S/4H, 4S/8H, 1D/2S, or 2D/4S

The remainder of this section describes each of the addition instructions in Table 11-8 in greater detail.

The `add` instruction, with vector register operands, does a lane-by-lane addition. Any overflow (signed or unsigned) is ignored, with the sum holding the LO bits of the result. If the type is 8B, 4H, or 2S, the `add` instruction adds only the lanes in the LO 64 bits of the registers, zeroing out the HO 64 bits of the destination register. Figure 11-20 provides an example of a 16B lane-by-lane addition.

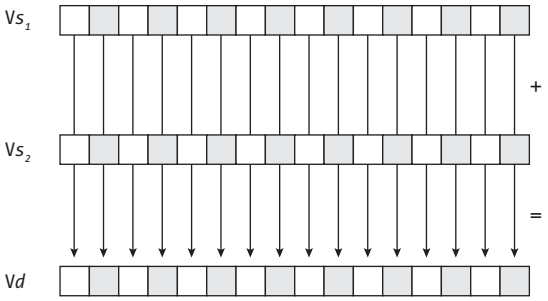


Figure 11-20: 16B lane-by-lane addition using `add` *Vd.16b, Vs1.16b, Vs2.16b*

The `fadd` instruction, with vector register operands, adds two- or four-lane single-precision values together, or a pair of double-precision floating-point values. With 2S types, the `fadd` instruction clears the HO 64 bits of the destination register.

The `sqadd` and `uqadd` instructions do a lane-by-lane addition (signed and unsigned, respectively), except they saturate their results in the case of overflow (or underflow, when adding signed numbers). As with `add`, those instructions that take 64-bit source operands produce a 64-bit result and zero out the HO 64 bits of the destination register.

The `saddl` and `uaddl` instructions take the lanes in the LO 64 bits of the source registers, sign- or zero-extend these values to twice their size,

compute the sum, and store the results in the full destination register (with double-sized lanes). The destination register type must be specified as twice the size of the source register types (see Figure 11-21). Because the sum of two n-bit numbers requires no more than $n + 1$ bits, these instructions will produce the correct result without any possibility of overflow or underflow.

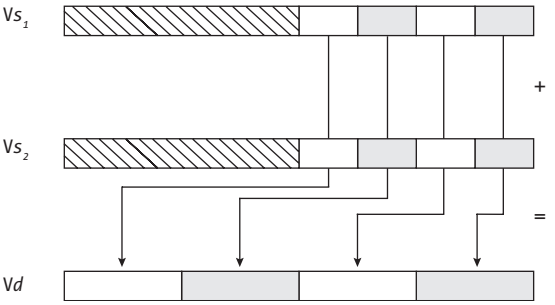


Figure 11-21: A `uaddl` operation (`uaddl Vd.4s, Vs1.4h, Vs2.4h`)

The `saddl2` and `uaddl2` instructions also sign- or zero-extend the values in the lanes in one-half of the source register and produce a sum in the full 128 bits of the destination register. However, the `saddl2` and `uaddl2` instructions compute the sum of the lanes in the HO 64 bits of the source registers (see Figure 11-22).

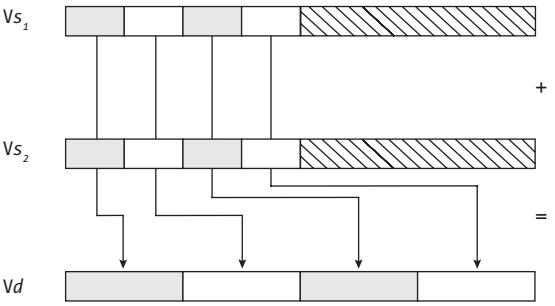


Figure 11-22: A `saddl2` operation (`saddl2 Vd.4s, Vs1.8h, Vs2.8h`)

Although the `saddl2` and `uaddl2` instructions' source operands are only 64 bits, you must specify the 128-bit types (16B, 8H, 4S) as the source type because the instruction retrieves the data from the upper 64 bits of a 128-bit value.

The `saddw`, `uaddw`, `saddw2`, and `uaddw2` instructions allow you to produce the sum of two operands whose sizes are different. The `saddw` and `uaddw` instructions expect the second source operand's type to be half the size of the first source and the destination operands' types, though you specify the same number of lanes for all three operands. These instructions will sign- or

zero-extend (respectively) the lanes in the second source operand to the size of the other two, compute the sum, and then store the data into the destination lanes (see Figure 11-23).

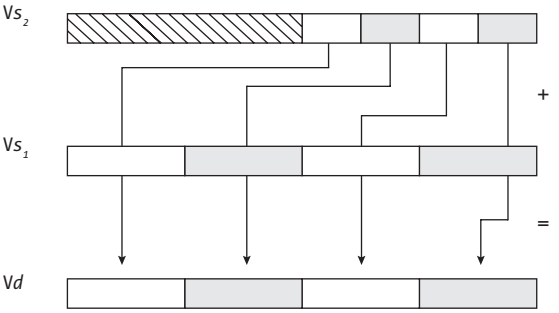


Figure 11-23: A `uaddw` operation (`uaddw Vd.4s, Vs1.4s, Vs2.4h`)

The `saddw2` and `uaddw2` instructions also sign- or zero-extend the second source operand, but they operate on the HO 64 bits rather than the LO 64 bits (see Figure 11-24). You must specify double the number of lanes for the second operand so that the instruction will operate on the full 128 bits of the second source operand.

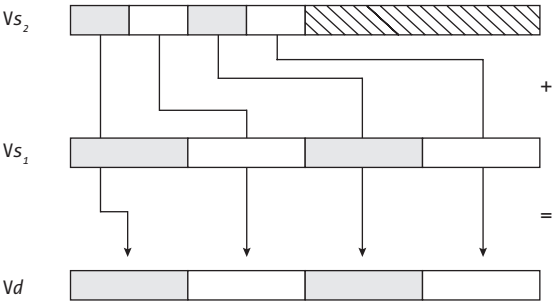


Figure 11-24: A `saddw2` operation (`saddw2 Vd.4s, Vs1.4s, Vs2.4h`)

Overflow (underflow) is possible when using the `saddw`, `uaddw`, `saddw2`, and `uaddw2` instructions (for example, when adding 0xFFFF with 0x01). These instructions will ignore the overflow and keep the LO bits of the result.

The `addhn` (vector add with narrowing) and `raddhn` (vector add, round, and narrow) instructions add the specified lanes together, then *narrow* the result by keeping only the HO bits. These instructions' destination type is half the size of the source types. For example, if you add half-word lanes together, the narrowing additions will keep only the HO byte of the results.

The `raddhn` instruction rounds the result before storing it into the destination register. If the LO half of the result contains a 1 in its HO bit

position, raddhn increments the HO byte by 1; otherwise, it returns the same result as addhn. Consider the following instruction:

```
raddhn  v0.8b, v1.8h, v2.8h
```

If V2 contained 0x00010001 and V1 contained 0xFE7FFE7F, then V0 would contain 0xFFFF after execution of this instruction. Had V1 contained 0xFE7EFE7E, though, V0 would contain 0xFEFE afterward.

Overflow can still occur during the execution of addhn and raddhn. Adding half words 0xFFFF and 0x0001 together will produce 0x00 in the corresponding destination byte lane.

The addhn2 and raddhn2 instructions also compute *add and narrowing* (with rounding, if specified); however, they store their results in the HO 64 bits of the destination register and leave the LO 64 bits of the destination unchanged. Because these instructions operate on the HO 64 bits of the destination, the destination's lane count must be twice that of the source registers. For example

```
addhn2  v0.16b, v1.8h, v2.8h
```

adds the LO 8 half words of V1 and V2 and stores the HO 8-bit result of each lane addition into the HO 8 bytes of V0 (leaving the LO 8 bytes untouched). You must specify the destination register's type as 16B, even though this instruction stores only 8 bytes into the register.

The shadd, uhadd, srhadd, and urhadd instructions add a pair of lanes together, shift right by 1 (with optional rounding, for those instructions containing an r), and store the result into the destination lane. As usual, the instructions beginning with an s handle signed values, while the instructions beginning with a u handle unsigned values. Because an addition of n bits never produces more than $n + 1$ bits, and a division by 2 is the same as a shift right by 1 bit, these instructions never produce an overflow. Consider the addition of the two largest single-byte values, 0xFF + 0xFF = 0x1FE. Shifting this sum to the right 1 bit gives you 0xFF, which fits just fine into 8 bits. Even with rounding, overflow will not occur.

These instructions are especially handy for processing digital audio. For example, suppose you want to mix together two 16-bit audio tracks. Simply summing the hwords from the two tracks will boost the volume by 3 decibels (dB) (equivalent to doubling the digital value). Halving the result after the sum reduces this volume increase by 3 dB. The urhadd instruction would be ideal for mixing these tracks as it would divide the result by 2, averaging the values of the two tracks.

11.7.1.2 Pairwise Addition

Thus far, all the addition operations have operated on corresponding lanes in the source operands, producing a result that the instructions store in the same lane in the destination register. This is known as a *vertical addition* because the data flows vertically from register to register, as shown previously

in Figure 11-20. On occasion, you may want to produce the sum of adjacent elements within a vector rather than the elements in corresponding lanes of two vectors (horizontal addition). You can accomplish this with the *pairwise addition* instructions from Table 11-8.

The pairwise addition instructions, as their name suggests, add adjacent pairs of lanes in vectors. Because the result requires half the number of lanes that are present in the source, the pairwise additions produce a single vector result from two source registers. Consider the following example that pairwise-adds the half words in V1 and V2, producing the pairwise sum in V0:

```
addp  v0.4s, v2.4s, v1.4s
```

This instruction computes the following results:

```
V0[0] = V2[0] + V2[1]
V0[1] = V2[2] + V2[3]
V0[2] = V1[0] + V1[1]
V0[3] = V1[2] + V1[3]
```

Figure 11-25 diagrams this operation.

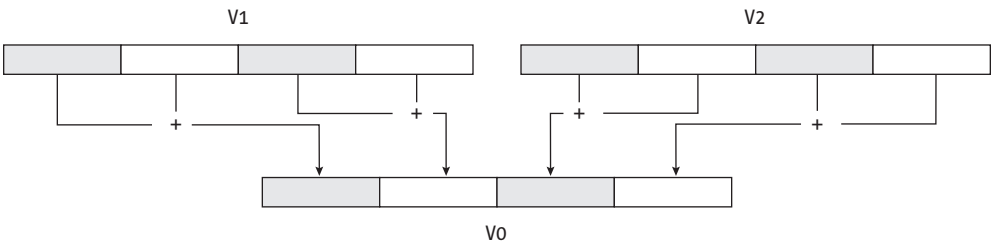


Figure 11-25: The `addp v0.4s, v2.4s, v1.4s` instruction

This instruction also has a floating-point version that adds adjacent single- or double-precision values in a pair of vectors: `faddp`. For example, the following instruction performs the same operation as the previous `addp` integer example but adds adjacent single-precision floating-point values rather than 32-bit integer values:

```
faddp  v0.4s, v1.4s, v2.4s
```

The `addp` instruction ignores any overflow during the addition. To produce a correct result, use the `saddlp` and `uaddlp` instruction (signed and unsigned pairwise add long) to sign- or zero-extend the lane values prior to the addition. The syntax for these two instructions is different from that of the other addition instructions: there are only two register operands (a source and a destination register). For example, because the following instruction doubles the size of the result to place in the destination

operand and sums adjacent elements of the source operand, there is no need for a second register operand:

```
saddlp  v0.2d, v1.4s
```

Note that the destination register type size must be twice that of the source, and the number of lanes must be half that of the source.

The `uaddalp` and `saddalp` instructions are functionally similar to `uaddlp` and `saddlp`, but rather than simply storing the pairwise sum into the destination lanes, they add the sum to the value already present in the destination lanes.

11.7.1.3 Vector Saturating Accumulate

The Neon instruction set includes two instructions that sum the lanes of a source vector into the corresponding lanes of a destination vector. The instructions are

```
usqadd  Vd.t, Vs.t  // Add lanes of Vs to Vd.
suqadd  Vd.t, Vs.t  // Add lanes of Vs to Vd.
```

where *t* is 8B, 4H, or 2S when operating on the LO 64 bits of the registers, or 16B, 8H, 4S, or 2D when operating on all 128 bits. The 64-bit variants will clear the HO 64 bits of *Vd*.

These instructions are unusual in that they allow you to add (with saturation) an unsigned input into a signed value or add a signed number into an unsigned value (usually instructions operate on only one type of data).

The `usqadd` instruction adds a signed value in the source lanes to the unsigned value in the corresponding destination lanes. Should the sum exceed the maximum (unsigned) value for the destination lane's size, this instruction will saturate the lane to the maximum value. Should the sum go negative, this instruction saturates the destination lane to 0. For example, if a half-word destination lane contains 0xFFF0 and the corresponding source lane contains 0xFF, the `usqadd` instruction (with a 4H or 8H type) will produce 0xFFFF in the destination lane. On the other hand, if the destination lane contains 0x08 and the source lane contains 0xFFF0 (–16), then their sum will produce 0 in the destination lane.

The `suqadd` instruction is the converse operation: it adds an unsigned source operand to a signed destination operand, saturating to the maximum signed value. For example, if a destination half-word lane contains 0x7FF0 and the corresponding source lane contains 0x00FF, their sum will produce 0x7FFF, the maximum signed value. Note that if the destination operand contains 0xFFFF (–1) and the source operand is 0x0002, you wind up with 0x0001 in the destination lane (–1 + 2 = 1).

The `usqadd` and `suqadd` instructions also have scalar variants

```
usqadd  Rd, Rs   // Add Rs to Rd.
suqadd  Rd, Rs   // Add Rs to Rd.
```

where *Rd* and *Rs* are one of the scalar registers `B`*n*, `H`*n*, `S`*n*, or `D`*n*.

The suqadd instruction will always produce the maximum signed value when overflowing, as you can't reduce the value by adding an unsigned number to it.

11.7.1.4 Horizontal Add

The addv (add across vector) instruction produces the sum of all the lanes in a single source vector register and leaves the result in a scalar element of another vector register (this is known as *reduction*). The syntax for this instruction is as follows

```
addv Rd, Vs.t
```

where *R* is the destination register and is one of B*n*, H*n*, or S*n*. The legal vector register type and lane count depend on the scalar register; Table 11-10 lists the valid types.

Table 11-10: Valid Vector Register Types for addv

Scalar register (R*d*)	Valid lane count and types
B*d*	8B or 16B
H*d*	4H or 8H
S*d*	4S

This instruction is useful for summing up the elements of an array. Unfortunately, the destination scalar type must be the same as the source lanes' type, and any overflow is ignored. There is no instruction that will zero- or sign-extend the sum to a double-sized result. Therefore, it's advisable to zero- or sign-extend the vector elements to the next larger size prior to executing addv if overflow is possible. You can accomplish this by using the saddlp or uaddlp instruction to add adjacent pairs and sign- or zero-extend (respectively), then use the addv instruction to sum the resulting double-sized lanes.

The addvl instruction is part of the ARM *scalable vector extensions (SVE)*, which are beyond the scope of this book. While you might expect addvl to be a long version of the addv instruction, it actually does something completely different. See the ARM SVE documentation for more details.

11.7.1.5 Scalar Saturating and Scalar Pairwise Addition

The Neon instruction set also provides a couple of saturating scalar addition instructions

```
sqadd  Rd, Rs1, Rs2
uqadd  Rd, Rs1, Rs2
```

where *R* represents one of the scalar register names B, H, S, or D. These instructions operate on the 8-, 16-, 32-, or 64-bit signed or unsigned integer

values found in the LO bits of the specified V register (see Figure 11-2 for the correspondence between the V*n*, B*n*, H*n*, S*n*, and D*n* registers). These instructions do the same thing as their vector counterparts, except, of course, they operate only on a scalar value rather than doing a lane-by-lane vector operation.

The following are the scalar variants of the `addp` and `faddp` instructions:

```
addp   Dd, Vs.2D
faddp  Sd, Vs.2S
faddp  Dd, Vs.2D
```

Note the limited lane count and type support of these instructions.

The `addp` instruction ignores (discards) any overflow from the addition of the two dword elements from the source vector. The `addpl` and `addpl2` instructions have no scalar versions. Use the actual `addpl` and `addpl2` instructions (with a second vector containing 0s) if you need an extended-precision version of this instruction.

11.7.2 Subtraction

While there aren't quite as many Neon instructions for subtraction as there are for addition, most of the addition instructions have a subtraction complement. Table 11-11 provides the syntax for the various vector subtraction instructions and associated data types; these instructions generally compute `Vd` = `Vl` − `Vr` (exceptions as noted), where `Vd` = destination, `Vl` = left operand, and `Vr` = right operand.

Table 11-11: Neon Subtraction Instructions

Instruction mnemonic	Syntax	Description
`sub`	`sub Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane integer difference
`fsub`	`fsub Vd.t2, Vl.t2, Vr.t2`	Computes lane-by-lane floating-point difference
`uqsub`	`uqsub Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane unsigned integer subtraction with saturation
`sqsub`	`sqsub Vd.t1, Vl.t1, Vr.t1`	Computes lane-by-lane signed integer subtraction with saturation
`usubl`	`usubl Vd.t3, Vl.t4, Vr.t4`	Computes lane-by-lane unsigned long integer subtraction
`ssubl`	`ssubl Vd.t3, Vl.t4, Vr.t4`	Computes lane-by-lane signed long integer subtraction
`usubl2`	`usubl2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane unsigned long integer subtraction of the HO half of `Vr`
`ssubl2`	`ssubl2 Vd.t5, Vl.t6, Vr.t6`	Computes lane-by-lane signed long integer subtraction of the HO half of `Vr`
`usubw`	`usubw Vd.t3, Vl.t3, Vr.t4`	Computes lane-by-lane unsigned wide integer subtraction
`ssubw`	`ssubw Vd.t3, Vl.t3, Vr.t4`	Computes lane-by-lane signed wide integer subtraction
`usubw2`	`usubw2 Vd.t5, Vl.t4, Vr.t6`	Computes lane-by-lane unsigned wide integer subtraction involving the upper half of `Vl`

Instruction mnemonic	Syntax	Description
ssubw2	ssubw2 V*d.t5*, V*l.t5*, V*r.t6*	Computes lane-by-lane signed wide integer subtraction involving the upper half of V*l*
subhn	subhn V*d.t4*, V*l.t3*, V*r.t3*	Computes lane-by-lane subtraction with narrowing
rsubhn	rsubhn V*d.t4*, V*l.t3*, V*r.t3*	Computes lane-by-lane subtraction with rounding and narrowing
subhn2	subhn2 V*d.t6*, V*l.t5*, V*r.t5*	Computes lane-by-lane subtraction with narrowing (uses HO bits)
rsubhn2	rsubhn2 V*d.t6*, V*l.t5*, V*r.t5*	Computes lane-by-lane subtraction with rounding and narrowing (uses HO bits)
uhsub	uhsub V*d.t7*, V*l.t7*, V*r.t7*	Computes lane-by-lane unsigned subtraction with halving
shsub	shsub V*d.t7*, V*l.t7*, V*r.t7*	Computes lane-by-lane signed subtraction with halving

The behavior of these instructions is very similar to their addition counterparts, except, of course, that they subtract the values in the lanes rather than adding them. See the previous section for more details.

There is also a saturating scalar subtraction instruction:

```
sqsub  Rd, Rs1, Rs2
uqsub  Rd, Rs1, Rs2
```

That subtracts the two source scalar registers (B*n*, H*n*, S*n*, or D*n*), producing a scalar result.

11.7.3 Absolute Difference

In addition to the normal subtraction instructions, the Neon instruction set includes several instructions that compute the difference of the values in corresponding lanes and then compute the absolute value of this difference. These instructions are handy for computing distances and other vector (as in physics) calculations.

Table 11-12 lists the available absolute difference instructions. In the Syntax column, V*d* = destination, V*l* = left operand, and V*r* = right operand. Each instruction generally computes V*d* = abs(V*l* − V*r*), unless otherwise noted.

Table 11-12: Neon Absolute Difference Instructions

Instruction mnemonic	Syntax	Description
uabd	uabd V*d.t1*, V*l.t1*, V*r.t1*	Vector unsigned absolute difference; lanes contain unsigned values.
sabd	sabd V*d.t1*, V*l.t1*, V*r.t1*	Vector signed absolute difference; lanes contain signed values.

(continued)

Table 11-12: Neon Absolute Difference Instructions *(continued)*

Instruction mnemonic	Syntax	Description
uaba	uaba V*d*.*t1*, V*1*.*t1*, V*r*.*t1*	Vector unsigned absolute difference and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain unsigned values.
saba	saba V*d*.*t1*, V*1*.*t1*, V*r*.*t1*	Vector signed absolute difference and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain signed values.
uabdl	uabdl V*d*.*t2*, V*1*.*t3*, V*r*.*t3*	Vector unsigned absolute difference long; lanes contain unsigned values.
sabdl	sabdl V*d*.*t2*, V*1*.*t3*, V*r*.*t3*	Vector signed absolute difference long; lanes contain signed values.
uabal	uabal V*d*.*t2*, V*1*.*t3*, V*r*.*t3*	Vector unsigned absolute difference long and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain unsigned values.
sabal	sabal V*d*.*t2*, V*1*.*t3*, V*r*.*t3*	Vector signed absolute difference long and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain signed values.
uabdl2	uabdl2 V*d*.*t4*, V*1*.*t5*, V*r*.*t5*	Vector unsigned absolute difference long; lanes contain unsigned values. Uses HO 64 bits of V*1* and V*r*.
sabdl2	sabdl2 V*d*.*t4*, V*1*.*t5* V*r*.*t5*	Vector signed absolute difference long; lanes contain signed values. Uses HO 64 bits of V*1* and V*r*.
uabal2	uabal2 V*d*.*t4*, V*1*.*t5*, V*r*.*t5*	Vector unsigned absolute difference long and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain unsigned values. Uses HO 64 bits of V*1* and V*r*.
sabal2	sabal2 V*d*.*t4*, V*1*.*t5*, V*r*.*t5*	Vector signed absolute difference long and accumulate; V*d* = V*d* + abs(V*1* – V*r*), where lanes contain signed values. Uses HO 64 bits of V*1* and V*r*.
fabd	fabd V*d*.*t6*, V*1*.*t6*, V*r*.*t6*	Vector floating-point absolute difference; lanes contain floating-point values.
fabd	fabd S*d*, S*1*, S*r*	Scalar single-precision floating-point absolute difference; S*d* = abs(S*1* – S*r*).
fabd	fabd D*d*, D*1*, D*r*	Scalar double-precision floating-point absolute difference; D*d* = abs(D*1* – D*r*).

Table 11-13 lists the legal types for the absolute difference instructions.

Table 11-13: Legal Types for Absolute Difference Instructions

t	Legal types
t1	8B, 16B, 4H, 8H, 2S, or 4S
t2/*t3*	8H/8B, 4S/4H, or 2D/2S
t4/*t5*	8H/16B, 4S/8H, or 2D/4S
t6	2S, 4S, or 2D

The uabd and sabd instructions compute the difference of each lane, take the absolute value of the difference, and store the result into the destination lane. Although the two instructions operate on unsigned and

signed source operands (respectively), the result is always an unsigned value. Effectively, these are just variants of the `sub` instruction that take the absolute value of the result. As long as you treat the result as an unsigned number (particularly in the case of the `sabd` instruction), these instructions will not produce an overflow (underflow).

The `uaba` and `saba` instructions add the absolute value of the difference to the corresponding lane in the destination register. If an overflow occurs (on the addition), these instructions store the LO bits (the lane size) into the corresponding destination lane. For signed operations, if `Vl` contains the most negative value (for example, 0x80 for a byte) and `Vr` contains 0, overflow occurs and the instruction winds up adding that most negative value to the destination lane.

The suffix-`l` and suffix-`l2` variants of these instructions compute a long calculation. The `uabdl` and `sabdl` instructions first zero- or sign-extend (respectively) the lane values to twice the lanes' size, then compute the absolute value of the difference and store the result into the corresponding double-sized lane. The `uabdl2` and `sabdl2` instructions do the same thing but grab the lane data from the HO 64 bits of the source operands (refer back to Figures 11-21 and 11-22 and substitute the appropriate instruction to see how this works).

The `fabd` instruction computes the absolute difference of two floating-point values. With vector register operands, it processes two double-precision or four single-precision floating-point values at a time. This instruction also supports scalar operations (single- or double-precision) by specifying the D*n* or S*n* registers as operands. Unfortunately, there is no floating-point absolute difference and accumulate instruction. You can simulate this instruction by following an `fabd` instruction with an `fadd` instruction (using a spare vector register to hold the temporary result from `fabd`).

11.7.4 Vector Multiplication

The Neon instruction set includes several instructions that compute the product of corresponding lanes in the vector register (both integer and floating-point products). The standard vector multiply instructions appear in Table 11-14. Note that `Vl` is the left source operand and `Vr` is the right source operand.

Table 11-14: Neon Vector Multiply Instructions

Mnemonic	Syntax	Description
`mul`	`mul Vd.t1, Vl.t1, Vr.t1`	Multiplication: `Vd` = `Vl` × `Vr`. Ignores overflow, keeps LO bits of result (lane by lane).
`mla`	`mla Vd.t1, Vl.t1, Vr.t1`	Multiply and accumulate: `Vd` = `Vd` + `Vl` × `Vr`. Ignores overflow, keeps LO bits of result (lane by lane).
`mls`	`mls Vd.t1, Vl.t1, Vr.t1`	Multiply and subtract: `Vd` = `Vd` – `Vl` × `Vr`. Ignores overflow, keeps LO bits of result (lane by lane).

(continued)

Table 11-14: Neon Vector Multiply Instructions *(continued)*

Mnemonic	Syntax	Description
smull	`smull Vd.t2, Vl.t3, Vr.t3`	Signed extended multiplication: Vd = Vl × Vr. Multiplies LO half of Vl by Vr and stores extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
umull	`umull Vd.t2, Vl.t3, Vr.t3`	Unsigned extended multiplication: Vd = Vl × Vr. Multiplies LO half of Vl by Vr and stores extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
smull2	`smull2 Vd.t4, Vl.t5, Vr.t5`	Signed extended multiplication: Vd = Vl × Vr. Multiplies HO half of Vl by Vr and stores extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
umull2	`umull2 Vd.t4, Vl.t5, Vr.t5`	Unsigned extended multiplication: Vd = Vl × Vr. Multiplies HO half of Vl by Vr and stores extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
smlal	`smlal Vd.t2, Vl.t3, Vr.t3`	Signed extended multiply and accumulate: Vd = Vd + Vl × Vr. Multiplies LO half of Vl by Vr and adds extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
umlal	`umlal Vd.t2, Vl.t3, Vr.t3`	Unsigned extended multiply and accumulate: Vd = Vd + Vl × Vr. Multiplies LO half of Vl by Vr and adds extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
smlal2	`smlal2 Vd.t4, Vl.t5, Vr.t5`	Signed extended multiply and accumulate: Vd = Vd + Vl × Vr. Multiplies HO half of Vl by Vr and adds extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
umlal2	`umlal2 Vd.t4, Vl.t5, Vr.t5`	Unsigned extended multiply and accumulate: Vd = Vd + Vl × Vr. Multiplies HO half of Vl by Vr and adds extended-precision result in the lanes of Vd (double lane size, lane-by-lane result).
smlsl	`smlsl Vd.t2, Vl.t3, Vr.t3`	Signed extended multiply and subtract: Vd = Vd – Vl × Vr. Multiplies LO half of Vl by Vr and subtracts this from the extended-precision value in the lanes of Vd (double lane size, lane-by-lane result).
umlsl	`umlsl Vd.t2, Vl.t3, Vr.t3`	Unsigned extended multiply and subtract: Vd = Vd – Vl × Vr. Multiplies LO half of Vl by Vr and subtracts this from the extended-precision value in the lanes of Vd (double lane size, lane-by-lane result).
smlsl2	`smlsl2 Vd.t4, Vl.t5, Vr.t5`	Signed extended multiply and subtract: Vd = Vd – Vl × Vr. Multiplies HO half of Vl by Vr and subtracts this from the extended-precision value in the lanes of Vd (double lane size, lane-by-lane result).
umlsl2	`umlsl2 Vd.t4, Vl.t5, Vr.t5`	Unsigned extended multiply and subtract: Vd = Vd – Vl × Vr. Multiplies HO half of Vl by Vr and subtracts this from the extended-precision value in the lanes of Vd (double lane size, lane-by-lane result).
fmul	`fmul Vd.t6, Vl.t6, Vr.t6`	Floating-point multiply: Vd = Vl × Vr. Multiplies the floating-point values in the Vl and Vr lanes and stores the product into the corresponding Vd lanes (lane by lane).
fmulx	`fmulx Vd.t6, Vl.t6, Vr.t6`	Floating-point multiply: Vd = Vl × Vr. Multiplies the floating-point values in the Vl and Vr lanes and stores the product into the corresponding Vd lanes (lane by lane). This variant handles the case where one source operand is 0 and the other is ±∞, producing the value ±2 (–2 if –∞, +2 otherwise).

Mnemonic	Syntax	Description
fmla	fmla Vd.t6, Vl.t6, Vr.t6	Floating-point multiply and accumulate: Vd = Vd + Vl × Vr (lane by lane).
fmls	fmls Vd.t6, Vl.t6, Vr.t6	Floating-point multiply and subtract: Vd = Vd – Vl × Vr (lane by lane).

Table 11-15 lists the legal types for the instructions appearing in Table 11-14.

Table 11-15: Legal Types for Vector Multiply Instructions

t	Types	Notes
t1	8B, 16B, 4H, 8H, 2S, or 4S	8B, 4S, and 2S operate only on LO 64 bits.
t2/t3	8H/8B, 4S/4H, or 2D/2S	*t3* lanes are taken from the LO 64 bits.
t4/t5	8H/16B, 4S/8H, or 2D/4S	*t5* lanes are taken from the HO 64 bits.
t6	2S, 4S, or 2D	

There are also pmul, pmull, and pmull2 (polynomial multiplication) instructions. However, polynomial multiplication isn't a traditional multiply operation, and a discussion of this is beyond the scope of this book. See the Arm documentation for more details on these instructions.

11.7.4.1 Vector Saturating Multiplication and Double

The vector saturating multiplication and double instructions build on the standard multiply, multiply and accumulate, and multiply and subtract instructions to produce an extended precision (long) result that doubles the product and saturates the result. The instructions in this set appear in Table 11-16 and compute Vd = saturate({Vd ± }(Vl × Vr) × 2), where Vd is the destination operand, Vl is the left operand, and Vr is the right operand; {Vd ± } indicates that Vd ± is an optional source operand.

Table 11-16: Vector Multiply and Double with Saturation Instructions

Mnemonic	Syntax	Description
sqdmull	sqdmull Vd.t1, Vl.t2, Vr.t2	Vd = (Vl × Vr) × 2 (lane by lane)
sqdmlal	sqdmlal Vd.t1, Vl.t2, Vr.t2	Vd = Vd + (Vl × Vr) × 2 (lane by lane)
sqdmlsl	sqdmlsl Vd.t1, Vl.t2, Vr.t2	Vd = Vd – (Vl × Vr) × 2 (lane by lane)
sqdmull2	sqdmull2 Vd.t3, Vl.t4, Vr.t4	Vd = (Vl × Vr) × 2 (lane by lane, HO 64 bits of source)
sqdmlal2	sqdmlal2 Vd.t3, Vl.t4, Vr.t4	Vd = Vd + (Vl × Vr) × 2 (lane by lane, HO 64 bits)
sqdmlsl2	sqdmlsl2 Vd.t3, Vl.t4, Vr.t4	Vd = Vd – (Vl × Vr) × 2 (lane by lane, HO 64 bits)
sqdmull	sqdmull Vd.t5, Vl.t6, Vr.t7[x]	Vd = (Vl × Vr) × 2 (Vl lanes × Vr[x] scalar)
sqdmlal	sqdmlal Vd.t5, Vl.t6, Vr.t7[x]	Vd = Vd + (Vl × Vr) × 2 (Vl lanes × Vr[x] scalar)

(continued)

Table 11-16: Vector Multiply and Double with Saturation Instructions *(continued)*

Mnemonic	Syntax	Description
sqdmlsl	sqdmlsl *Vd.t5*, *Vl.t6*, *Vr.t7*[*x*]	*Vd* = *Vd* – (*Vl* × *Vr*) × 2 (*Vl* lanes × *Vr*[*x*] scalar)
sqdmull2	sqdmull2 *Vd.t8*, *Vl.t9*, *Vr.t10*[*x*]	*Vd* = (*Vl* × *Vr*) × 2 (*Vl* lanes × *Vr*[*x*] scalar, HO 64 bits)
sqdmlal2	sqdmlal2 *Vd.t8*, *Vl.t9*, *Vr.t10*[*x*]	*Vd* = *Vd* + (*Vl* × *Vr*) × 2 (*Vl* lanes × *Vr*[*x*] scalar, HO 64 bits)
sqdmlsl2	sqdmlsl2 *Vd.t8*, *Vl.t9*, *Vr.t10*[*x*]	*Vd* = *Vd* – (*Vl* × *Vr*) × 2 (*Vl* lanes × *Vr*[*x*] scalar, HO 64 bits)

The legal types and lane counts appear in Table 11-17.

Table 11-17: Legal Types and Lane Counts for Vector Multiply and Double with Saturation

t	Types and lane counts
t1/*t2*	4S/4H, or 2D/2S
t3/*t4*	4S/8H, or 2D/4S
t5/*t6*/*t7*	4S/4H/H, or 2D/2S/S
t8/*t9*/*t10*	4S/8H/H, or 2D/4S/S

These instructions all sign-extend their source operands to twice their size and multiply them to produce a product. They then multiply this product by 2. The standard multiply variants saturate and store this product into the corresponding destination lane. The multiply and accumulate variants add the product (multiplied by 2) to the destination and saturate the result. The multiply and subtract variants subtract the product (multiplied by 2) from the destination and saturate the result.

The instructions without a `2` suffix extract their lanes from the LO 64 bits of the first source register, while those with a `2` suffix extract their lanes from the HO 64 bits of the second source register.

The last six instructions in Table 11-16 multiply the lanes in `Vl` by a scalar value selected from one of the lanes in `Vr` (selected by the [*x*] index operator). Here, *x* must be an appropriate value for the source type (0 to 7 for bytes, 0 to 3 for half words, or 0 to 1 for words). For the last six forms, if *t10* is H, then the `Vr` register number (*r*) must be in the range 0 to 15.

A couple of "short" versions of the `sqdmul*` instructions don't double the type size in the destination register: `sqdmulh` and `sqrdmulh`. These instructions also multiply their source operands, double the result, and saturate it. However, they store only the HO 64 bits of the result into the destination lane (with saturation and possible rounding). Table 11-18 lists these instructions.

Table 11-18: Saturating Multiply and Double Instructions, HO Bits

Mnemonic	Syntax	Description
sqdmulh	sqdmulh V*d.t1*, V*l.t1*, V*r.t1*	Lane-by-lane multiply, double, saturate, and keep HO half of product.
sqrdmulh	sqrdmulh V*d.t1*, V*l.t1*, V*r.t1*	Lane-by-lane multiply, double, round, saturate, and keep HO half of product.
sqdmulh	sqdmulh V*d.t2*, V*l.t3*, V*r.t4*[*x*]	Multiply the lanes in V*l* by the scalar selected by V*r*[*x*]; double the result, saturate, and keep the HO half of the product.
sqrdmulh	sqrdmulh V*d.t2*, V*l.t3*, V*r.t4*[*x*]	Multiply the lanes in V*l* by the scalar selected by V*r*[*x*]; double the result, round, saturate, and keep the HO half of the product.
sqdmulh	sqdmulh *Rd*, *Rl*, *Rr*	Scalar version of sqdmulh.
sqrdmulh	sqrdmulh *Rd*, *Rl*, *Rr*	Scalar version of sqrdmulh.

In this table, *t1* is 4H, 8H, 2S, or 4S. For the 4H and 2S types, the instruction works only with the LO 64 bits of the registers; the 8H and 4S types use all 128 bits.

The type specification *t2/t3/t4* is 4H/4H/H, 8H/8H/H, 2S/2S/S, or 4S/S; the 4H and 2S types work with the LO 64 bits of the registers, and the 8H and 4S types work on all 128 bits of the registers. If the type is H, V*r*'s register number must be in the range 0 to 15.

If the [*x*] index appears after V*r.t4*, the instruction multiplies the lanes in V*l* by the scalar value extracted from lane *x* of V*r*, which must be an appropriate value for the source type (0 to 7 for bytes, 0 to 3 for half words, or 0 to 1 for words).

These instructions have two scalar variants. *R* (in *Rd*, *Rl*, and *Rr*) must be H or S. For example

```
sqdmulh h0, h1, h2
```

computes H0 = saturate(H1 × H2 × 2).

11.7.4.2 Vector Multiplication by a Scalar Element

The Neon instruction set provides several instructions that multiply all the elements of a vector by a single scalar value, as listed in Table 11-19.

Table 11-19: Vector Multiply by Scalar Instructions

Mnemonic	Syntax	Description
mul	mul V*d.t1*, V*l.t1*, V*r.t2*[*x*]	Multiply integer vector element by scalar value. Multiply each lane in V*l* by V*r*[*x*] (scalar value) and store the product into the corresponding lane in V*d* (that is, for each lane *i*, V*d*[*i*] = V*l*[*i*] × V*r*[*x*]).
mla	mla V*d.t1*, V*l.t1*, V*r.t2*[*x*]	Multiply vector elements by a scalar and accumulate. For each lane *i*, V*d*[*i*] = V*d*[*i*] + V*l*[*i*] × V*r*[*x*].

(continued)

Table 11-19: Vector Multiply by Scalar Instructions *(continued)*

Mnemonic	Syntax	Description
`mls`	`mls Vd.t1, Vl.t1, Vr.t2[x]`	Multiply vector elements by a scalar and subtract. For each lane *i*, Vd[*i*] = –*d*[*i*] – Vl[*i*] × Vr[*x*].
`smull`	`smull Vd.t3, Vl.t4, Vr.t5[x]`	Signed vector multiply by scalar, long. Sign-extend the (LO) lanes in Vl to twice their size, multiply by Vr[*x*], and store the result into the double-sized lane in Vd. Uses only the LO 64 bits of Vl.
`smlal`	`smlal Vd.t3, Vl.t4, Vr.t5[x]`	Signed vector multiply by scalar and accumulate, long. Similar to `smull`, but sums the product into Vd rather than just storing it into Vd.
`smlsl`	`smlsl Vd.t3, Vl.t4, Vr.t5[x]`	Signed vector multiply by scalar and subtract, long. Similar to `smull`, but subtracts the product from Vd rather than just storing it into Vd.
`smull2`	`smull2 Vd.t6, Vl.t7, Vr.t8[x]`	Signed vector multiply by scalar, long. Sign-extend the (HO) lanes in Vl to twice their size, multiply by Vr[*x*], and store the result into the double-sized lane in Vd. Uses only the HO 64 bits of Vl.
`smlal2`	`smlal2 Vd.t6, Vl.t7, Vr.t8[x]`	Signed vector multiply by scalar and accumulate, long (HO source). Similar to `smull2`, but sums the product into Vd rather than just storing it into Vd.
`smlsl2`	`smlsl2 Vd.t6, Vl.t7, Vr.t8[x]`	Signed vector multiply by scalar and subtract, long (HO source). Similar to `smull2`, but subtracts the product from Vd rather than just storing it into Vd.
`umull`	`umull Vd.t3, Vl.t4, Vr.t5[x]`	Unsigned vector multiply by scalar, long. Zero-extend the (LO) lanes in Vl to twice their size, multiply by Vr[*x*], and store the result into the double-sized lane in Vd. Uses only the LO 64 bits of Vl.
`umlal`	`umlal Vd.t3, Vl.t4, Vr.t5[x]`	Unsigned vector multiply by scalar and accumulate, long. Similar to `umull`, but sums the product into Vd rather than just storing it into Vd.
`umlsl`	`umlsl Vd.t3, Vl.t4, Vr.t5[x]`	Unsigned vector multiply by scalar and subtract, long. Similar to `umull`, but subtracts the product from Vd rather than just storing it into Vd.
`umull2`	`umull2 Vd.t6, Vl.t7, Vr.t8[x]`	Unsigned vector multiply by scalar, long. Zero-extend the (HO) lanes in Vl to twice their size, multiply by Vr[*x*], and store the result into the double-sized lane in Vd. Uses only the HO 64 bits of Vl.
`umlal2`	`umlal2 Vd.t6, Vl.t7, Vr.t8[x]`	Unsigned vector multiply by scalar and accumulate, long (HO source). Similar to `umull2`, but sums the product into Vd rather than just storing it into Vd.
`umlsl2`	`umlsl2 Vd.t6, Vl.t7, Vr.t8[x]`	Unsigned vector multiply by scalar and subtract, long (HO source). Similar to `umull2`, but subtracts the product from Vd rather than just storing it into Vd.
`fmul`	`fmul Vd.t9, Vl.t10, Vr.t11[x]`	Floating-point vector multiply by scalar. Multiply each lane in Vl by Vr[*x*] (scalar value) and store the product into the corresponding lane in Vd (that is, for each lane *i*, Vd[*i*] = Vl[*i*] × Vr[*x*]).

Mnemonic	Syntax	Description
`fmulx`	`fmulx Vd.t9, Vl.t10, Vr.t11[x]`	Like `fmul`, except it's a special variant that handles the case where one source operand is 0 and the other is ±∞. This produces the value ±2 (–2 if –∞, +2 otherwise).
`fmla`	`fmla Vd.t9, Vl.t10, Vr.t11[x]`	Floating-point vector multiply by scalar and accumulate. Multiply each lane in `Vl` by `Vr[x]` (scalar value) and add the product into the corresponding lane in `Vd` (that is, for each lane `i`, `Vd[i] = Vd[i] + Vl[i] × Vr[x]`).
`fmls`	`fmls Vd.t9, Vl.t10, Vr.t11[x]`	Floating-point vector multiply by scalar and subtract. Multiply each lane in `Vl` by `Vr[x]` (scalar value) and subtract the product from the corresponding lane in `Vd` (that is, for each lane `i`, `Vd[i] = Vd[i] – Vl[i] × Vr[x]`).

Table 11-20 lists the legal types and lane counts for the instructions in Table 11-19.

Table 11-20: Legal Types and Lane Counts for Vector Multiply by Scalar

t	Legal types and lane counts
t1/t2	4H/H, 8H/H, 2S/S, or 4S/S
t3/t4/t5	4S/4H/H or 2D/2S/S
t6/t7/t8	4S/8H/H, or 2D/4S/S
t9/t10/t11	2S/2S/S, 4S/S, or 2D/D

Figure 11-26 shows the basic operation of the `mul`, `mla`, `mls`, `fmul`, `fmla`, and `fmls` instructions.

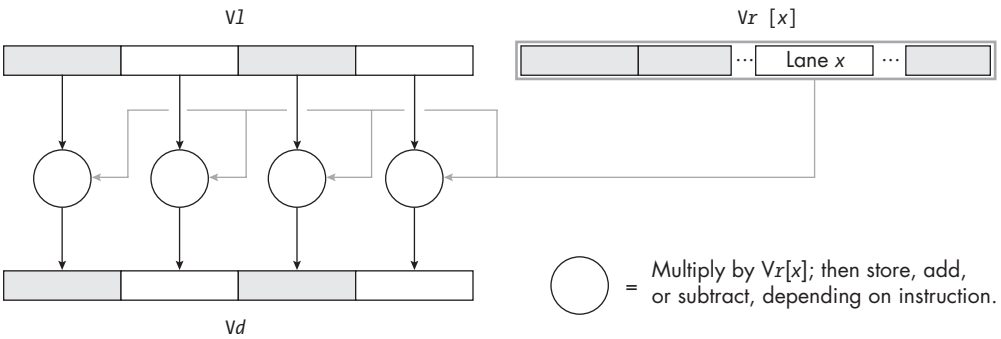


Figure 11-26: Vector multiply by scalar operation

Figure 11-27 shows the basic operation of the `smull`, `umull`, `smlal`, `umlal`, `smlsl`, and `umlsl` instructions.

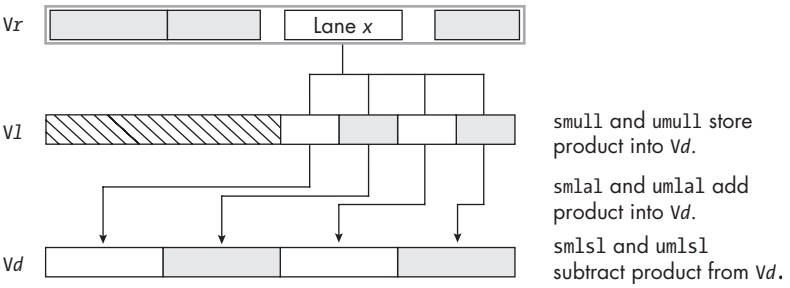


Figure 11-27: Vector multiply by scalar, long (LO bits)

Figure 11-28 shows the basic operation of the `smull2`, `umull2`, `smlal2`, `umlal2`, `smlsl2`, and `umlsl2` instructions.

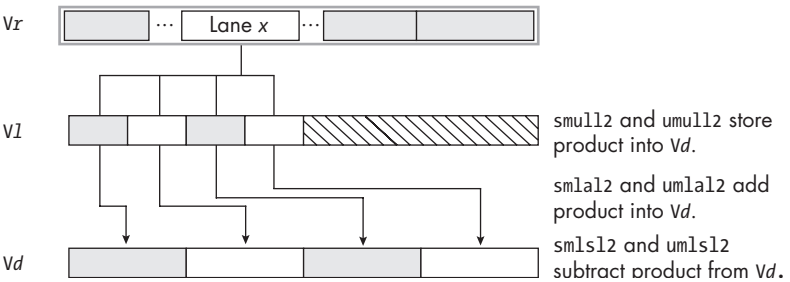


Figure 11-28: Vector multiply by scalar, long (HO bits)

Because the product of two n-bit numbers fits into $2n$ bits, the `smul`/`smul2` and `umul`/`umul2` instructions will not produce an overflow. However, keep in mind that an addition or subtraction after the multiplication could require an additional bit ($2n + 1$ bits). Should that occur, these instructions will ignore the overflow and keep the LO bits.

11.7.4.3 Scalar Multiplication by a Vector Element

The Neon instruction set provides variants of the `fmul` instructions that multiply a scalar register (`Sn` or `Dn`) by a vector element (`Vn[x]`), storing the result back into a scalar register. Table 11-21 lists the syntax for these instructions, where `Fl` is the left source operand and `Vr` is the right source operand.

Table 11-21: Floating-Point Scalar Multiplication by Vector Element Instructions

Mnemonic	Syntax	Description
`fmul`	`fmul Fd, Fl, Vr.t[x]`	`Fd = Fl × Vr.t[x]`
`fmulx`	`fmulx Fd, Fl, Vr.t[x]`	`Fd = Fl × Vr.t[x]`. Handles case where `Fl` = 0.0 and `Vr.t` is ±∞, which produces ±2.0.
`fmla`	`fmla Fd, Fl, Vr.t[x]`	`Fd = Fd + Fl × Vr.t[x]`
`fmls`	`fmls Fd, Fl, Vr.t[x]`	`Fd = Fd – Fl × Vr.t[x]`

Registers `Fd` and `Fl` are each one of the scalar floating-point registers (`Sn` or `Dn`). Type `t` must be a matching size (S or D). If the type is single-precision (`Sn`), then `Vr` must be a register in the range V0 to V15.

These multiplication instructions have no integer equivalents.

11.7.5 Vector Division

The Neon instruction set does not provide any instructions to perform integer division on vectors. It does, however, provide an instruction to perform floating-point division on lanes in a pair of vectors

```
fdiv Vd.t, Vl.t, Vr.t  // Computes Vd = Vl / Vr (lane by lane)
```

where *t* is 2S, 4S, or 2D. (A division by zero produces NaN in the destination lane.)

Because floating-point division is rather slow, especially when iterated over all the lanes, the Neon instruction set includes a pair of instructions that will compute the reciprocal of a floating-point value. Multiplication by a reciprocal is usually much faster than division. If you're dividing by a constant, you can precompute the reciprocal value at assembly time and use that (no runtime cost). If the value is a variable that you cannot compute at assembly time, you can use the `frecpe` instruction to *approximate* the reciprocals of all the lanes in a vector register

```
frecpe Vd.t, Vs.t
```

where *t* is 2S, 4S, or 2D (2S operates on the LO 64 bits of the registers).

There is a scalar version of `frecpe`

```
frecpe Rd, Rs
```

where *Rd* and *Rs* are either S*n* or D*n*.

The `frecpe` instruction produces a reciprocal approximation that is within 8 bits of the correct value—not great, but good enough for quick-and-dirty calculations. If you need better accuracy, use the `frecps` instruction (same syntax except for the mnemonic) to compute another step in the Newton-Raphson reciprocal approximation algorithm, using code like the following:

```
// Compute V0.4S = V1.4S / V2.4S by computing the reciprocal
// of V2 and multiplying V1 by this reciprocal value:

    frecpe  v3.4s, v2.4s         // Get first approximation.
    frecps  v0.4s, v1.4s, v3.4s  // *** Refinement step
    fmul    v3.4s, v3.4s, v0.4s  // *** Refinement step (cont.)

// Repeat "Refinement step" as many times as desired here.

    fmul    v0.4s, v1.4s, v3.4s  // Compute quotient.
```

The more times you repeat the refinement step, the more accurate your result will be. However, at some point, the cost of executing all these floating-point instructions will exceed the time consumed by a single `fdiv` instruction, so take care because using `frecps` has diminishing returns.

There is a `urecpe` instruction for estimating fixed-point reciprocals, but fixed-point arithmetic is beyond this scope of this book. To learn more, see the ARM Architecture Reference Manual, linked in section 11.15, "For More Information," on page 700.

11.7.6 Sign Operations

The Neon instruction set includes four instructions that allow you to negate or take the absolute values of the lanes in a vector register

```
abs   Vd.t1, Vs.t1
neg   Vd.t1, Vs.t1
sqabs Vd.t1, Vs.t1
sqneg Vd.t1, Vs.t1
fabs  Vd.t2, Vs.t2
fneg  Vd.t2, Vs.t2
```

where *t1* represents the usual integer types (8B, 16B, 4H, 8H, 2S, 4S, or 2D) and *t2* represents the usual floating-point types (2S, 4S, and 2D). The 8B, 4H, and 2S types reference only the LO 64 bits of the vector register.

The `abs` and `fabs` instructions compute the absolute values of each of the lanes in the source register, storing the result into the destination register. Obviously, `abs` works on (signed) integer values, while `fabs` works on floating-point values.

The `neg` and `fneg` instruction negate (change the sign of) the source lanes, leaving the negated result in the corresponding destination lane. As expected, `neg` works on signed integers, and `fneg` works on floating-point values.

The `sqabs` and `sqneg` instructions are special saturating variants of the `abs` and `neg` instructions that never overflow. The most negative value (for example, 0x80 for byte values) will overflow when you take its absolute value or negate it; in both cases, you wind up with the same value. The `sqabs` and `sqneg` instruction will produce the maximum positive value (for example, 0x7F for byte values) if you attempt to negate it or take its absolute value.

The `abs`, `neg`, `sqabs`, and `sqneg` instructions also have scalar versions, as shown in Table 11-22. For `abs` and `neg`, *Rd* and *Rs* can be only *Dn*; for `sqabs` and `sqneg`, *Rd* and *Rs* are one of the scalar registers *Bn*, *Hn*, *Sn*, or *Dn*.

Table 11-22: Scalar Sign Operations

Mnemonic	Syntax	Description
abs	abs *Rd*, *Rs*	*Rd* = abs(*Rs*)
neg	neg *Rd*, *Rs*	*Rd* = –*Rs*
sqabs	sqabs *Rd*, *Rs*	*Rd* = abs(*Rs*), saturated to maximum signed value
sqneg	sqneg *Rd*, *Rs*	*Rd* = –*Rs*, saturated to signed range

The instructions in Table 11-22 operate on the scalar value in the specified register.

11.7.7 Minimum and Maximum

The Neon instruction set provides several instructions that will select the minimum or maximum value from corresponding lanes in two vector registers and store that value into the corresponding lane of a destination register, as shown in Table 11-23.

Table 11-23: Vector Min and Max Instructions

Mnemonic	Syntax	Description
smin	smin V*d.t1*, V*l.t1*, V*r.t1*	V*d* = min(V*l*, V*r*) (signed integer values)
smax	smax V*d.t1*, V*l.t1*, V*r.t1*	V*d* = max(V*l*, V*r*) (signed integer values)
umin	umin V*d.t1*, V*l.t1*, V*r.t1*	V*d* = min(V*l*, V*r*) (unsigned integer values)
umax	umax V*d.t1*, V*l.t1*, V*r.t1*	V*d* = max(V*l*, V*r*) (unsigned integer values)
fmin	fmin V*d.t2*, V*l.t2*, V*r.t2*	V*d* = min(V*l*, V*r*) (floating-point values)
fmax	fmax V*d.t2*, V*l.t2*, V*r.t2*	V*d* = max(V*l*, V*r*) (floating-point values)
fminnm	fminnm V*d.t2*, V*l.t2*, V*r.t2*	V*d* = min(V*l*, V*r*) (floating-point values)
fmaxnm	fmaxnm V*d.t2*, V*l.t2*, V*r.t2*	V*d* = max(V*l*, V*r*) (floating-point values)

In this table, *t1* must be 8B, 16B, 4H, 8H, 2S, or 4S. If *t1* is 8B, 4H, or 2S, the instructions operate only on the lanes in the LO 64 bits of the vector registers; if it is 16B, 8H, or 4S, the instructions operate on all 128 bits of the vector registers.

The type *t2* must be 2S, 4S, or 2D. If it is 2S, the instructions operate only on the LO 64 bits of the vector registers; otherwise, they operate on the entire 128 bits.

The `fmin` and `fmax` instructions return NaN if either (or both) of the corresponding source lanes contain a NaN. The `fminnm` and `fmaxnm` instructions, on the other hand, return the numeric value if one lane contains a valid number and the other contains a NaN. If both lanes contain a valid floating-point value, all four instructions behave the same and return the minimum or maximum value (as appropriate).

11.7.7.1 Pairwise Minimum and Maximum

The minimum and maximum instructions also have pairwise variants, as shown in Table 11-24, where *t1* and *t2* are the same as for the lane-by-lane instructions.

Table 11-24: Pairwise Minimum and Maximum Instructions

Mnemonic	Syntax	Description	Operates on
sminp	sminp *Vd.t1*, *Vl.t1*, *Vr.t1*	*Vd* = pairwise_min(*Vl*, *Vr*)	Signed integers
smaxp	smaxp *Vd.t1*, *Vl.t1*, *Vr.t1*	*Vd* = pairwise_max(*Vl*, *Vr*)	Signed integers
uminp	uminp *Vd.t1*, *Vl.t1*, *Vr.t1*	*Vd* = pairwise_min(*Vl*, *Vr*)	Unsigned integers
umaxp	umaxp *Vd.t1*, *Vl.t1*, *Vr.t1*	*Vd* = pairwise_max(*Vl*, *Vr*)	Unsigned integers
fminp	fminp *Vd.t2*, *Vl.t2*, *Vr.t2*	*Vd* = pairwise_min(*Vl*, *Vr*)	Floating-point values
fmaxp	fmaxp *Vd.t2*, *Vl.t2*, *Vr.t2*	*Vd* = pairwise_max(*Vl*, *Vr*)	Floating-point values
fminnmp	fminnmp *Vd.t2*, *Vl.t2*, *Vr.t2*	*Vd* = pairwise_min(*Vl*, *Vr*)	Floating-point values
fmaxnmp	fmaxnmp *Vd.t2*, *Vl.t2*, *Vr.t2*	*Vd* = pairwise_max(*Vl*, *Vr*)	Floating-point values

The pairwise topology is the same as for the addp instruction (see Figure 11-29 for a uminp example).

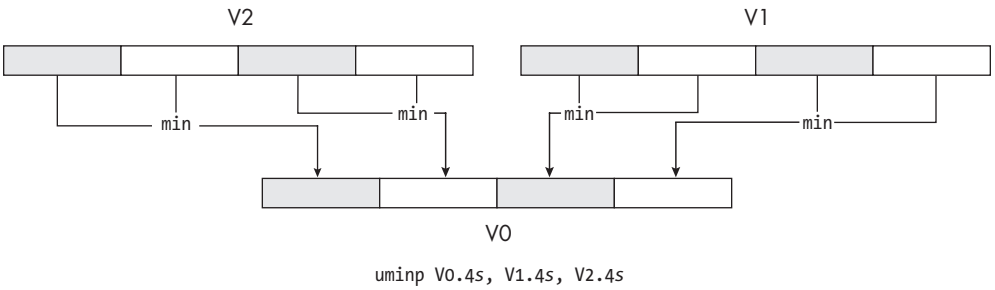


Figure 11-29: Pairwise minimum and maximum operations

There are also a set of pairwise-scalar floating-point minimum and maximum instructions, as shown in Table 11-25, where *Rd*/*t* must be S*n*/2S or D*n*/2D.

Table 11-25: Pairwise-Scalar Floating-Point Minimum and Maximum Instructions

Mnemonic	Syntax	Description
fmaxp	fmaxp *Rd*, Vs.*t*	*Rd* = max(Vs)
fmaxnmp	fmaxnmp *Rd*, Vs.*t*	*Rd* = max(Vs) (choose number over NaN)
fminp	fminp *Rd*, Vs.*t*	*Rd* = min(Vs)
fminnmp	fminnmp *Rd*, Vs.*t*	*Rd* = min(Vs) (choose number over NaN)

These instructions have no integer versions.

11.7.7.2 Horizontal Minimum and Maximum

The horizontal minimum and maximum instructions select the minimum or maximum value within a single vector, as shown in Table 11-26.

Table 11-26: Horizontal (Across Vector) Minimum and Maximum Instructions

Mnemonic	Syntax	Description
sminv	sminv *Rd*, Vs.*t1*	Extract minimum signed lane value from Vs and store into *Rd*.
smaxv	smaxv *Rd*, Vs.*t1*	Extract maximum signed lane value from Vs and store into *Rd*.
uminv	uminv *Rd*, Vs.*t1*	Extract minimum unsigned lane value from Vs and store into *Rd*.
umaxv	umaxv *Rd*, Vs.*t1*	Extract maximum unsigned lane value from Vs and store into *Rd*.
fminv	fminv *Sd*, Vs.*t2*	Extract minimum real lane value from Vs and store into *Sd*.
fmaxv	fmaxv *Sd*, Vs.*t2*	Extract maximum real lane value from Vs and store into *Rd*.
fminnmv	fminnmv *Sd*, Vs.*t2*	Extract minimum real lane value from Vs and store into *Sd*.
fmaxnmv	fmaxnmv *Sd*, Vs.*t2*	Extract maximum real lane value from Vs and store into *Rd*.

In this table, *Rd*/*t1* is B/8B, B/16B, H/4H, H/8H, or S/4S. If *t1* is 8B, or 4H, the instruction operates only on the lanes in the LO 64 bits of Vs. For floating-point minimum and maximum, only single-precision operands are legal; *t2* must be 2S or 4S (operating on the LO 64 bits or the full 128 bits of the source register). As for the standard fmin and fmax instructions, the nm variants differ insofar as they return the numeric value if one of the operands is NaN.

11.8 Floating-Point and Integer Conversions

The Neon instruction set provides several instructions to convert between floating-point and integer (or fixed-point) formats. Section 6.9.4, "Floating-Point Conversion Instructions," on page 343 provided examples of these conversion instructions when operating on scalar registers; the following subsections present the vector equivalents.

11.8.1 Floating-Point to Integer

The Neon instruction set provides vector equivalents of the fcvt* instructions that convert floating-point values to their integer equivalents, as shown in Table 11-27.

Table 11-27: Floating-Point to Integer Conversion Instructions

Mnemonic	Syntax	Description
fcvtns	fcvtns V*d*.*t*, Vs.*t*	Round to nearest signed integer. Exactly one-half rounds to nearest even integer.
fcvtas	fcvtas V*d*.*t*, Vs.*t*	Round to nearest signed integer. Exactly one-half rounds away from zero.
fcvtps	fcvtps V*d*.*t*, Vs.*t*	Round toward +∞ (signed integer).
fcvtms	fcvtms V*d*.*t*, Vs.*t*	Round toward –∞ (signed integer).
fcvtzs	fcvtzs V*d*.*t*, Vs.*t*	Round toward 0 (signed integer).

(continued)

Table 11-27: Floating-Point to Integer Conversion Instructions *(continued)*

Mnemonic	Syntax	Description
`fcvtnu`	`fcvtnu V`*d*`.`*t*`, V`*s*`.`*t*	Round to nearest unsigned integer. Exactly one-half rounds to nearest even integer.
`fcvtau`	`fcvtau V`*d*`.`*t*`, V`*s*`.`*t*	Round to nearest unsigned integer. Exactly one-half rounds away from 0.
`fcvtpu`	`fcvtpu V`*d*`.`*t*`, V`*s*`.`*t*	Round toward +∞ (unsigned integer).
`fcvtmu`	`fcvtmu V`*d*`.`*t*`, V`*s*`.`*t*	Round toward –∞ (unsigned integer).
`fcvtzu`	`fcvtzu V`*d*`.`*t*`, V`*s*`.`*t*	Round toward 0 (unsigned integer).

In this table, *t* is 2S (which uses only the LO 64 bits of the vector registers), 4S, or 2D. The source operand is always assumed to contain floating-point values (single- or double-precision), and the destination lanes will receive signed or unsigned integer values (words or dwords). Note that when converting negative floating-point values to unsigned integers, the conversion saturates the conversion to 0.0.

The `fcvtz*` instruction also has some fixed-point variants:

```
fcvtzs Vd.t, Vs.t, #imm
fcvtzu Vd.t, Vs.t, #imm
```

The *imm* operand specifies the number of fractional bits to maintain in the fixed-point value (this must be 1 to 31 for single-precision or word types and 1 to 63 for double-precision or dword types). Because integer operations are somewhat faster than floating-point calculations, sometimes it is faster to convert operands to fixed-point, do a chain of calculations, then convert the result back to floating-point. However, this book doesn't cover fixed-point arithmetic in depth, so I won't discuss this technique any further. See section 11.15, "For More Information," on page 700 for additional information.

11.8.2 Integer to Floating-Point

The `ucvtf` and `scvtf` instructions convert 32- and 64-bit integers to single- and double-precision values, respectively. Their syntax is roughly the same as that of `fcvt*`:

```
scvtf Vd.t, Vs.t
ucvtf Vd.t, Vs.t
```

As with `fcvt*`, *t* must be 2S, 4S, or 2D (2S converts only the LO 64 bits).

Because double-precision values have only a 56-bit mantissa and single-precision values have only a 24-bit mantissa, you cannot exactly represent certain 32- and 64-bit integers as single- or double-precision floating-point values. In those cases, the `scvtf` and `ucvtf` instructions produce the closest approximation. However, keep in mind that executing `*cvtf` followed by an `fcvt*` instruction may not return the exact same integer.

11.8.3 Conversion Between Floating-Point Formats

The Neon instruction set provides three instructions that will convert a small floating-point format to a larger form, or a larger form to a smaller form. This is one of the few instructions in the ARM instruction set that supports half-precision (16-bit) floating-point numbers. Table 11-28 shows the available instructions.

Table 11-28: Floating-Point Conversion Instructions

Mnemonic	Syntax	Description Lane-by-lane conversion
`fcvtl`	`fcvtl Vd.t1, Vs.t2`	Convert from a smaller size to the next larger size by using the LO 64 bits of the source register.
`fcvtl2`	`fcvtl2 Vd.t3, Vs.t4`	Convert from a smaller size to the next larger size by using the upper 64 bits of the source register (does not affect the LO bits of the destination register).
`fcvtn`	`fcvtn Vd.t5, Vs.t6`	Convert from a larger size to a smaller size by using the LO 64 bits of the destination register.
`fcvtn2`	`fcvtn2 Vd.t7, Vs.t8`	Convert from a larger size to a smaller size by using the HO 64 bits of the destination register (does not affect the LO bits of the destination register).
`fcvtxn`	`fcvtxn Vd.2S, Vs.2D`	Like `fcvtn`, except rounding is different (see text).
`fcvtxn2`	`fcvtxn2 Vd.4S, Vs.2D`	Like `fcvtn2`, except rounding is different (see text).

The legal types and lane counts for the instructions in Table 11-28 appear in Table 11-29, where H = 16-bit half-precision floating-point, S = 32-bit single-precision floating-point, and D = 64-bit double-precision floating-point.

Table 11-29: Legal Types and Lane Counts for Floating-Point Conversions

t	Types and lane counts
t1/t2	4S/4H or 2D/2S
t3/t4	4S/8H or 2D/4S
t5/t6	4H/4S or 2S/2D
t7/t8	8H/4S or 4S/2D

Conversion from a smaller size to a larger size always produces an exact result. Conversion from a larger size down to a smaller size may require rounding the result to fit in the smaller size (worst case, overflow or underflow will occur if the larger value cannot be represented at all in the smaller floating-point format).

When rounding larger values to fit into a smaller format, the `fcvtn` and `fcvtn2` instructions use the standard IEEE-754 round-to-nearest-even algorithm. In some cases, this may not produce the best result. For example, it is generally better to round to the nearest odd when converting a half-precision value to a double-precision value (which requires two steps: convert half-precision to single-precision, then convert single-precision to double-precision). The `fcvtxn` and `fcvtxn2` instructions employ this non-IEEE rounding scheme to produce better results.

11.8.4 Floating-Point Values Rounded to the Nearest Integral

Certain algorithms require rounding a floating-point value to an integer but require the result to be maintained in the floating-point format. The `frint*` instructions listed in Table 11-30 provide this capability.

Table 11-30: Rounding a Floating-Point Value to an Integral Value

Mnemonic	Syntax	Description Lane-by-lane rounding operation
`frintn`	`frintn Vd.t, Vs.t`	Round to nearest integer. Exactly one-half rounds to nearest even integer.
`frinta`	`frinta Vd.t, Vs.t`	Round to nearest integer. Exactly one-half rounds away from 0.
`frintp`	`frintp Vd.t, Vs.t`	Round toward +∞.
`frintm`	`frintm Vd.t, Vs.t`	Round toward –∞.
`frintz`	`frintz Vd.t, Vs.t`	Round toward 0.
`frinti`	`frinti Vd.t, Vs.t`	Round using FPCR rounding mode.
`frintx`	`frintx Vd.t, Vs.t`	Round using FPCR rounding mode with exactness test.

In this table, *t* must be 2S, 4S, or 2D. If it is 2S, these instructions use only the LO 64 bits of the registers.

The `frintx` instruction generates a floating-point inexact result exception if the rounded result is not equal to the original source value. You won't normally use this instruction unless you have an appropriate exception handler in place.

11.9 Vector Square-Root Instructions

The Neon instruction set provides two instructions for computing the square root of a floating-point value and computing (and refining) an estimate of the reciprocal of the square root of a floating-point value. Table 11-31 lists these instructions.

Table 11-31: Vector Square Root Instructions

Mnemonic	Syntax	Description Lane-by-lane operation
`fsqrt`	`fsqrt V`*`d`*`.`*`t`*`, V`*`s`*`.`*`t`*	Compute square root of source and store into destination
`frsqrte`	`frsqrte V`*`d`*`.`*`t`*`, V`*`s`*`.`*`t`*	First step of Newton-Raphson approximation of the reciprocal of the square root
`frsqrts`	`frsqrts V`*`d`*`.`*`t`*`, V`*`s1`*`.`*`t`*`, V`*`s2`*`.`*`t`*	Additional steps of the Newton-Raphson approximation

In the table, *t* must be 2S, 4S, or 2D. If it's 2S, these instructions operate on the lanes in the LO 64 bits of the vector registers.

These three instructions also have scalar versions

```
fsqrt   Rd, Rs
frsqrte Rd, Rs
frsqrts Rd, Rs1, Rs2
```

where *Rd* and *Rs* are one of the floating-point scalar registers S*n* or D*n*.

Note that the `frsqrts` instruction multiplies corresponding floating-point values in the lanes of the two source registers, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the destination register.

11.10 Vector Comparisons

Vector comparisons are fundamentally different from normal (general-purpose register) comparisons. When comparing general-purpose registers (or even individual floating-point scalar values), the ARM CPU sets the condition codes based on the result of the comparison; the code following the comparison then tests those condition codes, using a conditional branch, for example. This scheme doesn't work when comparing vector elements because the CPU always performs multiple comparisons in parallel. There's only one set of condition codes, so the CPU cannot put the results of multiple comparisons into the condition codes, meaning vector comparisons require a different mechanism to make the comparison results available to the program.

Rather than a generic comparison that produces less than, greater than, or equal results simultaneously (in the condition codes), a vector comparison asks for a specific comparison, such as, "Are the elements of this vector greater than the elements of another vector?" The result is true or false for each lane-by-lane comparison. A vector comparison will store the true or false result into the corresponding lanes of a destination vector. Vector comparisons use all 0 bits in a lane to represent false and all 1 bits in a lane to represent true.

Neon has two general sets of vector comparison instructions: one for integer comparisons and another for floating-point comparisons. The following subsections discuss each of these forms.

11.10.1 Vector Integer Comparisons

Table 11-32 lists the generic vector integer compare instructions, where *t* is 8B, 16B, 4H, 8H, 2S, 4S, or 2D. For the 8B, 4H, and 2S types, these instructions operate only on the LO 64 bits of the registers and clear the HO 64 bits of the destination register.

Table 11-32: Vector Integer Comparison Instructions

Mnemonic	Syntax	Description Lane-by-lane comparison
cmeq	cmeq V*d*.*t*, V*l*.*t*, V*r*.*t*	Signed or unsigned comparison for equality
cmhs	cmhs V*d*.*t*, V*l*.*t*, V*r*.*t*	Unsigned comparison for greater than or equal (V*d* = V*l* ≥ V*r*)
cmhi	cmhi V*d*.*t*, V*l*.*t*, V*r*.*t*	Unsigned comparison for greater than (V*d* = V*l* > V*r*)
cmge	cmge V*d*.*t*, V*l*.*t*, V*r*.*t*	Signed comparison for greater than or equal (V*d* = V*l* ≥ V*r*)
cmgt	cmgt V*d*.*t*, V*l*.*t*, V*r*.*t*	Signed comparison for greater than (V*d* = V*l* > V*r*)

There is no cmne instruction. You can invert all the bits in the destination register (using the not instruction) if you need this comparison, or you can use 0 bits to imply true and 1 bits to imply false. Likewise, there are no cmls, cmlo, cmle, or cmlt instructions; you can derive these from cmgt, cmge, cmhs, or cmhi by reversing the operands.

These instructions have scalar variants, as shown in Table 11-33, where *Rd*, *Rl*, and *Rr* must be D*n*.

Table 11-33: Scalar Integer Comparison Instructions

Mnemonic	Syntax	Description Scalar register comparison
cmeq	cmeq *Rd*, *Rl*, *Rr*	Signed or unsigned comparison for equality
cmhs	cmhs *Rd*, *Rl*, *Rr*	Unsigned comparison for greater than or equal (*Rd* = *Rl* ≥ *Rr*)
cmhi	cmhi *Rd*, *Rl*, *Rr*	Unsigned comparison for greater than (*Rd* = *Rl* > *Rr*)
cmge	cmge *Rd*, *Rl*, *Rr*	Signed comparison for greater than or equal (*Rd* = *Rl* ≥ *Rr*)
cmgt	cmgt *Rd*, *Rl*, *Rr*	Signed comparison for greater than (*Rd* = *Rl* > *Rr*)

A special set of vector comparison instructions exists to compare the lanes of a single vector against 0. This saves setting up a register to contain all 0s for this common case. The available instructions perform only signed comparisons (it doesn't make much sense to compare unsigned values against 0). Table 11-34 lists these instructions.

Table 11-34: Signed Vector Comparisons Against 0

Mnemonic	Syntax	Description Lane-by-lane comparison against 0
cmeq	cmeq V*d*.*t*, V*l*.*t*, #0	Signed comparison for lanes equal to 0
cmge	cmge V*d*.*t*, V*l*.*t*, #0	Signed comparison for lanes greater than or equal to 0
cmgt	cmgt V*d*.*t*, V*l*.*t*, #0	Signed comparison for lanes greater than 0
cmle	cmle V*d*.*t*, V*l*.*t*, #0	Signed comparison for lanes less than or equal to 0
cmlt	cmlt V*d*.*t*, V*l*.*t*, #0	Signed comparison for lanes less than 0

The type *t* must be 8B, 16B, 4H, 8H, 2S, 4S, or 2D. For the 8B, 4H, and 2S types, these instructions operate only on the LO 64 bits of the registers and clear the HO 64 bits of the destination register. The only legal immediate constant is 0 with these instructions.

Table 11-35 lists the scalar versions of these instructions.

Table 11-35: Scalar Vector Comparisons Against 0

Mnemonic	Syntax	Description Scalar register comparison against 0
cmeq	cmeq *Rd*, *Rl*, #0	Signed comparison for register equal to 0
cmge	cmge *Rd*, *Rl*, #0	Signed comparison for register greater than or equal to 0
cmgt	cmgt *Rd*, *Rl*, #0	Signed comparison for register greater than 0
cmle	cmle *Rd*, *Rl*, #0	Signed comparison for register less than or equal to 0
cmlt	cmlt *Rd*, *Rl*, #0	Signed comparison for register less than 0

In this table, *Rd*, *Rl*, and *Rr* must be D*n*.

11.10.2 Vector Floating-Point Comparisons

You can also compare floating-point values in the vector registers' lanes. Table 11-36 lists the various fcm* instructions available for this purpose, where *t* is 2S, 4S, or 2D. If *t* is 2S, these instructions use only the LO 64 bits of the registers.

Table 11-36: Vector Floating-Point Comparison Instructions

Mnemonic	Syntax	Description Lane-by-lane comparison
fcmeq	fcmeq V*d*.*t*, V*l*.*t*, V*r*.*t*	Floating-point comparison for equality
fcmge	fcmge V*d*.*t*, V*l*.*t*, V*r*.*t*	Floating-point comparison (V*d* = V*l* ≥ V*r*)
fcmgt	fcmgt V*d*.*t*, V*l*.*t*, V*r*.*t*	Floating-point comparison (V*d* = V*l* > V*r*)

Table 11-37 lists variants of the `fcm*` instructions that compare the lanes in a vector register against 0.0, where *t* is 2S, 4S, or 2D. If *t* is 2S, these instructions use only the LO 64 bits of the registers.

Table 11-37: Vector Floating-Point Comparison Against 0.0

Mnemonic	Syntax	Description Lane-by-lane comparison against 0.0
`fcmeq`	`fcmeq V`*d*`.`*t*`, V`*1*`.`*t*`, #0`	Floating-point comparison for register equal to 0.0
`fcmge`	`fcmge V`*d*`.`*t*`, V`*1*`.`*t*`, #0`	Floating-point comparison for register greater than or equal to 0.0
`fcmgt`	`fcmgt V`*d*`.`*t*`, V`*1*`.`*t*`, #0`	Floating-point comparison for register greater than 0.0
`fcmle`	`fcmle V`*d*`.`*t*`, V`*1*`.`*t*`, #0`	Floating-point comparison for register less than or equal to 0.0
`fcmlt`	`fcmlt V`*d*`.`*t*`, V`*1*`.`*t*`, #0`	Floating-point comparison for register less than 0.0

Note that the immediate constant is 0 (versus 0.0), even though this is a floating-point comparison. The only legal operand for this instruction is `#0`.

As for the integer comparisons, the `fcm*` instructions provide a set of scalar instructions that also store true (all 1 bits) or false (all 0 bits) into the destination register (in contrast to the `fcmp` instructions that set the condition code flags). Table 11-38 lists the scalar versions of these instructions, where *Rd*, *R1*, and *Rr* must be `S`*n* or `D`*n*.

Table 11-38: Scalar Variants of the Vector Floating-Point Comparisons

Mnemonic	Syntax	Description Scalar register comparison (including against 0.0)
`fcmeq`	`fcmeq` *Rd*`,` *R1*`,` *Rr*	Floating-point comparison for equality
`fcmge`	`fcmge` *Rd*`,` *R1*`,` *Rr*	Floating-point comparison (*Rd* = *R1* ≥ *Rr*)
`fcmgt`	`fcmgt` *Rd*`,` *R1*`,` *Rr*	Floating-point comparison (*Rd* = *R1* > *Rr*)
`fcmeq`	`fcmeq` *Rd*`,` *R1*`, #0`	Floating-point comparison for register equal to 0.0
`fcmge`	`fcmge` *Rd*`,` *R1*`, #0`	Floating-point comparison for register greater than or equal to 0.0
`fcmgt`	`fcmgt` *Rd*`,` *R1*`, #0`	Floating-point comparison for register greater than 0.0
`fcmle`	`fcmle` *Rd*`,` *R1*`, #0`	Floating-point comparison for register less than or equal to 0.0
`fcmlt`	`fcmlt` *Rd*`,` *R1*`, #0`	Floating-point comparison for register less than 0.0

Neon has a couple of additional floating-point comparisons: `fac*` (vector floating-point absolute value compare). These instructions compare the absolute values of corresponding lanes in the source vector register and set the destination register accordingly. Table 11-39 lists these instructions.

Table 11-39: Floating-Point Absolute-Value Comparisons

Mnemonic	Syntax	Description
facge	facge V*d*.*t*, V*l*.*t*, V*r*.*t*	Floating-point comparison (V*d* = abs(V*l*) ≥ abs(V*r*))
facgt	facgt V*d*.*t*, V*l*.*t*, V*r*.*t*	Floating-point comparison (V*d* = abs(V*l*) > abs(V*r*))

There is no faceq instruction, since there's no need for one; just use fcmeq.

The fac* instructions also have scalar versions, listed in Table 11-40.

Table 11-40: Scalar Floating-Point Absolute-Value Comparisons

Mnemonic	Syntax	Description
facge	facge *Rd*, *Rl*, *Rr*	Floating-point comparison (*Rd* = abs(*Rl*) ≥ abs(*Rr*))
facgt	facgt *Rd*, *Rl*, *Rr*	Floating-point comparison (*Rd* = abs(*Rl*) > abs(*Rr*))

Note that *Rd*, *Rl*, and *Rr* must be S*n* or D*n*.

11.10.3 Vector Bit Test Instructions

The Neon instruction set provides a vector version of the tst instruction, cmtst, which has the following syntax

```
cmtst Vd.t, Vl.t, Vr.t
```

where *t* can be 8B, 16B, 4H, 8H, 2S, 4S, or 2D. If *t* is 8B, 4H, or 2S, this instruction operates only on the LO 64 bits of the source registers and clears the HO 64 bits of the destination register.

This instruction does a lane-by-lane logical AND operation between V*l* and V*r*. If the result is nonzero, it stores all 1 bits into the corresponding destination lane. Otherwise, it stores all 0s into the destination lane.

This instruction also has a scalar version:

```
cmtst Dd, Dl, Dr
```

This form supports only 64-bit register operands (D*n*).

11.10.4 Vector Comparison Results

Throughout your programming experience, including with HLLs, you've probably become accustomed to using the result of comparisons (such as Boolean expressions) to divert control flow (such as with an if/then/else statement). Vector comparisons present a completely different paradigm because the lanes in a comparison could all produce different results. What's the best way to deal with this?

First, consider the easy stuff: complex Boolean expressions involving ANDs, ORs, and other logical operators. Because the vector comparisons

compute convenient results (all 1s or all 0s), it's easy to compute something like this:

```
(V1 > V2) AND (V3 < V4)  // Assume unsigned 8H lanes.
```

Consider the following code:

```
cmhi v0.4h, v1.4h, v2.4h
cmhi v5.4h, v4.4h, v3.4h  // Same as cmlo v5.4h, v3.4h, v4.4h
and  v0.8b, v0.8b, v5.8b  // (assuming cmlo existed)
```

This leaves the result of the previous Boolean calculation (0x0000 or 0xFFFF) in the LO 64 bits of the V0 register (lanes 0 to 3; remember that the `and` instruction allows only 8B and 16B types, but they produce the same result as 4H would if it were a legal type).

You can use similar instruction sequences for OR, NOT, and any of the other logical vector operations (see Table 11-2). Such calculations will use complete Boolean evaluation.

NOTE *Short-circuit evaluation doesn't make sense for vector operations; see section 7.6.3, "Complex `if` Statements Using Complete Boolean Evaluation," on page 378 for more information on complete Boolean evaluation.*

If you absolutely, positively must branch to some locations based on the result of all the vector comparisons, keep in mind that the number of branch locations increases exponentially with the number of lanes (specifically, its 2^n different possibilities, where n is the number of lanes). For example, if you execute the following instruction

```
cmeq v0.4h, v1.4h, v2.4h
```

then the LO 64 bits of V0 will contain four Boolean values, yielding 16 combinations of the four comparisons. It's ugly, but you could create a jump table (see section 7.6.7.3, "Indirect Jump `switch` Implementation," on page 391) with 16 entries and then transfer control by using code like the following

```
ldr     q3, mask              // mask: .hword 0b1000, 0b100, 0b10, 0b1
cmeq    v0.4h, v1.4h, v2.4h
and     v0.8b, v0.8b, v3.8b   // Keep 1 bit of each lane.
addv    h0, v0.4h             // Merge the bits into H0.
umov    w0, v0.h[0]
adr     x1, JmpTbl
ldr     x0, [x1, x0, lsl #3]
add     x0, x0, x1
br      x0
```

where `mask` is

```
mask:   .dword  0x0008000400020001
```

and JmpTbl is a 16-entry .dword table with the offset to the labels to jump to, based on all the combinations of true and false for four lane comparisons. This code moves bit 0 of lane 0 into bit 0 of X0, bit 0 of lane 1 into bit 1 of X0, bit 0 of lane 2 into bit 2 of X0, and bit 0 of lane 3 into bit 3 of X0. This forms a 4-bit index (16 possible values) into JmpTbl.

Theoretically, you could create a jump table with 16 entries and write code to transfer control, but this would be so ugly it's not an option worth seriously considering.

Sometimes you don't need to know the particular configuration of matches in a vector comparison, only whether any matches exist at all. For example, suppose you were looking for a 0 byte in a string of characters (such as when computing the length of a zero-terminating string). You could load 16 characters at a time from the string and search for a 0 byte by comparing all of them against 0:

```
cmeq  v0.16b, v1.16b, #0  // Assume V1 contains 16 chars.
```

This instruction sets the particular lane in V0 to 0xFF, corresponding to any lane in V1 that contains a 0 byte. You can use the following sequence to check whether there were any 0 bytes at all:

```
cmeq v0.16b, v1.16b, #0  // Check for a 0 byte.
addv b0, v0.16b          // Sum comparison bytes.
umov w0, v0.b[0]         // Put sum where you can compare it.
cmp  w0, #0              // See if there were any 0 bytes.
beq  noZeroBytes         // No 0s, go fetch 16 more bytes.
```

In the SIMD paradigm, an ideal solution would be to do calculations in parallel and use masks to disable certain calculations. For example, suppose you have a vector of 32-bit integers to which you would like to add another vector's lanes, with the caveat that you don't want to add anything if a particular lane contains a value greater than 16 bits (0xFFFF). Consider the following code:

```
// Add V1.S to V0.S, but don't add a value to a particular
// lane in V0 if its value exceeds 0xFFFF.
//
// Note: Assume V2 contains 0x0000FFFF0000ffff0000FFFF0000ffff.

        // no cmls v3.4s, v0.4s, v2.4s, so use the following:

        cmhi    v3.4s, v2.4s, v0.4s

        and     v4.16b, v1.16b, v3.16b
        add     v0.4s, v0.4s, v4.4s
```

The and instruction sets dword lanes greater than 0xFFFF to 0 so that they will have no impact on the final lane sums.

11.11 A Sorting Example Using SIMD Code

Sorting data is a common vector solution. Listing 11-1 demonstrates a simple sort of eight elements by using a vectorized bitonic sorting algorithm.

```
// Listing11-1.S
//
// Demonstrates a simple bitonic sort
// of eight elements, using vector instructions

        #include    "aoaa.inc"
        .text

        .pool
ttlStr: wastr  "Listing 11-1"

// Format strings for printf:

fmtPV:  wastr   " %016llx %016llx"
nl:     wastr   "\n"

// Sample data to sort
// (eight unsigned 32-bit integers
// to be loaded into vector
// registers):

qval1:  .word   8, 7, 6, 4
qval2:  .word   3, 2, 1, 0

// Lookup tables for TBL instruction,
// used to move around integers
// within the vector registers
//
// TBL works with bytes; the following
// constants map 32-bit integers to
// a block of 4 bytes in the
// vector registers:

_a = 0x03020100
_b = 0x07060504
_c = 0x0b0a0908
_d = 0x0f0e0d0c

_e = 0x13121110
_f = 0x17161514
_g = 0x1b1a1918
_h = 0x1f1e1d1c

_e1 = 0x03020100    // Special case
_f1 = 0x07060504    // for single-
_g1 = 0x0b0a0908    // register lists
_h1 = 0x0f0e0d0c

lut1:   .word   _f1, _e1, _h1, _g1
lut2:   .word   _a, _f, _c, _h
```

```
lut3:   .word   _b, _e, _d, _g
lut4:   .word   _h1, _g1, _f1, _e1
lut5:   .word   _a, _b, _g, _h
lut6:   .word   _c, _d, _f, _e
lut7:   .word   _a, _e, _b, _f
lut8:   .word   _c, _g, _d, _h
lut9:   .word   _a, _e, _b, _f
lut10:  .word   _c, _g, _d, _h

// Usual function that returns
// a pointer to the name of this
// program in the X0 register:

        proc    getTitle, public
        lea     x0, ttlStr
        ret
        endp    getTitle

// printV
//
// Prints the two 128-bit values sitting
// on the top of the stack (prior to call)
// as hexadecimal values:

        proc    printV

        locals  p
        qword   p.v0
        qword   p.v1
        qword   p.v2
        qword   p.v3
        qword   p.v4
        byte    p.stk, 64
        endl    p

        enter   p.size

        // Preserve vector registers
        // (this program uses them):

        str     q0, [fp, #p.v0]
        str     q1, [fp, #p.v1]
        str     q2, [fp, #p.v2]
        str     q3, [fp, #p.v3]
        str     q4, [fp, #p.v4]

        // Print the first value on
        // the stack:

        ldr     w1, [fp, #16]
        ldr     w2, [fp, #20]
        ldr     w3, [fp, #24]
        ldr     w4, [fp, #28]
        lea     x0, fmtPV
        mstr    x1, [sp]
```

```
        mstr    x2, [sp, #8]
        mstr    x3, [sp, #16]
        mstr    x4, [sp, #24]
        bl      printf

        // Print the second value on
        // the stack:

        ldr     w1, [fp, #32]
        ldr     w2, [fp, #36]
        ldr     w3, [fp, #40]
        ldr     w4, [fp, #44]
        lea     x0, fmtPV
        mstr    x1, [sp]
        mstr    x2, [sp, #8]
        mstr    x3, [sp, #16]
        mstr    x4, [sp, #24]
        bl      printf

        lea     x0, nl
        bl      printf

        ldr     q0, [fp, #p.v0]
        ldr     q1, [fp, #p.v1]
        ldr     q2, [fp, #p.v2]
        ldr     q3, [fp, #p.v3]
        ldr     q4, [fp, #p.v4]
        leave
        endp    printV

// Here's the main program:

        proc    asmMain, public

        // Reserve stack space for parameters:

        locals  am
        byte    am.stk, 64
        endl    am

        enter   am.size

        // Load the values to sort
        // into V0 and V1:

        ldr     q0, qval1
        ldr     q1, qval2

        // Bitonic sort of eight
        // elements:

        // Step 1:

        umin    v2.4s, v0.4s, v1.4s
        umax    v3.4s, v0.4s, v1.4s
```

```
// Step 2:

ldr     q4, lut1
tbl     v3.16b, {v3.16b}, v4.16b

umin    v0.4s, v2.4s, v3.4s
umax    v1.4s, v2.4s, v3.4s
ldr     q4, lut2
tbl     v2.16b, {v0.16b, v1.16b}, v4.16b
ldr     q4, lut3
tbl     v3.16b, {v0.16b, v1.16b}, v4.16b

// Step 3:

umin    v0.4s, v2.4s, v3.4s
umax    v1.4s, v2.4s, v3.4s

// Step 4:

ldr     q4, lut4
tbl     v1.16b, {v1.16b}, v4.16b

umin    v2.4s, v0.4s, v1.4s
umax    v3.4s, v0.4s, v1.4s

ldr     q4, lut5
tbl     v0.16b, {v2.16b, v3.16b}, v4.16b
ldr     q4, lut6
tbl     v1.16b, {v2.16b, v3.16b}, v4.16b

uminp   v2.4s, v0.4s, v1.4s
umaxp   v3.4s, v0.4s, v1.4s

ldr     q4, lut7
tbl     v0.16b, {v2.16b, v3.16b}, v4.16b
ldr     q4, lut8
tbl     v1.16b, {v2.16b, v3.16b}, v4.16b

umin    v2.4s, v0.4s, v1.4s
umax    v3.4s, v0.4s, v1.4s

// Merge results:

ldr     q4, lut9
tbl     v0.16b, {v2.16b, v3.16b}, v4.16b
ldr     q4, lut10
tbl     v1.16b, {v2.16b, v3.16b}, v4.16b

str     q0, [sp]
str     q1, [sp, #16]
bl      printV

leave                       // Return to caller.
endp    asmMain
```

Here's the `build` command and the sample output for Listing 11-1:

```
% ./build Listing11-1
% ./Listing11-1
Calling Listing11-1:
00000000 00000001 00000002 00000003 00000004 00000006 00000007 00000008
Listing11-1 terminated
```

As you can see, this code properly sorted the data.

11.12 A Numeric-to-Hex-String Example Using SIMD Code

Listing 11-2 is a Neon example of something you should be familiar with: Chapter 9's `dtoStr` function that converts a dword into a hexadecimal string. It's a practical example of converting existing code to SIMD.

```
// Listing11-2.S

Usual source file information at the beginning of the file,
deleted for brevity

// dtoStr
//
//  Converts the dword passed in X1 to 16
//  hexadecimal digits (stored into buffer pointed
//  at by X0; buffer must have at least 24 bytes
//  available)

    .equ    convert0toA, 'A' - ('0' + 10)    // val + '0' to val + 'A'
    .equ    invert0ToA,  ~convert0toA & 0xFF // Invert the bits for BIC.

    proc    dtoStr
    stp     q0, q1, [sp, #-32]!     // Preserve registers.

    rev     x1, x1                  // Reverse bytes (for output).
    mov     v0.d[0], x1             // Set V0 to the LO nibbles
    rev     x1, x1                  // and V1 to the HO nibbles,
    ushr    v1.8b, v0.8b, #4        // also, restore X1.
    bic     v0.4h, #0xf0
    bic     v0.4h, #0xf0, lsl #8

    zip1    v0.16b, v1.16b, v0.16b  // Interleave the HO and LO nibbles.

    orr     v0.8h, #0x30            // Convert binary to ASCII,
    orr     v0.8h, #0x30, lsl #8    // note only 0-9 will be correct.

    movi    v1.16b, #'9'            // Determine which bytes
    cmgt    v1.16b, v0.16b, v1.16b  // should be A-F.

    bic     v1.8h, #invert0ToA      // Update bytes that should be A-F.
    bic     v1.8h, #invert0ToA, lsl #8
    add     v0.16b, v0.16b, v1.16b
```

```
    str     q0, [x0]                // Output the string.
    strb    wzr, [x0, #16]

    ldp     q0, q1, [sp], #32       // Restore registers.
    ret
    endp    dtoStr
```

Here's the `build` command and the sample output for Listing 11-2:

```
% ./build Listing11-2
% ./Listing11-2
Calling Listing11-2:
Value(fedcba9876543210) = string(FEDCBA9876543210)
Listing11-2 terminated
```

If you were to time this code, you'd find that it runs significantly faster than the scalar code in Chapter 9.

11.13 Use of SIMD Instructions in Real Programs

If you've read through this chapter but aren't sure how to apply SIMD instructions in real programs, don't feel like you're missing something. *SIMD* might as well stand for "*S*IMD *I*nstruction sets are *M*assively *D*ifficult to use." Although ARM's Neon instruction set is a bit more general-purpose than, say, Intel's SSE/AVX extensions, SIMD instructions were created to accelerate the execution of very specific algorithms. I like to paraphrase a line from this book's technical reviewer, Tony Tribelli, with respect to SIMD instructions' applicability: "I look at a particular SIMD instruction and ask myself, 'What benchmark was this instruction created for?'" That is, it often seems like SIMD instructions were added to the instruction set to make one benchmark program run faster and make the ARM CPU look better, though the instruction probably isn't useful outside the context of that benchmark.

In many respects, this statement is dead on: many SIMD instructions were created to solve one particular problem, and their applicability beyond that solution is merely coincidental. If you can't figure out how to use a given instruction, you probably haven't yet discovered the problem it was originally created to address.

If nothing else, the vector registers' lanes are a good place to store temporary values when you're already using all the general-purpose registers. You can use the `mov` instruction to copy data between a general-purpose register and a lane in a vector register; this is much faster than spilling the register to memory.

If you really want to use the Neon instruction set for high-performance computing, see section 11.15, "For More Information," on the next page, or search "SIMD parallel algorithms" or "SIMD vector algorithms" on the internet.

11.14 Moving On

This lengthy chapter covered many instructions. It began with a brief history of SIMD instruction sets; covered the vector registers on the ARM; discussed SIMD data types, lanes, and scalar operations; and then presented the Neon instruction set. This chapter ended with a pair of short examples that demonstrated bitonic sorting and numeric-to-hexadecimal-string conversion using the vector registers. These constitute useful ways to use the SIMD instructions on the ARM. Although SIMD instructions aren't often applicable in general programs, with a little thought you should be able to use them to speed up your code in certain situations.

A couple of the remaining chapters will employ SIMD instructions to improve performance. Chapter 12 uses Neon instructions to improve the performance of various bit operations, while Chapter 14 uses Neon instructions to implement fast memory move operations. You can apply what you've learned in this chapter to algorithms you've learned in previous chapters that could benefit from using SIMD instructions, such as the numeric-to-hex-string code. I'll leave it to you to implement these changes.

11.15 For More Information

- For more information on fixed-point arithmetic and other Neon instructions and data types, consult the ARM Architecture Reference Manual at *https://developer.arm.com/documentation/ddi0487/latest*.
- For more on ARM scalable vector extensions (SVEs), see the documentation at *https://developer.arm.com/documentation/102476/0001/SVE-architecture-fundamentals*.
- ARM offers a guide on implementing fixed-point arithmetic on 32-bit CPUs at *https://developer.arm.com/documentation/dai0033/a*.
- Those interested in vector sorting with ARM SVE can reference "A Fast Vectorized Sorting Implementation Based on the ARM Scalable Vector Extension (SVE)" by Bérenger Bramas for one implementation: *https://arxiv.org/pdf/2105.07782.pdf*.
- See the Vector Sorting Algorithms page of the CMSIS DSP Software Library for more on vector sorting with ARM: *https://arm-software.github.io/CMSIS_5/DSP/html/group__Sorting.html*.
- The ARM documentation also provides more detail on vector sorting on Neon at *https://developer.arm.com/documentation/den0018/a/NEON-Code-Examples-with-Optimization/Median-filter/Basic-principles-and-bitonic-sorting*.
- "Fast and Robust Vectorized In-Place Sorting of Primitive Types" by Mark Blacher, Joachim Giesen, and Lars Kühne at *https://drops.dagstuhl.de/opus/volltexte/2021/13775/pdf/LIPIcs-SEA-2021-3.pdf* covers quick sorting with vector instructions (written for AVX2, but easily translatable to Neon) using the bitonic sorting algorithm.

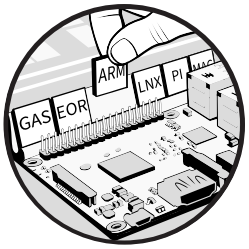
- See the master's thesis "A Study of the Use of SIMD Instructions for Two Image Processing Algorithms" by Eric Welch at *https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=3686&context=theses* for a discussion of SIMD image processing algorithms.
- Another master's thesis on SIMD signal processing algorithms is "Performance Optimization of Signal Processing Algorithms for SIMD Architectures" by Sharan Yagneswar, which can be found at *https://www.diva-portal.org/smash/get/diva2:1138490/FULLTEXT01.pdf*.

TEST YOURSELF

1. What is a lane?
2. What is the difference between a scalar instruction and a vector instruction?
3. What instruction would you use to move the data from a 32-bit general-purpose integer register into the LO 32 bits of a vector register?
4. What instruction would you use to move the data from a 64-bit general-purpose integer register into the LO 64 bits of a vector register?
5. What instruction would you use to arbitrarily rearrange the bytes in a vector register?
6. What instruction would you use to take a byte, word, dword, or qword in a general-purpose register and insert it somewhere in a vector register?
7. If you want to shift the two qwords in a vector register *n* bit positions to the left, what instruction would you use?
8. What is the difference between a vertical addition and a horizontal addition?
9. How can you set all the bits in the V0 register to 0?
10. How can you set all the bits in the V0 register to 1?

12

BIT MANIPULATION



Manipulating bits in memory is perhaps the feature for which assembly language is most famous. Even the C programming language, known for bit manipulation, doesn't provide as complete a set of bit-manipulation operations.

This chapter discusses how to manipulate strings of bits in memory and registers by using ARM assembly language. It begins with a review of the bit-manipulation instructions covered thus far, introduces a few new instructions, then reviews information on packing and unpacking bit strings in memory, which is the basis for many bit-manipulation operations. Finally, this chapter discusses several bit-centric algorithms and their implementation in assembly language.

12.1 What Is Bit Data, Anyway?

Bit manipulation refers to working with *bit data*, data types consisting of strings of bits that are noncontiguous or are not multiples of 8 bits long. Generally, these bit objects will not represent numeric integers, although I will not place this restriction on bit strings.

A *bit string* is a contiguous sequence of 1 or more bits. It does not have to start or end at any special point. For example, a bit string could start in bit 7 of one byte in memory and continue through to bit 6 of the next byte in memory. Likewise, a bit string could begin in bit 30 of W0, consume the upper 2 bits of W0, and then continue from bit 0 through bit 17 of W1. In memory, the bits must be physically contiguous (that is, bit numbers always increase except when crossing a byte boundary, and at byte boundaries the memory address increases by 1 byte). In registers, if a bit string crosses a register boundary, the application defines the continuation register, but the bit string always continues in bit 0 of that second register.

A *bit run* is a sequence of bits with all the same value. A *run of 0s* is a bit string that contains all 0s, and a *run of 1s* is a bit string containing all 1s. The *first set bit* in a bit string is the bit position of the first bit containing a 1 in a bit string—that is, the first 1 bit following a possible run of 0s. A similar definition exists for the *first clear bit*. The *last set bit* is the last bit position in a bit string that contains 1s; the remainder of the string forms an uninterrupted run of 0s. A similar definition exists for the *last clear bit*.

A *bit set* is a collection of bits, not necessarily contiguous, within a larger data structure. For example, bits 0 to 3, 7, 12, 24, and 31 in a double word form a set of bits. Typically, we will deal with bit sets that are part of a *container object* (the data structure that encapsulates the bit set) that is no more than about 32 or 64 bits in size, though this limit is completely artificial. Bit strings are special cases of bit sets.

A *bit offset* is the number of bits from a boundary position (usually a byte boundary) to the specified bit. As noted in Chapter 2, these bits are numbered starting from 0 at the boundary location.

A *mask* is a sequence of bits used to manipulate certain bits in another value. For example, the bit string 0b0000_1111_0000, when used with the `and` instruction, masks away (clears) all bits except bits 4 through 7. Likewise, if you use the same value with the `orr` instruction, it can set bits 4 through 7 in the destination operand. The term *mask* derives from the use of these bit strings with the `and` instruction, where the 1 and 0 bits behave like masking tape when you're painting something: they pass through certain bits unchanged while masking out (clearing) the other bits.

Armed with these definitions, you're ready to start manipulating some bits!

12.2 Instructions That Manipulate Bits

Let's begin by reviewing the instructions this book has covered so far that manipulate bits, along with introducing a few additional bit-manipulation instructions.

Bit manipulation generally consists of six activities: setting bits, clearing bits, inverting bits, testing and comparing bits, extracting bits from a bit string, and inserting bits into a bit string. The most basic bit-manipulation instructions are the `and`/`ands` `bic`, `orr`, `orn`, `eor`, `mvn` (`not`), `tst`, and shift and rotate instructions. This section discusses these instructions, concentrating on how to use them to manipulate bits in memory or registers.

12.2.1 Isolating, Clearing, and Testing Bits

The `and/ands` instruction provides the ability to clear bits in a bit sequence. This instruction is especially useful for isolating a bit string or a bit set that is merged with other, unrelated data (or, at least, data that is not part of the bit string or bit set). For example, if a bit string consumes bit positions 12 through 24 of the W0 register, you can isolate this bit string by using the following instruction to set all other bits in W0 to 0:

```
and w0, w0, 0b11111111111111000000000000
```

Figure 12-1 shows the result of this instruction.

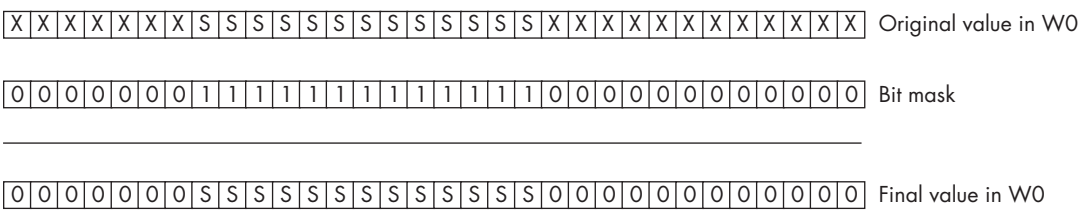


Figure 12-1: Isolating a bit string by using the and instruction

Once you've cleared the unneeded bits in a set of bits, you can often operate on the bit set in place. For example, to see whether the string of bits in positions 12 through 24 of W0 contains 0x2F3, you could use the following code:

```
mov w1, #0x2f3
lsl w1, w1,   #12 // Make it 0x2f3000.
and w0, w0,   #0b11111111111111000000000000
cmp w0, w1  // 0b00010111100110000000000000
```

You cannot use the immediate constant 0x2F3000 with the `cmp` instruction, so this code first loads that constant into W1 and compares W0 against W1. You also can't use `movz` to load 0x2F3 preshifted to the left 12 bits because `movz` allows only shifts of 0, 16, 32, or 48 bits; likewise, 0x2F3 is not a logical immediate pattern, so you can't use a `mov` instruction with 0x2F3000.

NOTE *The instruction* `mov w1, #logical_pattern` *is equivalent to* `orr w1, wzr, #logical_pattern`.

To make the constants you use in conjunction with this value easier to deal with, you can use the `lsr` instruction to align the bit string with bit 0 after you've masked it, like this:

```
and w0, w0, #0b11111111111111000000000000
lsr w0, w0, #12
cmp w0, #0x2F3
```

The plain `and` instruction does not affect any condition code flags. Use the `ands` variant if you would like to update the N and Z flags based on the result of the `AND` operation, remembering that this instruction always clears the carry and overflow flags.

If you want to capture the result of the `ands` operation in the N and Z flags but don't want to keep the logical result, you can use the `tst` instruction. This is equivalent to (and an alias of) `ands` supplying WZR or XZR as the destination register (which throws away the result).

Don't forget that you can also use the `and` instruction with the vector registers (both vector and scalar operations); just keep in mind that the vector `and` instruction doesn't affect the flags (there is no `ands` variant).

Because `tst` is an alias of `ands`, and there is no vector `ands` instruction, the Neon instruction set provides this `cmtst` instruction

```
cmtst Vd.t, Vd.t, Vd.t
```

where *t* is 8B, 16B, 4H, 8H, 2S, 4S, or 2D (8B, 4H, and 2S types operate on the LO 64 bits of V*n*, while the others operate on all 128 bits).

This instruction logically ANDs each lane in `Vd.t` with `Vd.t`. If the result is not 0, `cmtst` sets the destination lane to all 1s; if the result is 0, it sets the destination lane to all 0s. You can use the result as a bitmask for further vector (bit) operations.

12.2.2 Setting and Inserting Bits

The `orr` instruction is especially useful for inserting a bit set into another bit string (in general-purpose or vector registers), using the following steps:

1. Clear all the bits surrounding your bit set in the source operand.
2. Clear all the bits in the destination operand where you wish to insert the bit set.
3. OR the bit set and destination operand together.

For example, suppose you have a value in bits 0 to 11 of W0 that you wish to insert into bits 12 to 23 of W1 without affecting any of the other bits in W1. You would begin by stripping out bits 12 and higher from W0, then strip out bits 12 to 23 in W1. Next, you would shift the bits in W0 so the bit string occupied bits 12 to 23 of W0. Finally, you'd OR the value in W0 into W1, as shown in the following code:

```
and     w0, w0, #0xFFF   // Strip all but bits 0 to 11 from W0.
bic     w1, w1, 0xFFF000 // Clear bits 12 to 23 in W1.
lsl     w0, w0, 12       // Move bits 0 through 11 to 12 through 23 in W0.
orr     w1, w1, w0       // Merge the bits into W1.
```

Figure 12-2 shows the result of these four instructions.

Figure 12-2: Inserting bits 0 to 11 of W0 into bits 12 to 23 of W1

Step 1 in Figure 12-2 clears the `U` bits in W0. After clearing the `U` bits, step 2 masks out the destination bit field (`Y`). Step 3 shifts the `A` bits (bits 0 to 11 in W0) 12 positions to the left to align them with the destination bit

field. Step 4 ORs the value in W0 with the value in W1, leaving the final result in W1.

In Figure 12-2, the desired bits (AAAAAAAAAAAA) form a bit string. However, this algorithm still works fine even if you're manipulating a noncontiguous set of bits—you just have to create a bitmask with 1s in the appropriate places.

When working with bitmasks, it is incredibly poor programming style to use literal numeric constants, as in the past few examples. I've used "magic numbers" thus far because the examples have been simple and using literal constants is clearer in this context. However, you should always create symbolic constants in Gas. Combining these with constant expressions allows you to produce code that is much easier to read and maintain. The previous example code is more properly written as the following:

```
StartPosn = 12;
BitMask   = 0xFFF << StartPosn  // Mask occupies bits 12 to 23.
   .
   .
   .
  lsl w0, w0, #StartPosn  // Move into position.
  and w0, w0, #BitMask    // Strip all but bits 12 to 23.
  and w1, w1, #~BitMask   // Clear bits 12 to 23 in W1.
  orr w1, w1, w0          // Merge the bits into W1.
```

The use of the compile time NOT operator (~) for the bitmask inversion saves having to create another constant in the program that must be changed anytime you modify the BitMask constant. Maintaining two separate symbols whose values are dependent on each other is not good practice.

Of course, in addition to merging one bit set with another, the orr instruction is also useful for forcing bits to 1 in a bit string. By setting various bits in a source operand to 1, you can use the orr instruction to force the corresponding bits in the destination operand to 1.

You can use the orn (OR NOT) instruction to insert a 1 bit in a destination everywhere there is a 0 bit in the source. Other than that, the instruction behaves identically to orr.

If the and w1, w1, #~BitMask cannot be assembled because Gas will not accept the immediate constant, use bic w1, w1, #BitMask instead, as described in the next section.

12.2.3 Clearing Bits

Whereas the orr instruction is useful for setting bits in a register, the bic instruction clears them

```
bic  Rd, Rl, Rr  // Rd = Rl && ~Rr
bics Rd, Rl, Rr  // Also affects condition codes
```

where *Rd*, *Rl*, and *Rr* are all W*n* or X*n*. The bics instruction sets the N and Z condition codes based on the value left in *Rd*. Everywhere a 1 bit appears

in *Rr*, the `bic` instruction will clear the corresponding bit in *Rl* (storing the result into *Rd*).

The vector version of this instruction allows you to clear arbitrary bits in a 64- or 128-bit vector register

```
bic Vd.t, Vl.t, Vr.t
```

where *t* = 8B for 64 bits and *t* = 16B for 128 bits. This instruction also has an immediate version

```
bic Vd.t, #imm8
bic Vd.t, #imm8, lsl #shift
```

where *t* = 4H for 64 bits and *t* = 8H for 128 bits, or *t* = 2S for 64 bits and *t* = 4S for 128 bits. The optional shift value can be `0` or `8` for half-word lanes, or `0`, `8`, `16`, or `24` for word lanes.

12.2.4 Inverting Bits

The `eor` instruction allows you to invert selected bits in a bit set (in general-purpose or vector registers). If you want to invert all the bits in a destination operand, the `mvn` (not) instruction is more appropriate; however, to invert selected bits while not affecting others, `eor` is the way to go.

The `eor` instruction lets you manipulate known data in just about any way imaginable. For example, if you know that a field contains 0b1010, you can force that field to 0 by XORing it with 0b1010. Similarly, you can force it to 0b1111 by XORing it with 0b0101.

Although this might seem like a waste because you can easily force this 4-bit string to 0 or all 1s by using `and` or `orr`, the `eor` instruction has two advantages. First, you are not limited to forcing the field to all 0s or all 1s; you can set these bits to any of the 16 valid combinations via `eor`. Second, if you need to manipulate other bits in the destination operand at the same time, `and` or `orr` may not be able to do the job.

For example, suppose one field contains 0b1010, which you want to force to 0, while another field in the same operand contains 0b1000, and you wish to increment that field by 1 (that is, set it to 0b1001). You cannot accomplish both operations with a single `and` or `orr` instruction, but you can do so with a single `eor` instruction: just XOR the first field with 0b1010 and the second field with 0b0001. However, note that this trick works only if you know the current value of a bit set within the destination operand.

The `eon` (exclusive–OR NOT) works just like `eor`, except that it inverts bits in the destination wherever a 0 bit appears in the right source operand.

12.2.5 Shift and Rotate

The shift and rotate instructions are another group you can use to manipulate and test bits. The standard and Neon instruction sets provide a wide

range of shift and rotate instructions that allow you to rearrange bit data as needed:

`asr` Integer arithmetic shift right

`lsl` Integer logical shift left

`lsr` Integer logical shift right

`ror` Integer rotate right

`shl` Vector or scalar shift left or right

`ushl`, `sshl`, `ushr`, **and** `sshr` Vector or scalar shift right

`sli` Vector or scalar shift left and insert

`sri` Vector or scalar shift right and insert

In addition to these generic shifts and rotates, specialized variants saturate, round, narrow, and extend. Generally, those specialized instructions aren't as useful for bit manipulation.

The integer shift and rotate instructions (those that operate on general-purpose registers) are quite useful for moving bits into their final position when constructing bit strings from multiple sources. As you saw in section 2.12, "Bit Fields and Packed Data," on page 85, you can use the shift and rotate instructions (along with the logical instructions) to pack and unpack data.

However, the biggest problem with the shift and rotate instructions is that they don't provide an option to set the condition codes. For example, on many other CPUs (including the 32-bit ARM instruction set), the last bit shifted out of a register during a shift or rotate instruction winds up in the carry flag. The instructions typically set the Z and N flags, based on the final result. This is convenient for many operations, particularly those that use a loop to process each bit in the register; you could, for example, shift the bits out of a register by using a shift-right instruction, capture the output bit in the carry (using `bcc` or `bcs` to test the bit), and also be able to test the zero flag to see whether any more (set) bits are left in the register. This is not possible (with a single `shift` or `rotate` instruction) on the ARM.

This rule has one exception. Although the shifts do not affect the carry flag, the `adds` instruction does. Using an instruction such as

```
adds w0, w0, w0
```

is equivalent to a shift left (by 1 bit) on W0, setting all the condition code flags (including capturing the bit shifted out of bit 31 in the carry flag). The `adcs` instruction behaves like a "rotate through carry by 1 bit" instruction (see section 8.1.11.1, "Shift Left," on page 467). Because Chapters 2 and 8 covered the use of the shift instructions to insert and extract bit data (packed fields), this chapter doesn't consider that further.

The vector shift instructions warrant further discussion because they include interesting variants (`sli` and `sri`) and don't provide rotate instructions. Simulating a vector rotate left takes three instructions:

```
// Simulate rol v1.4s, v1.4s, #4:

        ushr    v2.4s, v1.4s, #28      // HO 4 bits to 0:3
        shl     v1.4s, v1.4s, #4       // Bits 0:27 to 4:31
        orr     v1.16b, v1.16b, v2.16b // Merge in LO bits.
```

The biggest problem with this code is that it takes an extra register to hold a temporary result.

Here are the instructions to implement a vector rotate right:

```
// Simulate ror v1.4s, v1.4s, #4:

        shl    v2.4s, v1.4s, #28      // Bits 0:3 to 28:31
        ushr   v1.4s, v1.4s, #4       // Bits 4:31 to 0:27
        orr    v1.16b, v1.16b, v2.16b // Merge bits.
```

See section 11.6.4, "Shift and Insert," on page 652 for more information about the `sli` and `sri` instructions.

Because the vector shift instructions operate only on lanes, you cannot directly shift all 128 bits of a vector register with a single instruction. With five instructions, however, you can pull it off:

```
// Simulate shl for 128 bits:

        ushr    v2.2d, v1.2d, #60      // Save bits 60:63.
        mov     v2.b[8], v2.b[0]       // Move into HO dword.
        mov     v2.b[0],  wzr          // No mask at original
        Shl     v1.4s, v1.4s, #4       // Shift bits left.
        orr     v1.16b, v1.16b, v2.16b // Merge bits 60:63.
```

I'll leave it up to you to do the shift-right operation, which is just a straightforward modification of the `shl` code.

12.2.6 Conditional Instructions

The `csel`, `csinc/cinc`, `csinv/cinv`, `cset`, and `csetm` instructions are also useful for manipulating bits. In particular, `cset` and `csetm` can help you initialize a register with 0, 1, or all 1 bits based on the condition codes. All these instructions are useful for dealing with operations that have set the condition codes (see section 12.3, "Flag Modification by Arithmetic and Logical Instructions," on page 715). These conditional instructions have no vector equivalents.

12.2.7 Counting Bits

The `cls` and `clz` instructions allow you to count leading 1 and 0 bits (*leading* means from the HO bit position down to the LO bit position). The `cnt` instruction (population count) counts all the set bits in the byte lanes in a (vector) register. The syntax for these instructions is

```
cls Rd, Rs
clz Rd, Rs
```

```
cls Vd.t1, Vs.t1
clz Vd.t1, Vs.t1
cnt Vd.t2, Vs.t2
```

where *Rd* and *Rs* are W*n* or X*n; t1* is 8B, 16B, 4H, 8H, 2S, or 4S; and *t2* is 8B or 16B. For the vector instructions, if the operand is 8B, 4H, or 2S, the instruction operates only on the LO 64 bits of the vector register.

The `cls` instruction (general-purpose register form) counts the number of leading sign bits in *Rs* and stores this count into *Rd*. This instruction counts the number of bits (below the HO bit) that match the HO bit. Note that the sign (HO) bit is not included in the count.

The `clz` instruction works the same way but counts the number of leading 0 bits including the sign bit. To count the actual number of leading 1 bits (including the HO sign bit), invert the source register value and use the `clz` instruction to count 0 bits.

The vector versions of the `cls` and `clz` instructions count the number of leading 1s or 0s in each lane of the source register, in the same manner as the scalar versions, and stores this count into the corresponding lane of the destination register.

The `cnt` instruction counts the number of set bits in each byte (lane) of the source register and stores the bit count into the corresponding lane of the destination register. To find the total population (bit) count for the vector register (64 or 128 bits), use the `addv` instruction to sum up all the bytes within the destination register. To compute the population count for hwords, words, or dwords, use an `addp` instruction to add the pairs of bytes (to produce hword counts). You can use a second `addp` instruction (again, on pairs of bytes, producing an array of byte results) to count the pairs of hwords, producing word counts, and so on.

12.2.8 Bit Reversal

The `rbit` instruction reverses the bits in its source operand and stores the reversed result into the destination operand. The syntax is as follows:

```
rbit Wd, Ws
rbit Xd, Xs
rbit Vd.t, Vs.t // t = 8B or 16B
```

With 32-bit register operands, this instruction swaps the bits in positions 0 and 31, 1 and 30, 2 and 29, . . . , and 15 and 16. With 64-bit operands, `rbit` swaps the bits in positions 0 and 63, 1 and 62, 2 and 61, . . . , and 31 and 32. With vector operands, it reverses the bits in each byte lane of the source vector register, storing the results into the corresponding byte lane in the destination vector register.

The vector variant of this instruction reverses only the bits in byte lanes. To reverse the bits in a 16-, 32-, or 64-bit object, also execute an 8-bit lane `rev16`, `rev32`, or `rev64` instruction (before or) after the `rbit`. For example, the following two instructions reverse all the bits in a 64-bit vector register:

```
rbit  v1.16b, v1.16b  // Do a 2D bit reversal; first bits,
rev64 v1.16b, v1.16b  // then bytes.
```

You would use similar code (with `rev32`) to reverse the bits in two doublewords in a vector register.

12.2.9 Bit Insertion and Selection

The `bif` (bit insert if false), `bit` (bit insert if true), and `bsl` (bit select) instructions allow you to manipulate individual bits in a vector register. These instructions have the following syntax:

```
bif Vd.t, Vs.t, Vm.t    // t = 8B (64 bits) or 16B (128 bits)
bit Vd.t, Vs.t, Vm.t    // Vd = dest, Vs = source, Vm = mask
bsl Vd.t, Vs1.t, Vs0.t
```

The `bif` instruction first considers the mask register; everywhere a 0 bit appears in `Vm`, the instruction copies the corresponding bit in `Vs` (source) to the same bit position in `Vd` (destination). Everywhere a 1 appears in `Vm`, the `bif` instruction leaves the corresponding bit in `Vd` unchanged.

The `bit` instruction does the same thing as `bif`, except it copies bits on the opposite condition (where there is a 1 in `Vm`).

The `bsl` instruction uses the (original) bits in `Vd` to select the corresponding bit in `Vs1` or `Vs0`. Everywhere a 1 appears in `Vd`, `bsl` will copy the corresponding bit from `Vs1` to `Vd`; everywhere a 0 appears in `Vd`, `bsl` will copy the corresponding bit from `Vs0` to `Vd`.

12.2.10 Bit Extraction with ubfx

The `ubfx` instruction allows you to extract any number of bits from a position in a source register, moving those bits to bit 0 in the destination register. The syntax is

```
ubfx Rd, Rs, #lsb, #len
```

where *Rd* and *Rs* are both either W*n* or X*n*, *lsb* is the starting bit position for the extraction, and *len* is the size of the bit string to extract. The sum of *lsb* and *len* must not exceed the register size.

The `ubfx` instruction extracts *len* bits from *Rs*, starting at bit position *lsb*. It stores this bit string into *Rd* (at bit 0) and zeros out the HO bits of *Rd*. For example

```
ubfx x0, x1, #8, #16
```

copies bits 8 through 23 from X1 to bits 0 through 15 in X0 (and zeros out bits 16 through 63 in X0).

12.2.11 Bit Movement with ubfiz

The `ubfiz` (unsigned bit field insert in zero) copies bits from the LO bits of a source register to any other position in the destination register, as the converse of the `ubfx` instruction. The syntax for this instruction is

```
ubfiz Rd, Rs, #posn, #len
```

where *Rd* and *Rs* are both either W*n* or X*n*, *posn* is the destination location where bit 0 from *Rs* will be moved, and *len* is the size of the bit string. For example

```
ubfiz w1, w0, #12, #8
```

copies bits 0 through 7 from W0 to bits 12 through 19 in W1.

12.2.12 Bit Movement with ubfm

The `ubfm` instruction (unsigned bit field move) copies the LO bits from a source register to an arbitrary position in the destination register (and puts 0s everywhere else in the destination register). The syntax is

```
ubfm Rd, Rs, #immr, #imms
```

where *Rd* and *Rs* are both either W*n* or X*n*, *immr* and *imms* are values in the range 0 to 31 for 32-bit operations or 0 to 63 for 64-bit operations. This instruction does one of two operations based on the values of *immr* and *imms*:

- If *immr* ≤ *imms*, take bits *immr* through *imms* and rotate right by *immr*.
- If *immr* > *imms*, take *imms* + 1 LO bits and rotate right by *immr*.

The `ubfm` instruction is the basis for many instructions (aliases) in the ARM instruction set: `lsl` *Rd, Rs, #shift* is equivalent to

```
ubfm Rd, Rs, #(Rsize - shift) % Rsize, #Rsize - 1 - shift
```

where *Rsize* is the register size (32 or 64).

Meanwhile, `lsr` *Rd, Rs, #shift* is equivalent to

```
ubfm Rd, Rs, #shift, #Rsize - 1 // Rsize is register size.
```

and `ubfiz` *Rd, Rs, #lsb, #width* is equivalent to

```
ubfm Rd, Rs, #(Rsize-lsb) % Rsize, #width - 1
```

where *Rsize* is the register size (32 or 64). Finally, `ubfx` *Rd, Rs, #lsb, #width* is equivalent to

```
ubfm Rd, Rs, #lsb, #lsb + width - 1
```

where *lsb* is the LO bit of the bit string to move and *width* is the number of bits to move.

12.2.13 Bit Extraction with extr

The extr instruction allows you to extract a bit string across a pair of registers, using the syntax

```
extr Rd, Rl, Rr, #posn
```

where *Rd*, *Rl*, and *Rr* are all W*n* or X*n*, and *posn* is a constant in the range 0 to the register size – 1 (31 or 63).

This instruction begins by concatenating the *Rl* and *Rr* registers to form a 64- or 128-element bit string. It then extracts 32 or 64 bits (depending on the register sizes) from this string, starting at bit position *posn*.

NOTE *The* ror *(immediate) instruction is an alias of* extr *obtained by setting* Rl *and* Rr *to the same register.*

12.2.14 Bit Testing with tbz and tbnz

The tbz (test bit zero) and tbnz (test bit not zero) instructions allow you to branch to a location based on whether a particular bit is set in a register, using the following syntax:

```
tbz  Rs, #imm, target  // Rs = Wn or Xn, imm = 0 to 31 (Wn) or
tbnz Rs, #imm, target  // 0 to 63 (Xn). target is a stmt label.
```

These instructions test the bit specified by *imm* in Rs to see if it is a 0 or a 1. The tbz instruction transfers control to the specified target label if it is 0 (falling through if it is 1), while tbnz transfers control to the specified target label if it is 1 (falling through if it is 0).

You can use these instructions to turn any register into a 32- or 64-bit "pseudo condition code register," allowing you to branch based on whether a particular bit in that register is set or clear. Though no instructions will automatically set or clear these "condition codes," you can use any of the bit-manipulation instructions in this chapter to manipulate those pseudo condition codes.

Don't forget that you can also use the cbz (compare and branch if zero) and cbnz (compare and branch if not zero) instructions to compare a register against 0 and transfer control if it is (cbz), or is not (cbnz), equal to 0. This is useful after instructions such as addv, orr, or other instructions that don't set the Z flag, to see if they've produced a zero (or nonzero) result.

12.3 Flag Modification by Arithmetic and Logical Instructions

In the previous sections, the instructions manipulated bits in general-purpose and vector registers. Although the PSR is not a general-purpose

register, remember that the ands, bics, and tst instructions set the N and Z flags based on the computed result. If the HO (sign) bit of the result is 1, these instructions set the N flag; if the result is 0, these instructions set the Z flag. Otherwise, they clear the flags.

You can also use the adds, adcs, subs, sbcs, negs, ngcs, cmp, ccmp, and ccmn instructions to set the flags. In particular, keep in mind that

```
adds Rd, Rn, Rn // R = X or W
```

shifts the value in *Rn* to the left one bit and moves the (original) HO bit into the carry flag. This instruction sets the overflow flag if the two original HO bits contain 0b01 or 0b10. It sets the negative flag if the resulting HO bit is 1. Finally, of course, it sets the zero flag if the result of the addition is 0.

Also note that the instruction

```
adcs Rn, Rn, Rn // R = X or W
```

is equivalent to a "rotate left *Rn* 1 bit through the carry flag" instruction. You can use this to set the carry flag based on the HO bit of *Rn* and capture the previous carry flag value into the LO bit of *Rn*.

Keep in mind that only the arithmetic and logical instructions that operate on general-purpose registers (and have the s suffix) affect the flags. In particular, the vector instructions do not affect the flags.

While it's not an arithmetic or logical instruction, the mrs (move register to status) instruction, with the destination field NZCV, sets all the flags to the values found in bits 28–31 of the general-purpose register. This provides a quick way to create a multiway branch from 4 bits. Consider the following code (taken from section 11.10.4, "Vector Comparison Results," on page 691):

```
        lea     r3, mask            // 0x0008000400020001
        ldr     q3, [r3]
        cmeq    v0.4h, v1.4h, v2.4h
        and     v0.8b, v0.8b, v3.8b // Keep LO bit of each lane.
        addv    h0, v0.4h           // Merge the bits into H0.
        umov    w0, v0.h[0]
        lsl     w0, w0, #28
        mrs     x0, nzcv
```

Now you can test N, Z, C, and V flags to see if lanes 3, 2, 1, or 0 (respectively) in V1 were equal to the corresponding lanes in V2.

This section has provided a generic introduction to setting the condition code flags to capture a bit value. Because instructions with an s suffix affect the flags differently, it is important to discuss how instructions affect the individual condition code flags; the following sections handle that task.

12.3.1 The Zero Flag

The zero flag (Z) setting is one of the most important results produced by the ands instruction. Indeed, programs reference this flag so often after

the ands instruction that ARM added a separate tst instruction whose main purpose is to logically AND two results and set the flags without otherwise affecting either instruction operand.

NOTE *Technically, tst is not a new instruction, but rather an alias for ands when the destination register is WZR or XZR.*

The zero flag can be used to check three things after the execution of an ands or tst instruction: whether a particular bit in an operand is set, whether at least one of several bits in a bit set is 1, and whether an operand is 0. The first use case is actually a special instance of the second, in which the bit set contains only a single bit. The following paragraphs explore each of these uses.

To test whether a particular bit is set in a given operand, ands or tst an operand with a constant value containing a single set bit you wish to test. This clears all the other bits in the operand, leaving a 0 in the bit position under test if the operand contained a 0 in that bit position and a 1 if it contained a 1. Because all other bits in the result are 0, the entire result will be 0 if that particular bit is 0; the entire result will be nonzero if that bit position contains a 1. The ARM CPU reflects this status in the zero flag (Z = 1 indicates a 0 bit; Z = 0 indicates a 1 bit). The following instruction sequence demonstrates how to test if bit 4 is set in W0:

```
  tst w0, 0b10000  // Check bit #4 to see if it is 0/1.
  bne bitIsSet

  Do this if the bit is clear.
   .
   .
   .
bitIsSet:   // Branch here if the bit is set.
```

You can also use the ands and tst instructions to see whether any one of several bits is set. Simply supply a constant that has a 1 in all the positions you want to test (and 0s everywhere else). ANDing an operand with such a constant will produce a nonzero value if one or more of the bits in the operand under test contain a 1. The following example tests whether the value in W0 contains a 1 in bit positions 1 and 2:

```
    tst w0, 0b0110
    beq noBitsSet

    Do whatever needs to be done if one of the bits is set.

noBitsSet:
```

You cannot use a single ands or tst instruction to see whether all the corresponding bits in the bit set are equal to 1. To accomplish this, you must first mask out the bits that are not in the set and then compare the

result against the mask itself. If the result is equal to the mask, all the bits in the bit set contain 1s. You must use the ands instruction for this operation because the tst instruction does not modify the result. The following example checks whether all the bits in a bit set (bitMask) are equal to 1:

```
    ldr  w1, =bitMask     // Assume not valid immediate const.
    ands w0, w0, w1
    cmp  w0, w1
    bne  allBitsArentSet

// All the bit positions in W0 corresponding to the set
// bits in bitMask are equal to 1 if we get here.

    Do whatever needs to be done if the bits match.

allBitsArentSet:
```

Of course, once you stick the cmp instruction in there, you don't really have to check whether all the bits in the bit set contain 1s. You can check for any combination of values by specifying the appropriate value as the operand to cmp.

The tst and ands instructions will set the zero flag in the preceding code sequences only if all the bits in W0 (or other destination operand) have 0s in the positions where 1s appear in the constant operand. This suggests another way to check for all 1s in the bit set: invert the value in W0 prior to using the ands or tst instruction. In this case, if the zero flag is set, you know that the (original) bit set contained all 1s. For example, the following code checks whether any bits specified by 1s in a bitmask are nonzero:

```
    ldr  w1, =bitMask     // Assume not valid immediate const.
    mvn  w0, w0
    tst  w0, w1
    bne  NotAllOnes

// At this point, W0 contained all 1s in the bit positions
// occupied by 1s in the bitMask constant.

    Do whatever needs to be done at this point.

NotAllOnes:
```

The previous paragraphs all suggest that the bitMask (the source operand) is a constant, but you can use a variable or other register too. Simply load that variable or register with the appropriate bitmask before you execute the tst, ands, or cmp instructions in the preceding examples.

12.3.2 The Negative Flag

The tst and ands instructions will also set the negative flag (N, also known as the sign flag) if the HO bit of the result is set. This allows you to test two individual bits in a register, assuming one of those bits is the HO bit. When

using these instructions, the mask value must contain a 1 in the HO bit as well as in the bit position of the other bit you want to test. After executing the `tst` or `ands` instruction, you must check the N flag before testing the Z flag (as the Z flag will be clear if the HO bit was set).

If the HO bit is set and you also want to see whether the other bit is set, you must test that other bit again (or use the `cmp` instruction, as in the previous section).

12.3.3 The Carry and Overflow Flags

The logical instructions (`ands` and `tst`) that affect the flags always clear the carry and overflow flags. However, the arithmetic instructions (`adds`, `adcs`, `subs`, `sbcs`, `negs`, `ngcs`, `cmp`, `ccmp`, and `ccmn`) modify these flags. In particular, when the two source operands are the same, `adds` and `adcs` shift the HO bit of the (original) source into the carry flag.

Negating the most negative value (for example, the word value `0x80000000`) will set the overflow flag:

```
orr     w1, wzr, #0x80000000  // mov x1, #0x80000000
negs    w1, w1                // Sets V (and N) flags
```

Refer to the ARM AARCH64 documentation to determine how the various instructions affect the carry and overflow flags. Note that many instructions will not affect these two flags (especially the overflow flag) even though they affect the N and Z flags.

12.4 Packing and Unpacking Bit Strings

Inserting a bit string into an operand and extracting a bit string from an operand are common operations. Chapter 2 provided simple examples of packing and unpacking such data; this section formally describes how to do this, now that you've learned more instructions and have more tools to work with.

12.4.1 Inserting One Bit String into Another

For the purposes of this chapter, I will assume that we're dealing with bit strings that fit within a byte, half-word, word, or double-word operand. Large bit strings that cross object boundaries require additional processing; I discuss bit strings that cross double-word boundaries later in this section.

When packing and unpacking a bit string, you must consider its starting bit position and length. The *starting bit position* is the bit number of the LO bit of the string in the operand. The *length* is the number of bits in the string.

To insert (pack) data into a destination operand, start with a bit string of the appropriate length that is right-justified (starts in bit position 0) and zero-extended to 8, 16, 32, or 64 bits. Next, insert this data at the appropriate starting position in another operand that is 8, 16, 32, or 64 bits wide.

The destination bit positions are not guaranteed to contain any particular value.

The first two steps (which can occur in any order) are to clear out the corresponding bits in the destination operand and shift a copy of the bit string so that the LO bit begins at the appropriate bit position. The third step is to OR the shifted result with the destination operand. This inserts the bit string into the destination operand. Figure 12-3 diagrams this process.

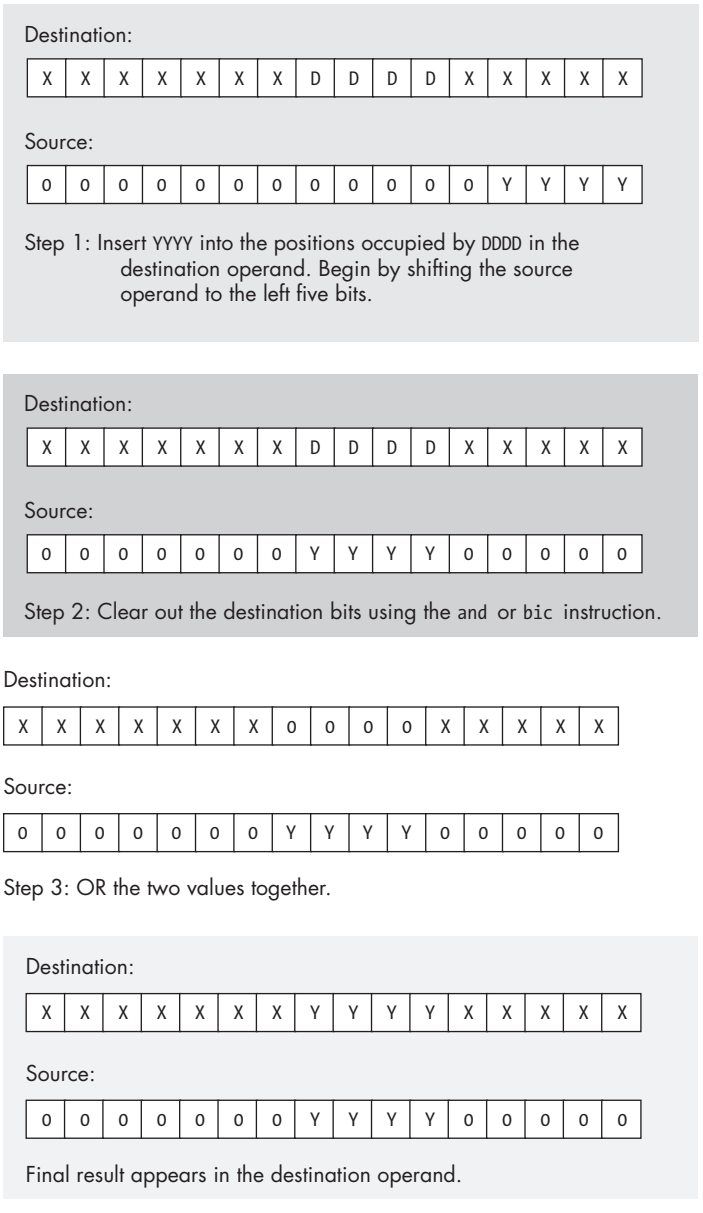


Figure 12-3: Inserting a bit string into a destination operand

The following three instructions insert a bit string of known length into a destination operand, as shown in Figure 12-3. These instructions assume that the source bit string is in W1 (with 0s in positions outside the bit string) and the destination operand is in W0:

```
lsl  w1, w1, #5
bic  w0, w0, #0b111100000
orr  w0, w0, w1
```

For the special case when the destination bit position and bit string length are constants (known at assembly time), the ARM CPU provides an instruction to handle bit insertion for you: `bfi` (bit field insert). It has the following syntax

```
bfi Rd, Rs, #posn, #len
```

where *Rd* and *Rs* are both either W*n* or X*n*. The sum of *posn* and *len* must not exceed the register size (32 for W*n* and 64 for X*n*).

The `bfi` instruction takes the LO *len* bits of *Rs* and inserts them into the destination register (*Rd*) starting at bit position *posn*. Consider the instruction

```
bfi w0, w1, #12, #16
```

assuming W0 contained 0x33333333 (the destination value) and W1 contained 0x1200 (the insertion value). This would leave 0x31200333 in W0.

If you don't know the length and the starting position while writing the program (that is, you have to calculate them at runtime), you must use several instructions to do a bit-string insertion. Suppose you have two values—a starting bit position for the field you're inserting and a nonzero length value—and that the source operand is in W1 and the destination operand is in W0. The `mergeBits` procedure in Listing 12-1 demonstrates how to insert a bit string from W1 into W0.

```
// Listing12-1.S
//
// Demonstrate inserting bit strings into a register.
//
// Note that this program must be assembled and linked
// with the "LARGEADDRESSAWARE:NO" option.

#include    "aoaa.inc"

            .text
            .pool

ttlStr:     wastr   "Listing 12-1"
```

```
// Sample input data for the main program:

Value2Merge:
            .dword  0x12, 0x1e, 0x5555
            .dword  0x1200, 0x120

MergeInto:
            .dword  0xffffffff, 0, 0x12345678
            .dword  0x33333333, 0xf0f0f0f

LenInBits:  .dword  5, 9, 16, 16, 12
szLenInBits =       (.-LenInBits)/8

StartPosn:  .dword  7, 4, 4, 12, 18

// Format strings used to print results:

fmtstr1:    wastr   "merge( %x, "
fmtstr2:    wastr   "%x, "
fmtstr3:    wastr   "%d ) = "
fmtstr4:    wastr   "%x\n"
fmtstr:     wastr   "Here I am!\n"

// getTitle
//
// Returns a pointer to the program's name
// in X0:

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

// MergeBits( Val2Merge, MergeWith, Start, Length )
//
// Length (LenInBits[i]) value is passed in X3.
// Start (StartPosn[i]) is passed in X2.
// Val2Merge (Value2Merge[i]) and MergeWith (MergeInto[i])
// are passed in X1 and X0.
//
// mergeBits result is returned in X0.

            proc    mergeBits

            locals  mb
            qword   mb.x1x2
            qword   mb.x3x4
            byte    mb.stk, 64
            endl    mb

            enter   mb.size

            stp     x1, x2, [fp, #mb.x1x2]
            stp     x3, x4, [fp, #mb.x3x4]
```

```
                // Generate mask bits
                // 1 in bits 0 to n - 1:

              ❶ mov     x4, #1
                lsl     x4, x4, x3  // Compute 2**n.
                sub     x4, x4, #1  // 2**n - 1

                // Position mask bits to target location:

              ❷ lsl     x4, x4, x2

                // Mask out target bits:

              ❸ bic     x0, x0, x4

                // Merge the bits:

              ❹ lsl     x1, x1, x2
                orr     x0, x0, x1

                // Restore registers and return:

                ldp     x3, x4, [fp, #mb.x3x4]
                ldp     x1, x2, [fp, #mb.x1x2]
                leave
                endp    mergeBits

// Here is the asmMain function:

              ❺ proc    asmMain, public

                locals  am
                qword   am.x20x21
                qword   am.x22x23
                dword   am.x24
                byte    am.stk, 256
                endl    am

                enter   am.size
                stp     x20, x21, [fp, #am.x20x21]
                stp     x22, x23, [fp, #am.x22x23]
                str     x24, [fp, #am.x24]

                // The following loop calls mergeBits as
                // follows
                //
                // mergeBits
                // (
                //      Value2Merg[i],
                //      MergeInto[i],
                //      StartPosn[i],
                //      LenInBits[i] );
                //
                // where "i" runs from 4 down to 0.
```

```
                //
                // Index of the last element in the arrays:

                mov     x20, #szLenInBits - 1

testLoop:

                // Fetch the Value2Merge element and write
                // its value to the display while it is
                // handy:

                lea     x1, Value2Merge
                ldr     x1, [x1, x20, lsl #3]
                mstr    x1, [sp]
                lea     x0, fmtstr1
                mov     x22, x1             // Save for later.
                bl      printf

                // Fetch the MergeInto element and write
                // its value to the display:

                lea     x1, MergeInto
                ldr     x1, [x1, x20, lsl #3]
                mstr    x1, [sp]
                mov     x21, x1             // Save for later.
                lea     x0, fmtstr2
                bl      printf

                // Fetch the StartPosn element and write
                // its value to the display:

                lea     x1, StartPosn
                ldr     x1, [x1, x20, lsl #3]
                mstr    x1, [sp]
                mov     x23, x1             // Save for later.
                lea     x0, fmtstr2
                bl      printf

                // Fetch the LenInBits element and write
                // its value to the display:

                lea     x1, LenInBits
                ldr     x1, [x1, x20, lsl #3]
                mstr    x1, [sp]
                mov     x24, x1             // Save for later.
                lea     x0, fmtstr3
                bl      printf

                // Call MergeBits:
                //      (
                //          Value2Merge,
                //          MergeInto,
                //          StartPosn,
                //          LenInBits
                //      );
```

```
            mov     x0, x21
            mov     x1, x22
            mov     x2, x23
            mov     x3, x24
            bl      mergeBits

            // Display the function result (returned in
            // X0. For this program, the results are
            // always 32 bits, so it prints only the L0
            // 32 bits of X0):

            mov     x1, x0
            mstr    x1, [sp]
            lea     x0, fmtstr4
            bl      printf

            // Repeat for each element of the array:

            subs    x20, x20, #1
            bpl     testLoop

allDone:
            ldp     x20, x21, [fp, #am.x20x21]
            ldp     x22, x23, [fp, #am.x22x23]
            ldr     x24, [fp, #am.x24]
            leave
            endp    asmMain
```

The `mergeBits` function is where the merging occurs. This code begins by generating a mask containing all 1 bits from location 0 to $n - 1$, where n is the length of the bit string to insert ❶. The code uses a simple mathematical trick to generate these bits: if you compute 2^n and then subtract 1 from this value, the resulting value contains 1 bits in positions 0 to $n - 1$. After generating this mask, the code positions the mask bits to the position where `mergeBits` will insert the bit string ❷. It then masks out (sets to 0) those bit positions in the destination location ❸.

To complete the merge, `mergeBits` moves the bits to merge to the proper position and ORs those bits into the destination location (which contains 0s at that point) ❹. The `mergeBits` function assumes that the source bits (the bits to merge) form a bit string that is exactly n bits long (n being the value passed in X3) and is located in bit positions 0 to $n - 1$. Note that if you need to handle bit-insertion values that might have 1 bits in positions n or greater, you should logically AND the value to merge (passed in X1) with the bitmask after shifting it ❹. The version of `mergeBits` in Listing 12-1 assumes that the `val2Merge` argument (X1) doesn't contain any extra 1 bits.

The `asmMain` function ❺ is a loop that steps through the `ValueToMerge`, `MergeInto`, `LenInBits`, and `StartPosn` arrays. This loop fetches these four values, prints them, and then calls the `mergeBits` function to merge the `ValueToMerge` entry into `MergeInto`. The `LenInBits` element contains the size (in bits) to merge, and the value from the `StartPosn` array is the bit position where the merger should take place.

Here's the `build` command and sample output for Listing 12-1:

```
% ./build Listing12-1
% ./Listing12-1
CallingListing 12-1:
merge( 120, f0f0f0f, 12, 12 ) = 4830f0f
merge( 1200, 33333333, c, 16 ) = 31200333
merge( 5555, 12345678, 4, 16 ) = 12355558
merge( 1e, 0, 4, 9 ) = 1e0
merge( 12, ffffffff, 7, 5 ) = fffff97f
Listing12-1 terminated
```

The `mergeBits` function is very general, allowing you to specify the bit string length and the destination position as variable parameter values. If the length and destination position values are constants within your code (a common special case), you can use a more efficient way to insert bits from one register into another: the `bfm` (bit field move) instruction. This instruction has the syntax

```
bfm Rd, Rs, #rotate, #bitposn
```

where *`Rd`* and *`Rs`* are both either W*n* or X*n*, *`rotate`* is the number of rotate-right positions, and *`bitposn`* is the leftmost bit in the source (starting at bit 0) to move.

This instruction rotates (a copy of) the LO *`bitposn`* bits in *`Rs`* the specified number of bit positions, then replaces the corresponding bits in *`Rd`* with these rotated bits.

NOTE *The `bfi` instruction is an alias of `bfm` with a slight modification to the meaning of the two immediate operands (see the ARM documentation for more details).*

The examples in this section assume that the bit string appears completely within a double-word (or smaller) object. This will always be the case if the bit string is less than or equal to 64 bits in length. However, if the length of the bit string plus its starting position (modulo 8) within an object is greater than 64, the bit string will cross a double-word boundary within the object.

Inserting such bit strings requires up to three operations: one to extract the start of the bit string (up to the first double-word boundary), one to copy whole double words (assuming the bit string is so long it consumes several double words), and one to copy leftover bits in the last double word at the end of the bit string. I'll leave the implementation of this operation as an exercise for you.

12.4.2 Extracting a Bit String

The previous section described how to insert one bit string into another. This section covers the converse operation: extracting a bit string from a larger string.

The bfxil (bit field extract and insert at low end) instruction extracts any number of bits (at any position) from a source register and copies those bits to the LO bit positions of a destination register. Its syntax is

```
bfxil Rd, Rs, #posn, #len
```

where *Rd* and *Rs* are either W*n* or X*n* . The sum of *posn* and *len* must not exceed the register size (32 for W*n* and 64 for X*n*), and *posn* must be less than the register size.

This instruction extracts *len* bits, starting at bit number *posn*, from Rs and inserts them into the LO *len* bits of *Rd*. It does not affect the other bits (at bit positions *len* and higher) in *Rd*. Generally, you'll want to set *Rd* to 0 before using this instruction, as shown in the following example:

```
mov   w0, wzr         // Extract bits 5 through 12 from W1
bfxil w0, w1, #5, #8  // and store them in W0.
```

Like the bfi instruction, bfxil supports only immediate constants for the *posn* and *len* operands. If you need to specify variables for either (or both) of these arguments, you must write an extractBits function (similar to mergeBits in the previous section). The following instructions do the actual bit extraction in extractBits:

```
// Generate mask bits
// 1 in bits 0 to n - 1:

mov     x4, #1
lsl     x4, x4, x3  // Compute 2**n.
sub     x4, x4, #1  // 2**n - 1

// Position mask bits to target location:

lsl     x4, x4, x2

// Extract the target bits:

and     x1, x1, x4

// Right-justify the bits to bit 0:

lsr     x0, x1, x2
```

This leaves the extracted bits in the LO bit positions of X0.

12.4.3 Clearing a Bit Field

The Gas assembler provides an alias of the bfm instruction that you can use to clear bits in a register: bfc (bit field clear). Its syntax is

```
bfc Rd, #posn, #len
```

where *Rd* is W*n* or X*n* and *posn* and *len* have the same meanings and restrictions as the `bfi` instruction. If you supply a *len* field of 1, you can clear individual bits (specified by a bit number) with `bfc`.

The `bfc` instruction zeros out *len* bits starting at bit position *posn* in *Rd*. It is equivalent to the following instruction

```
bfi Rd, Rzr, #posn, #len  // Rzr = WZR or XZR
```

where *Rd* is W*d* or X*d*, as appropriate.

The `bfc` instruction is available only on ARMv8.2-a and later CPUs, not on Raspberry Pi (3 and 4) and other lower-end systems. (Note that the Raspberry Pi 5 does support this instruction.)

12.4.4 Using bfm

The `bfxil` and `bfi` (as well as `bfc`) instructions are actually aliases for the instruction `bfm`:

```
bfm Rd, Rs, #immr, #imms
```

Like the `ubfm` instruction, they do two operations based on the values of *immr* and *imms*:

- If *immr* ≤ *imms*, take bits *immr* through *imms* from *Rs* and rotate right by *immr*, merging with the existing bits in *Rd*.
- If *immr* > *imms*, take *imms* + 1 LO bits from *Rs* and rotate right by *immr*, merging with the existing bits in *Rd*.

  For example

```
ldr     w0, =0xffffffff
mov     w1, #0x2
bfm     w0, w1, #4, #2
```

produces 0xAFFFFFFF in W0.

The `bfi` instruction is equivalent to the following:

```
bfm Rd, Rs, #(-posn % 64), #(len-1)
```

The `bfxil` instruction is equivalent to this:

```
bfm Rd, Rs, #posn, #(len+posn-1)
```

Generally, you would use these aliases rather than the `bfm` mnemonic.

12.5 Common Bit Operations

You'll encounter many bit-manipulation design patterns in assembly language programs. This section covers some of the more common algorithms and patterns.

12.5.1 Coalescing Bit Sets and Distributing Bit Strings

Inserting and extracting bit sets is only a little different from inserting and extracting bit strings if the "shape" of the bit set you're inserting (or resulting bit set you're extracting) is the same as the shape of the bit set in the main object. The *shape* of a bit set is the distribution of the bits in the set, ignoring the starting bit position of the set. A bit set that includes bits 0, 4, 5, 6, and 7 has the same shape as a bit set that includes bits 12, 16, 17, 18, and 19 because the distribution of the bits is the same.

The code to insert or extract this bit set is nearly identical to that of the previous sections; the only difference is the mask value you use. For example, to insert this bit set starting at bit 0 in W0 into the corresponding bit set starting at position 12 in W1, you could use the following code:

```
ldr w2, =0b11110001000000000000 // Bit set mask in posn.
lsl w0, w0, #12                 // Move src bits into posn.
and w0, w0, w2                  // Mask out source bits.
bic w1, w1, w2                  // Clear out destination bits.
orr w1, w1, w0                  // Merge bit set into W1.
```

However, suppose you have five bits in bit positions 0 through 4 in W0 and want to merge them into bits 12, 16, 17, 18, and 19 in W1. Somehow you have to *distribute* the bits in W0 prior to logically ORing the values into W1; that is, you have to move the bits from positions 0 to 4 into positions 12, 16, 17, 18, and 19.

The converse operation, *coalescing bits*, extracts the bits from various bit positions and packs them (coalesces them) into the LO bit positions of a destination location. The following code demonstrates how to distribute the bits in a bit string according to the values in a bitmask:

```
// W0- Contains the source value to insert the bits into
// W1- Contains the bits to insert, justified against bit 0
// W2- Counter (size of register, 32 in this case)
// W3- Bitmap; 1s specify bits to copy, 0 specifies bits
//      to preserve

            mov     w2, #32      // Number of bits to rotate
            b.al    DistLoop

CopyToW0:
            extr    w0, w1, w0, #1
            lsr     w1, w1, #1
            cbz     w2, Done
DistLoop:
          ❶ sub     w2, w2, #1
            tst     w3, #1
            lsr     w3, w3, #1
            bne     CopyToW0

          ❷ ror     w0, w0, #1
            cbnz    w2, DistLoop
Done:
```

The main entry point to this loop is `DistLoop`. It begins by decrementing the loop counter held in W2 ❶. This code will check the value in W2 to see if the loop is done a little later. Next, the `tst` instruction checks whether bit 0 of the bitmap contains a 1. If it does, the code needs to copy a bit from the LO bit position of W1 into W0; otherwise, it keeps the current bit value.

The `bne` instruction transfers control to `CopyToW0` if it needs to copy a bit from W1; otherwise, it falls through to ❷ if it's going to keep the current bit in W0. The `ror` instruction rotates the existing W0 LO bit into the HO bit position (after 32 iterations of this loop, the bit winds up back in its original position). After the `ror`, the code checks whether the loop has executed 32 times (the `cbnz` instruction). If so, the code exits; otherwise, it repeats.

If the LO bit of W3 was a 1, control transfers to the `CopyToW0` label, which is responsible for shifting the (current) LO bit of W1 into W0. The code at `CopyToW0` uses the `extr` instruction to grab bit 0 from W1 and place it in bit 31 of W0 (shifting bits 1 to 31 in W0 down 1 bit). The `lsr w1, w1, #1` instruction removes the used bit from W1 and places the next bit to merge into W1 in bit position 0. After a quick check to see whether the loop is complete, the code falls down into `DistLoop` and repeats.

NOTE *This code would be a bit simpler if the ARM had an instruction that would rotate a register right by one bit through the carry flag. However, since no such instruction exists, this code has to simulate it by using* `extr`.

The general algorithm for coalescing bits is a tad more efficient than general distribution. Here's the code that will extract bits from W1 via the bitmask in W3 and leave the result in W0:

```
// W0- Destination register
// W1- Source register
// W3- Bitmap with 1s representing bits to copy to W0

        mov    w0, wzr    // Clear destination register.
        b.al   ShiftLoop

ShiftInW0:
        extr    w0, w0, w1, #31
      ❶ lsl     w1, w1, #1

ShiftLoop:
      ❷ tst     w3, #0x80000000
        lsl     w3, w3, #1
        bne     ShiftInW0   // W3 HO bit was set.

      ❸ lsl     w1, w1, #1
        cbnz    w3, ShiftLoop
```

As with the distribution code, the coalescing code loops through the bits, copying one bit at a time from W1 to W0 wherever there are 1 bits in the bitmap ❷. The `extr` instruction creates a 32-bit string from bit 31 of W1 and bits 0 to 30 of W0, then puts the result into W0. On each loop iteration,

the code shifts the bits in W1 one position to the left ❶ ❸ so that the next bit to (possibly) move into W0 is in the HO bit position. Unlike the distribution code, this code will terminate after it processes all the 1 bits present in the bitmap.

Another way to coalesce bits is via table lookup. By grabbing a byte of data at a time (so your tables don't get too large), you can use that byte's value as an index into a lookup table that coalesces all the bits down to bit 0. Finally, you can merge the bits at the low end of each byte together. This may produce a more efficient coalescing algorithm in certain cases. The implementation is left to you.

12.5.2 Creating Packed Arrays of Bit Strings

Though far less efficient, it is possible to create arrays of elements whose size is not a multiple of 8 bits. The drawback is that calculating the "address" of an array element and manipulating it involves a lot of extra work. This section presents a few examples of packing and unpacking array elements that are an arbitrary number of bits long.

Why would you want arrays of bit objects? The answer is simple: space. If an object consumes only 3 bits, you can get 2.67 times as many elements into the same space if you pack the data rather than allocating a whole byte for each object. For very large arrays, this can result in substantial savings. Of course, the cost of saving space is speed: you must execute extra instructions to pack and unpack the data, slowing access to it.

The calculation for locating the bit offset of an array element in a large block of bits is almost identical to the standard array access:

```
Element_Address_in_bits =
    Base_address_in_bits + index × element_size_in_bits
```

Once you calculate the element's address in bits, you must convert it to a byte address (because you must use byte addresses when accessing memory) and extract the specified element. Because the base address of an array almost always starts on a byte boundary, you can use the following equations to simplify this task:

```
Byte_of_1st_bit =
    Base_Address + (index × element_size_in_bits ) / 8

Offset_to_1st_bit =
    (index × element_size_in_bits) % 8
```

For example, suppose you have an array of 200 3-bit objects declared as follows:

```
A03Bobjects:
    .space  (200 * 3)/8 + 2  // "+2" handles truncation.
```

The constant expression in the preceding dimension reserves space for enough bytes to hold 600 bits (200 elements, each 3 bits long). As the comment notes, the expression adds 2 extra bytes at the end to ensure you don't lose any odd bits and to allow you to access 1 byte beyond the end of the array when storing data to the array. (Losing odd bits wouldn't occur in this example because 600 is evenly divisible by 8, but in general you can't count on this; adding 2 extra bytes usually won't hurt.)

Now suppose you want to access the *i*th 3-bit element of this array. You can extract these bits with the following code:

```
// Extract the ith group of 3 bits in AO3Bobjects
// and leave this value in W0:

        mov     w2, wzr         // Put i / 8 remainder here.
        ldr     w0, [fp, #i]    // Get the index into the array.

        mov     w4, #3
        mul     w0, w0, w4      // W0 = W0 * 3 (3 bits/element)
        ubfiz   w2, w0, #0, #3  // W2 = LO 3 bits of W0
        lsr     w0, w0, #3      // W0 / 8 -> W0 and W0 % 8 -> W2

// Okay, fetch the word containing the 3 bits you want to
// extract. You have to fetch a word because the last bit or two
// could wind up crossing the byte boundary (that is, bit
// offset 6 and 7 in the byte).

        lea     x1, AO3Bobjects

        ldrh    w0, [x1, x0]    // Fetch 16 bits.
        lsr     w0, w0, w2      // Move bits down to bit 0.
        And     w0, w0, #0b111  // Remove the other bits.
```

Inserting an element into the array is a little more difficult. In addition to computing the base address and bit offset of the array element, you must also create a mask to clear out the bits in the destination where you're going to insert the new data. The following code inserts the LO 3 bits of W0 into the *i*th element of the `AO3Bobjects` array:

```
Masks:
            .hword    ~ 0b0111,           ~ 0b00111000
            .hword    ~ 0b000111000000,   ~ 0b1110
            .hword    ~ 0b01110000,       ~ 0b001110000000
            .hword    ~ 0b00011100,       ~ 0b11100000
              .
              .
              .

// Get the index into the array (assume i is a local variable):

            ldr     w1, [fp, #i]

// Use LO 3 bits as index into Masks table:
```

```
            and     w2, w1, #Ob111
            lea     x4, Masks
            ldrh    w4, [x4, w2, uxtw #1] // Get bitmask.

// Convert index into the array into a bit index.
// To do this, multiply the index by 3:

            mov     w3, #3
            mul     w1, w1, w3

// Divide by 8 to get the byte index into W1
// and the bit index (the remainder) into W2:

            and     w2, w1, #Ob111
            lsr     w1, w1, #3

// Grab the bits and clear those you're inserting:

            lea     x5, AO3Bobjects
            ldrh    w6, [x5, w1, uxtw #O]
            and     w3, w4, w6

// Put your 3 bits in their proper location:

            lsl     wO, wO, w2

// Merge bits into destination:

            orr     w3, w3, wO

// Store back into memory:

            strh    w3, [x5, w1, uxtw #O]
```

Assuming `AO3Bobjects` contained all 0s, `i` contained 5, and W0 (the value to insert) was 7 upon executing this code, the first couple of bytes would contain `0x38000` after the execution of this code sequence. Because each element is 3 bits, the array looks like

```
000 000 000 000 000 111 000 000 000 000 00 ...
```

where bit 0 is the leftmost bit. Flipping the 32 bits around to make them more readable, and grouping them in blocks of 4 bits to make it easy to convert to hexadecimal, we get

```
0000 0000 0000 0011 1000 0000 0000 0000
```

which is `0x38000`.

This code uses a lookup table (`Masks`) to generate the masks needed to clear out the appropriate position in the array. Each element of this array contains all 1s, except for three 0s in the position you need to clear for a

given bit offset. Note the use of the NOT operator (~) to invert the constants in the table.

12.5.3 Searching for Bits

A common bit operation is to locate the end of a run of bits. A special case of this operation is to locate the first (or last) set or clear bit in a 16-, 32-, or 64-bit value. This section explores ways to handle this special case.

First set bit means the first bit in a value, scanning from bit 0 toward the high-order bit, which contains a 1. A similar definition exists for the *first clear bit*. The *last set bit* is the first bit in a value, scanning from the high-order bit toward bit 0, which contains a 1. Likewise, a similar definition exists for the *last clear bit*.

One obvious way to scan for the first or last bit is to use a shift instruction in a loop and count the number of iterations before you shift out a 1 (or 0). The number of iterations specifies the position. Here's some sample code that checks for the first set bit (from bit 0) in W0 and returns that bit position in W1:

```
          mov  w1, #31    // Count off the bit positions in W1.
TstLp:    adds w0, w0, w0 // Check whether the current bit
                          // position contains a 1.
          bcs  Done       // Exit loop if it does.
          subs w1, w1, #1 // Decrement your bit position counter by 1.
          bpl  TstLp      // Exit after 32 iterations.
Done:
```

Note that this code returns –1 in W1 if W0 has no set bits.

Searching for the first (or last) set bit is such a common operation that Arm added an instruction specifically to accelerate this process: `clz` (count leading 0 bits). In particular, the `clz` instruction counts the number of leading 0s, which tells you the position of the most significant set bit. Consider the following code:

```
clz w0, w0
sub w0, w0, #31
neg w0, w0
```

This code computes the bit position of the 1 in the highest position in W0 (leaving the result in W0). This produces –1 if W0 contains 0 (no leading set bits).

Don't forget that `cls` doesn't count leading *set* bits but leading *sign* bits. To count the number of leading (HO) bits containing 1s, invert the number and use `clz` to count the leading 0 bits. To count the number of trailing 0 or 1 bits (that is, a bit run of 0s or 1s starting at the LO bit position), use the `rbit` instruction to reverse the bits and then count the HO bits you're interested in.

12.5.4 Merging Bit Strings

Another common bit string operation is to produce a single bit string by merging, or interleaving, bits from two sources. For example, the following code sequence creates a 32-bit string by merging alternate bits from two 16-bit strings:

```
          mov  w2, #16
          lsl  w0, w0, #16     // Put LO 16 bits in the HO
          lsl  w1, w1, #16     // bit positions.
MergeLp:  extr w3, w3, w0, #31 // Shift a bit from W0 into W3.
          Extr w3, w3, w1, #31 // Shift a bit from W1 into W3.
          lsl  w0, w0, #1      // Move on to the next bit in
          lsl  w1, w1, #1      // W0 and W1.
          subs w2, w2, #1      // Repeat 16 times.
          bne  MergeLp
```

This particular example merges two 16-bit values together, alternating their bits in the result value. For a faster implementation of this code, unroll the loop to eliminate one-third of the instructions.

With a few slight modifications, you can merge four 8-bit values together, or merge other bit sets from the source strings. For example, the following code copies bits 0 to 5 from W0, bits 0 to 4 from W1, bits 6 to 11 from W0, bits 5 to 15 from W1, and finally bits 12 to 15 from W0:

```
bfi  w3, w0, #0, #6   // W0[0:5] to W3[0:5]
bfi  w3, w1, #6, #5   // W1[0:4] to W3[6:10]
lsr  w0, w0, #6
bfi  w3, w0, #11, #6  // W0[6:11] to W3[11:16]
lsr  w1, w1, #5
bfi  w3, w1, #17, #11 // W1[5:15] to W3[17:27]
lsr  w0, w0, #6
bfi  w3, w0, #28, #4  // W0[12:15] to W3[28:31]
```

This code produces the result in W3, extracting the bits from W0 and W1.

12.5.5 Scattering Bits from a Bit String

You can also extract and distribute bits in a bit string among multiple destinations, known as *scattering bits*. The following code takes the 32-bit value in W0 and distributes alternate bits among the LO 16 bits in the W1 and W3 registers:

```
           ldr     w0, =0x55555555
           mov     w3, wzr
           mov     w1, wzr
           mov     w2, #16     // Count the loop iterations.
ExtractLp: adds    w0, w0, w0  // Extract odd bits to W3.
           adc     w3, w3, w3
           adds    w0, w0, w0  // Extract even bits to W1.
           adc     w1, w1, w1
```

```
        subs    w2, w2, #1  // Repeat 16 times.
        bne     ExtractLp
```

This code produces 0xffff in W1 and 0x0000 in W0.

12.5.6 Searching for a Bit Pattern

Another bit-related operation you may need is the ability to search for a particular bit pattern in a string of bits. For example, you might want to locate the bit index of the first occurrence of 0b1011 starting at a particular position in a bit string. This section explores simple algorithms to accomplish this task.

To search for a particular bit pattern, you must know four details:

- The pattern
- The length of that pattern
- The bit string to search through, known as the *source*
- The length of the bit string that you're searching

The basic idea behind the search is to create a mask based on the length of the pattern and to mask a copy of the source with this value. You can then directly compare the pattern with the masked source for equality. If they are equal, you're finished; if not, increment a bit position counter, shift the source one position to the right, and try again. You repeat the operation [*length*(*source*) – *length*(*pattern*)] times. The algorithm fails if it does not detect the bit pattern after that number of attempts, because it has exhausted all the bits in the source operand that could match the pattern's length.

Listing 12-2 searches for a 4-bit pattern.

```
// Listing12-2.S
//
// Demonstration of bit string searching

        #include    "aoaa.inc"

        .text
        .pool
ttlStr: wastr   "Listing 12-2"
noMatchStr:
        wastr   "Did not find bit string\n"

matchStr:
        wastr   "Found bit string at posn %d\n"

        proc    getTitle, public
        lea     x0, ttlStr
        ret
        endp    getTitle

        proc    asmMain, public
```

```
        locals  am
        word    pattern
        word    source
        word    mask
        byte    am.stk, 64
        endl    am

        enter   am.size

        // Initialize the local variables this code
        // will use:

        mov     w0, #0b1011110101101100
        str     w0, [fp, #source]
        mov     w0, #0b1011
        str     w0, [fp, #pattern]
        mov     w0, #0b1111
        str     w0, [fp, #mask]

        // Here's the code that will search for the
        // pattern in the source bit string:

        mov     w2, #28             // 28 attempts because 32 - 4 = 28
                                    // (len(src) - len(pat))
        ldr     w3, [fp, #mask]     // Mask for the comparison.
        ldr     w0, [fp, #pattern]  // Pattern to search for
        and     w0, w0, w3          // Mask unnecessary bits in W0.
        ldr     w1, [fp, #source]   // Get the source value.
ScanLp: mov     w4, w1              // Copy the LO 4 bits of W1.
        and     w4, w4, w3          // Mask unwanted bits.
        cmp     w0, w4              // See if you match the pattern.
        beq     Matched
        sub     w2, w2, #1          // Repeat specified number of times.
        lsr     w1, w1, #1
        cbnz    w1, ScanLp

// Do whatever needs to be done if you failed to
// match the bit string:

        lea     x0, noMatchStr
        bl      printf
        b.al    Done

// If you get to this point, you matched the bit string.
// You can compute the position in the original source as 28 - W2.

Matched:
        mov     x1, #28
        sub     x1, x1, x2
        mstr    x1, [sp]
        lea     x0, matchStr
        bl      printf
Done:
        leave                       // Return to caller.
        endp    asmMain
```

Here's the `build` command and sample output for Listing 12-2:

```
% ./build Listing12-2
% ./Listing12-2
Calling Listing12-2:
Found bit string at posn 2
Listing12-2 terminated
```

As you can see, this program properly located the bit pattern in the source.

BIT-STRING SCANNING AND STRING MATCHING

Bit-string scanning is a special case of string matching. String matching is a well-studied problem in computer science, and many of the algorithms you can use for it are applicable to bit-string matching as well.

These algorithms are beyond the scope of this chapter, but as a preview of how this works, you'd execute a function passing the pattern and the current source as parameters and use the result as an index into a lookup table to determine the number of bits you can skip. These algorithms let you skip several bits, rather than shifting only once for each iteration of the scanning loop, as the previous algorithm does.

12.6 Moving On

Assembly language is well known for its powerful bit-manipulation capabilities, difficult to replicate in an HLL. This chapter described those capabilities for the 64-bit ARM CPU. It began with definitions useful for describing bit operations, then introduced a bevy of instructions that manipulate bit data. This chapter also discussed using the condition code flags in the PSR as bit data, along with the instructions to manipulate those flags, specifically the negative (N), zero (Z), carry (C), and overflow (V) flags.

After discussing the basic set of bit-manipulation instructions, this chapter covered applications of those instructions, including packing and unpacking bit strings, inserting one bit string into another, extracting a bit string from a source string, coalescing and distributing bits, working with packed arrays of bits, searching for bit strings, merging bit strings, and scattering bits from a bit string.

For the most part, this chapter concludes the discussion of new ARM assembly language instructions. The remaining chapters discuss the application of these instructions and various software-engineering topics. The next chapter, for example, focuses on macros you can use to simplify your assembly language programs.

12.7 For More Information

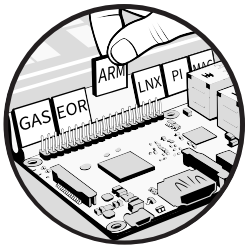
- The ultimate book on bit twiddling is *Hacker's Delight*, 2nd edition, by Henry S. Warren Jr. (Addison-Wesley Professional, 2012). While this book uses the C programming language for examples, almost all the concepts it discusses apply to assembly language programs as well.

TEST YOURSELF

1. What general instruction(s) would you use to clear bits in a register?
2. What instruction could use you to clear a bit, specified by a bitmask containing 1s where you want 0s in the destination register?
3. What general instruction would you use to set bits in a register?
4. What general instruction would you use to invert (selected) bits in a register?
5. What general instruction would you use to test a bit (or group of bits) for 0 or 1 in a register?
6. What single instruction could you use to extract a run of bits from a register?
7. What single instruction could you use to position and insert a run of bits in a register?
8. What instruction allows you to search for the most-significant set bit in a register?
9. How would you search for the last clear bit in a register (that is, the lowest-order bit containing a 0)?
10. What instruction can you use to count the number of bits in each byte of a vector register?

13

MACROS AND THE GAS COMPILE-TIME LANGUAGE



This chapter discusses the Gas compile-time language (CTL), including its macro expansion facilities. A *macro*, the CTL equivalent of a procedure, is an identifier that the assembler will expand into additional text. This allows you to abbreviate large amounts of code with a single identifier. Gas's macro facility is a computer language inside a computer language; that is, you can write short programs inside a Gas source file whose purpose is to generate other Gas source code to be assembled by Gas.

The Gas CTL consists of macros, conditionals (`if` statements), loops, and other statements. This chapter covers many Gas CTL features and how you can use them to reduce the effort needed to write assembly language code.

13.1 The Gas Compile-Time Language Interpreter

Gas is actually two languages rolled into a single program. The *runtime language* is the standard ARM/Gas assembly language you've been reading about in the previous chapters. This is called the runtime language because the programs you write execute when you run the executable file. Gas contains an interpreter for a second language, the *Gas CTL*.

Gas source files contain instructions for both the Gas CTL and the runtime program, and Gas executes the CTL program during assembly (compilation). Once Gas completes assembly, the CTL program terminates. Figure 13-1 shows the relationship between compile-time and runtime with respect to the Gas assembler and your assembly language source code.

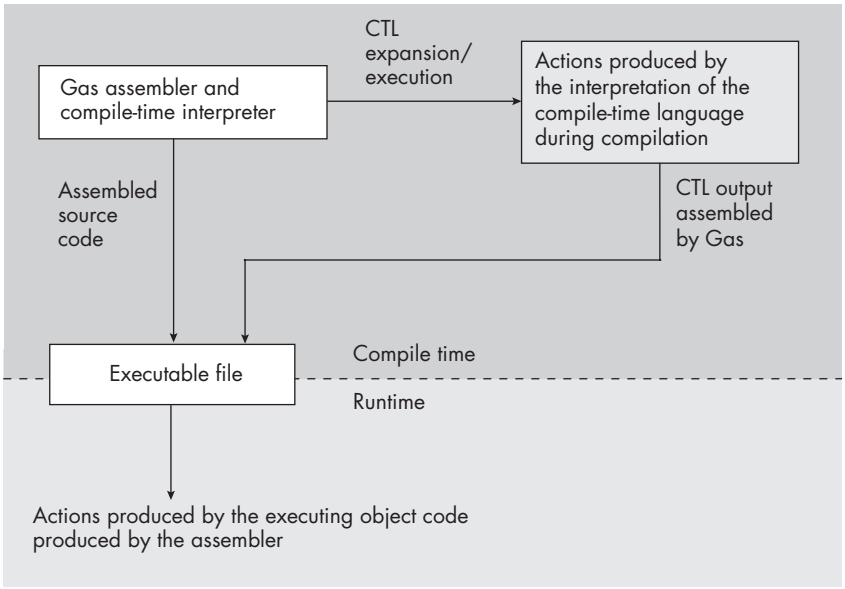


Figure 13-1: Compile-time versus runtime execution

The CTL application is not a part of the runtime executable that Gas emits, although the CTL application can write part of the runtime program for you. In fact, this is the major purpose of the CTL. Using automatic code generation, the CTL gives you the ability to emit repetitive code easily and elegantly. By learning how to use the Gas CTL and applying it properly, you can potentially develop assembly language applications as rapidly as HLL applications (even faster because Gas's CTL lets you create very HLL, or VHLL, constructs).

13.2 The C/C++ Preprocessor

The Gas CTL consists of two separate language processors: the Gas built-in macro processor and the C/C++ preprocessor (CPP). As noted in Chapter 1,

standard Gas assembly language source files use the *.s* suffix. However, if you specify *.S* as the suffix, Gas will run the source file through the CPP prior to processing the file (see Figure 13-2). The CPP emits a temporary source file (with the *.s* suffix), which the Gas assembler then assembles into object code.

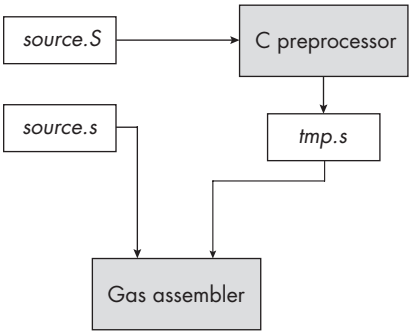


Figure 13-2: C preprocessor processing by Gas

It is extremely important to remember that the CPP runs independently of Gas and prior to it assembling the assembly language source file. In particular, Gas's macro processing takes place *after* the CPP runs. Therefore, you cannot use Gas statements, symbols, or Gas macros to affect the operation of the CPP. Later in this chapter, I'll point out the areas where you must take care when mixing the macro facilities of the two languages. This section describes the various features of the CPP.

13.2.1 The #warning and #error Directives

When writing macros by using CPP, you'll sometimes encounter a problem (such as a bad parameter argument) that you'll want to report as an error or a diagnostic message during the assembly process. To do so, you can use the `#warning` and `#error` diagnostic statements with the following syntax:

```
#error arbitrary text
#warning arbitrary text
```

These statements must appear on a source line by themselves; nothing other than whitespace (spaces and tabs) should appear on the line before the # character. (Technically, whitespace is allowed to appear between the # and the error or the warning tokens, but good programming style dictates keeping them together.)

During assembly (or, more precisely, while CPP is processing the source file), the system should display a diagnostic message and print the line containing the `#error` or `#warning` statement, including all the arbitrary text up to the end of the line. By convention, most programmers surround the error or warning message (the arbitrary text) in quotes, but this isn't absolutely necessary.

If CPP encounters any `#error` statements, it will terminate the assembly after CPP is done scanning the source file, without running the Gas assembler to assemble that file. In this case, you will need to modify the source file as necessary to eliminate the error message before Gas even gets a chance to process the file (and, for example, report on any errors in the assembly language source code).

If the CPP encounters any `#warning` statements, it will print the appropriate message during assembly but will allow assembly to proceed after the CPP is done preprocessing the source file. Hence, you can use the `#warning` statement to display arbitrary text during the assembly and preprocessing process.

13.2.2 Compile-Time Constant Definition with CPP

You can use the CPP `#define` statement to create constant definitions in your source file:

```
#define identifier arbitrary_text
```

When CPP processes the source file, it will replace any following occurrences of *identifier* by *arbitrary_text*. Programmers commonly use this statement, for example, to define manifest (named) constants in their source file, as in the following example:

```
#define pi 3.14159
```

In an assembly language program, you would normally use the `.equ`, `.set`, or `=` directives to define named constants, as in the following example:

```
maxCnt = 10
```

However, various bugs in Gas might not allow you to use these constants the way you like. Consider the following:

```
pi = 3.14159
    .
    .
    .
   .double pi
```

If you try to assemble this, Gas will complain that `3.14159` is not a valid constant and that `pi` is not a valid floating-point constant. (The Clang assembler under macOS will accept `pi = 3.14159` but will still complain that `pi` is not a valid floating-point constant.) However, if you replace this with

```
#define pi 3.14159
   .
   .
   .
   .double pi
```

then Gas will assemble the code just fine because CPP will preprocess the source file and replace each occurrence of `pi` with `3.14159`. Therefore, when Gas actually sees the source file, it will find

```
    .double 3.14159
```

which is perfectly acceptable. This is a good example of why it helps to use CPP in your Gas source files: it provides you with capabilities, such as real constant definitions, that you wouldn't normally have with Gas alone.

Because CPP does a textual replacement of the defined identifier wherever it finds that identifier (outside of a string or character constant), you're not limited to using `#define` for numeric constants. You can supply character constants, string constants, or even arbitrary text (including nothing) after the `#define`:

```
#define hw "Hello, World!"
```

You could even do something like the following if you really prefer the `xor` mnemonic over the `eor` mnemonic:

```
#define xor eor
    .
    .
    .
  xor x1, x0, x2
```

Although redefining instruction mnemonics like this is generally considered poor programming practice, ARM does it all over the place with its "instruction aliases." If it's good enough for the ARM, there is no reason you can't do it if it makes your code more readable to you.

Another important use of the `#define` statement is to create symbols that the CPP can recognize. The CPP is blissfully unaware of all identifiers appearing in your source file, except those you create with the `#define` statement. As you'll see starting in the next section, you'll sometimes want to use various expressions in a CPP CTL statement involving named constants. Those named constants must be defined with the `#define` statement, not one of Gas's equate directives.

13.2.3 CPP Compile-Time Expressions

Certain CPP statements allow simple arithmetic expressions involving constants. The arithmetic operators are the usual C arithmetic operators, including these:

```
+  -  *  /  %  == != <  <=  >  >=  !  ~  && ||  &  | << >>
```

Note that CPP supports only (signed) 64-bit integer and character expressions and will report an error if you attempt to use floating-point or

string constants. You can use a named constant in a CPP CTL expression as long as you've previously declared that name with a `#define` statement.

You can use the following CPP built-in function in CPP CTL expressions:

```
defined( identifier )
```

This function returns 1 if the identifier was previously defined in a `#define` statement; it returns 0 if no such definition exists (note that you can also use the GCC `-D` *`identifier=value`* command line option to define symbols). The `defined()` function recognizes only symbols defined in a `#define` statement, a good example of the "important use" mentioned in the preceding section. If you pass a normal Gas assembly language identifier here, the function will return 0 even if the definition occurred earlier in the source file.

13.2.4 Conditional Assembly

The CPP provides several statements that allow you to make decisions when processing the source file. Here are those directives:

```
#if expression
 .
 .
 .
#elif expression  // This is optional and may appear multiple times.
 .
 .
 .
#else             // This is optional.
 .
 .
 .
#endif

#ifdef identifier
 .
 .
 .
#else             // This is optional.
 .
 .
 .
#endif

#ifndef identifier
 .
  .
 .
#else             // This is optional.
 .
 .
 .
#endif
```

During preprocessing, CPP will evaluate *expression*. If it yields a nonzero value (true), the `#if` or `#elif` (else if) statements will process the text up to the next `#elif`, `#else`, or `#endif` statement.

If *expression* evaluates to false, CPP will skip over the following text (up to the next `#elif`, `#else`, or `#endif` statement) and will not write that text to the temporary output file. Therefore, Gas will not assemble that text during the assembly phase.

Remember that this conditional processing happens during preprocessing (assembly), not at runtime. This is not a generic `if/then/elseif/else/endif` statement you'd find in an HLL. The conditional compilation statements control whether the instructions will actually appear in the final object code, activity that should be familiar to anyone who has used conditional compilation in an HLL such as C/C++.

The `#ifdef` statement is equivalent to the following:

```
#if defined( identifier )
 .
 .
 .
#endif
```

The CPP checks the identifier to see whether it was previously defined with a `#define` statement (or the `-D` command line option). If so, CPP processes the text after the `#if` (or `#ifdef`) and up to the `#endif` (or up to an `#elif` or `#else` statement, if present).

The `#ifndef` (if not defined) statement is equivalent to this:

```
#if !defined( identifier )
 .
 .
 .
#endif
```

The `#ifdef` and `#ifndef` statements are common in code that is written for different execution environments. Consider the following example:

```
#ifdef isMacOS

    Code written for macOS

#else

    Assume the code was written for Linux or Pi OS.

#endif
```

Prior to this point, if the following statement appeared, then the former code would compile the section for macOS:

```
#define isMacOS
```

Had this definition not appeared, the code would compile the Linux or Pi OS code.

Another common use of conditional assembly is to introduce debugging and testing code into your programs. As a typical debugging technique, many Gas programmers insert print statements at strategic points, enabling them to trace through their code and display important values at various checkpoints. A big problem with this technique, however, is that they must remove the debugging code prior to completing the project. Moreover, programmers often forget to remove some debugging statements, creating defects in the final program. Finally, after removing a debugging statement, these programmers often discover that they need that same statement to debug a different problem at a later time. Hence, they must constantly insert and remove the same statements over and over again.

Conditional assembly provides a solution to this problem. By defining a symbol (say, `debug`) to control debugging output in your program, you can activate or deactivate *all* debugging output by modifying a single line of source code, as the following code fragment demonstrates:

```
// Uncomment to activate debug output or -D debug on the command line.

// #define debug
      .
      .
      .
     #ifdef debug

        #warning *** DEBUG build

        mov  x1, [fp, #i]
        mstr x1, [sp]
        lea  x0, debugMsg
        bl   printf

     #else

        #warning *** RELEASE build

     #endif
```

As long as you surround all debugging output statements with an `#if` statement like the one in the preceding code, you don't have to worry about debugging output accidentally appearing in your final application. Commenting out the debug symbol definition will automatically disable all such output (or, better yet, just use the `-D debug` command line option to turn on output when you want it). Likewise, you can leave the debugging statements from your programs in your code even after they've served their immediate purpose, since conditional assembly makes them easy to deactivate. Later, if you decide you need to view this same debugging information during assembly, you can reactivate it by defining the debug symbol.

13.2.5 CPP Macros

Previously, this chapter used the `#define` statement to define compile-time constants, a special case of a *macro definition*. This section describes CPP macro definitions and expansion in more depth, including a discussion of macro parameters, variable argument lists, and other CPP macro definition features. By using this information, you will be able to create and use CPP macros in your Gas assembly language source files.

13.2.5.1 Functional Macros

A macro is a mechanism that CPP uses to replace an identifier with arbitrary text. When defining constants with `#define`, you're telling CPP to replace every following occurrence of that identifier with the text, which happens to be the constant.

However, CPP provides a second type of macro, the *functional macro*, which behaves more like a (compile-time) function supporting an arbitrary number of arguments. The following example demonstrates a single-argument macro:

```
#define lcl( arg1 )  [fp, #arg1]
   .
   .
   .
  ldr w0, lcl( varName )
```

This last statement expands to

```
  ldr w0, [fp, #varName]
```

because the macro `lcl` expands to `[fp, #varName]`. The CPP calls these *functional macros* because their invocation resembles a function call in the C programming language.

13.2.5.2 CPP Macro Arguments

Functional macros support an arbitrary number of arguments. You can specify zero arguments thusly

```
#define zeroArgs()  text
```

where `zeroArgs()` will expand into the specified *`text`*.

This is a difference between the following two macro declarations that manifests when you invoke them:

```
#define noArgs  text1
#define zeroArgs() text2
```

You invoke the first macro with `noArgs` and the second macro with `zeroArgs()`. If the macro declaration has an empty set of parentheses, the macro

invocation must also include the empty parentheses. You can use this declaration scheme to differentiate between constant declarations and macro declarations.

You can also specify two or more arguments in a `#define` statement:

```
#define twoArgs( arg1, arg2 )  text to expand
```

The following is an example of a `twoArgs()` invocation:

```
mov w0, twoArgs( 1, 2 )
```

When invoking `twoArgs()`, you must supply exactly two parameters, or Gas will report an error. In general, the number of arguments you supply in a macro invocation must exactly match the parameter list in the `#define` declaration.

When CPP is processing a macro invocation, it usually separates actual parameters by scanning for commas. It ignores commas appearing in string or character constants, or in expressions surrounded by parentheses or square brackets:

```
  singleArg( "Strings can contain commas, that's okay!" )
  singleArg( ',' )    // Also okay
  singleArg( (1,2) )  // (1,2) is a single argument.
```

From CPP's perspective, each of these macro invocations has a single argument.

13.2.5.3 Macro Argument Expansion Issues

As any experienced C programmer knows, you must be careful when specifying macro parameters to avoid unintended consequences during expansion, especially if they involve arithmetic expressions. Consider the following macro:

```
#define reserve( amt )  amt + 1
```

Now consider the following macro invocation:

```
.space  reserve( Size ) * 2  // Size is a constant.
```

The expectation here is to reserve twice as much space as the `reserve()` macro would normally specify. However, consider the actual expansion of this macro:

```
.space  Size + 1 * 2
```

Gas's arithmetic rules specify that multiplication has a higher precedence than addition, so this expands to `Size` + 2 rather than (`Size` + 1) × 2, the desired expansion. C programmers work around this issue by always

surrounding macro expansions (that expand to arithmetic expressions) with parentheses, and they always surround the macro parameters themselves with parentheses, as shown in the following example:

```
#define reserve( amt )  ((amt) + 1)
```

This usually resolves the issue of a macro expansion in the middle of an arithmetic expression.

For the same reason, it's generally a good idea to enclose any expression you pass as a macro argument in parentheses:

```
.space  reserve( (Size + 5) ) * 2
```

Because the expansion could produce unintended consequences based on operator precedence (for example, suppose the `reserve` definition were `amt * 2`, which would expand to `Size + 5 * 2` if you didn't surround the actual parameter expression with parentheses).

13.2.5.4 Variable Argument Lists

CPP provides a mechanism for specifying a variable number of parameters:

```
#define varArgs( ... )  text to expand
```

To reference the arguments, use the predefined `__VA_ARGS__` symbol (which begins and ends with two underscores). The CPP will substitute the entire set of arguments in place of `__VA_ARGS__`. *This includes all the commas appearing in the varying argument list.* Consider the following macro definition and invocation:

```
#define bytes(...) __VA_ARGS__
   .
   .
   .
  .byte bytes( 1, 2, 3, 4 )
```

The `.byte` statement expands to the following:

```
.byte 1, 2, 3, 4
```

A varying argument list allows zero actual parameters, so the invocation `bytes()` is perfectly legal (and will expand to the empty string, given the previous definition). Therefore

```
.byte bytes()
```

will expand to

```
.byte
```

which, interestingly enough, will not produce an error (Gas does not generate any code for this statement).

Although the expansion of the entire argument list is sometimes useful, you'll much more often need to pick off individual arguments in the varying argument list, as discussed in the following two sections.

13.2.5.5 Macro Composition and Recursive Macros

CPP does not support recursive macro invocations. If a macro's name appears in the expansion text, CPP will simply emit that name as text to be assembled by Gas. This is unfortunate because recursion would be very useful for handling iteration, as CPP doesn't provide any looping constructs. In section 13.2.5.9, "Iteration with Macros," ~~on page~~ 757, I'll provide a workaround; in the meantime, I will discuss how CPP handles macro expansion when one macro invokes another.

Consider the following macro definitions and invocations:

```
#define inc(x) ((x)+1)
#define mult(y) ((y)*2)
   .
   .
   .
  .byte mult( inc( 5 ) )
```

When CPP encounters a macro invocation within the parameter list of another macro invocation, it will expand the parameter before passing that text on to the enclosing macro invocation. The expansion of the `.byte` statement happens in two steps:

```
.byte mult( ((5) + 1) )  // First step
```

and then

```
.byte (((5) + 1) * 2)   // Second step
```

which, of course, is equal to:

```
.byte 12
```

Now consider the following example:

```
#define calledMacro(x) mov w0, x
#define callingMacro(y) calledMacro(y)
   .
   .
   .
  callingMacro( 5 )
```

When CPP encounters the macro invocation at the end of this example, it will expand `callingMacro( 5 )` to the following text:

```
calledMacro(5)
```

CPP will then expand this macro to the following:

```
mov w0, 5
```

As long as CPP continues to find a macro invocation in the expanded text (except for a recursive invocation), it will continue to expand those invocations, regardless of how many iterations this process requires.

NOTE *The `#define calledMacro(x) mov w0, x` macro should really be `#define calledMacro(x) mov w0, #x` using the syntax presented thus far in this book. However, the `#` is a CPP operator (stringify, described a little later) that will turn the actual parameter `5` into the string `"5"`. Fortunately, Gas accepts a plain constant in place of `#constant` for the immediate addressing mode in this example.*

13.2.5.6 Macro Definition Limitations

The syntax for a CPP macro is the following

```
#define identifier( parameters ) text to expand \n
```

where (*parameters*) is optional and \n represents a newline character. CPP does not allow any newline characters within *text to expand*. Therefore, a macro can expand only to a single line of text.

CPP does allow macro definitions like the following

```
#define multipleLines( parms ) text1 \
                               text2 \
                                .
                                .
                                .
                               textn
```

which spreads a macro definition across *n* lines in the source file. Each line, except the last, must be terminated with a backslash character (\) immediately followed by a newline character.

Although this macro is physically split across multiple source lines, it is still a single line of text because the CPP will delete all the newline characters following the backslash characters. Therefore, you cannot create a macro like the following:

```
#define printi( i )        \
        lea   x0, iFmtStr  \
        lea   x1, i        \
        ldr   x1, [x1]     \
        mstr  x1, [sp]     \
```

```
        bl    printf
         .
         .
         .
        printi( var )
```

This macro definition won't work because CPP will expand all the assembly language statements on a single line (with spaces between them), generating a syntax error. Sadly, Gas doesn't seem to provide a general mechanism for supplying multiple assembly statements on the same line. Therefore, you cannot use CPP macros that expand to multiple assembly language statements. (Fortunately, Gas macros do allow this, as you'll learn in section 13.3.4, "Gas Macros," on page 765.)

NOTE *On some CPUs, Gas allows the use of the semicolon (`;`) character to put multiple statements on the same line. However, the ARM treats semicolons as line comment characters; one semicolon is equivalent to two forward slashes (`//`). Your mileage may vary with Gas; for example, under Pi OS you can use the semicolon as a statement separator.*

CPP's single-line macro definitions have another serious drawback: you cannot incorporate CPP's conditional compilation statements (such as `#if`) into a macro, since the conditional compilation statements must appear at the beginning of a source line. This is unfortunate, as the ability to make decisions in a macro would be useful. Fortunately, there are a couple of workarounds.

First, you can put macro definitions within conditional compilation sequences, as shown in the following example:

```
#ifdef isMacOS
#define someMacro text to expand if macOS
#else
#define someMacro text to expand if not macOS
#endif
```

This sequence has two macros, only one of which will be defined within a given assembly. You can therefore put the (presumably) macOS-only code in the first macro definition and the Linux code in the second definition.

Using two separate macro definitions like this will work in some, but not all, instances; sometimes you really do need the ability to put conditional text within a macro expansion. I'll provide a second workaround to address this need in section 13.2.5.8, "Conditional Macros," on page 756.

13.2.5.7 Text Concatenation and the Stringify Operator

CPP provides two special operators for manipulating textual data: the concatenation operator and the stringify operator. This section describes those two operators.

A *token* is an entity, such as an identifier or operator, recognized by the C/C++ language. The CPP *concatenation operator* (`##`) combines two tokens in

a macro, forming a single token. For example, within a macro body, the following text produces the single token `identifier`:

```
ident ## ifier
```

Any amount of whitespace may appear between the tokens and the `##` operator. CPP will remove all the whitespace and join the two tokens together—as long as the result is a legal C/C++ token. If a macro parameter identifier appears on either side of the `##` operator, CPP will expand that parameter to the actual parameter's text before doing the concatenation. Alas, if you pass another macro as the parameter, CPP does not properly expand the parameter:

```
#define one 1
#define _e  e
#define produceOne(x) on ## x
   .
   .
   .
  produceOne(_e)  // Expands to on_e
```

As a sneaky workaround to this problem, you can create a concatenation macro to create the identifier for later expansion:

```
#define produceOne  1
#define concat( x, y ) x ## y

    mov w0, #concat( produce, One )
```

This will generate the statement

```
mov w0, #produceOne
```

which then expands to this:

```
mov w0, #1
```

Section 13.8, "For More Information," on page 792 includes links to sites that describe CPP's text concatenation operator more fully.

NOTE *The concatenation operator is legal only within a CPP macro. The CPP will ignore ## everywhere else in the source file, leaving it up to Gas to handle it (which generally produces an error, as ## is not a legal token in ARM assembly language).*

The second CPP operator is the *stringify operator* (`#`), which you can use within a macro body as follows

```
# parmID
```

where *parmID* is the name of one of the macro parameters. The stringify operator will expand the parameter and convert it to a string by surrounding

the text with quotes. This is why the earlier `#define calledMacro(x) mov w0, #x` macro did not work properly—it stringified the argument rather than leaving it as an integer constant.

13.2.5.8 Conditional Macros

Although you cannot include conditional compilation directives (`#if`, `#else`, and so on) inside macro bodies, it is possible to create conditional macros by pulling some tricks with the CPP (otherwise known as *abusing CPP*). Here's an example of a conditional macro that implements an if . . . then macro expansion you can use to select a particular expansion based on other definitions in your source code:

```
if_else( expression ) (true expansion) (false expansion)
```

The (*true expansion*) and (*false expansion*) are compile-time expressions. The `if_else` macro will evaluate *expression*; if it evaluates to a nonzero value, this statement will be replaced by (*true expansion*). If *expression* evaluates to false, this statement will be replaced by (*false expression*). Here's a simple example:

```
.asciz if_else( MacOS ) ("macOS") ("LinuxOS")
```

If `MacOS` is nonzero, this produces the string `"macOS"`; otherwise, it produces the string `"LinuxOS"`.

The `if_else` macro is quite complex, and I won't describe how it works here; it is C rather than assembly language, which puts it beyond the scope of this text. See section 13.8, "For More Information," on page 792 for resources on this topic. Here's the implementation of `if_else` from one of those resources, by Jonathan Heathcote:

```
#define _secondArg_( a, b, ... ) b

#define _is_probe_( ... ) _secondArg_( __VA_ARGS__, 0 )
#define _probe_() ~, 1

#define _cat_( a, b ) a ## b

#define _not_(x) _is_probe_( _cat_( _not_, x ))
#define _not_0 _probe_()

#define _bool_(x) _not_( _not_( x ))

#define if_else( condition ) _if_else_( _bool_( condition ))
#define _if_else_( condition ) _cat_( _if_ , condition )

#define _if_1( ... ) __VA_ARGS__ _if_1_else
#define _if_0( ... )             _if_0_else

#define _if_1_else( ... )
#define _if_0_else( ... ) __VA_ARGS__
```

Unlike the conditional compilation directives, you can embed the `if_else` macro into the bodies of other macros. Consider the following code:

```
#define macStr( x ) if_else( x ) ("macOS")("Linux")
```

Invoking this macro yields the string `"macOS"` or `"Linux"`, depending on the compile-time value of the parameter.

13.2.5.9 Iteration with Macros

The `__VA_ARGS__` feature in CPP is useful for passing a group of arguments as a single argument. However, it would be nicer if we could process each argument in a varying parameter list one at a time by iterating through the list in order. Unfortunately, the CPP doesn't provide any statements to support iteration. Because CPP doesn't support recursion, we can't even (directly) use recursion to handle iteration.

However, if you abuse the CPP a bit, a limited form of recursion is possible, as this section demonstrates. The ultimate goal of this section is to create a macro, let's call it `map`, that will execute a single-argument macro on each argument in a varying parameter list. Ideally, you'd call `map` thusly

```
map( macroName, arg1, arg2, ..., argn )
```

and the map function would generate *n* calls to `macroName`:

```
macroName( arg1 )
macroName( arg2 )
  .
  .
  .
macroName( argn )
```

The following set of macros, along with the `if_else` macro from the previous section, provides this functionality (also from Heathcote; see section 13.8, "For More Information," on page 792 for implementation details):

```
// Include the macro definitions for if_else from
// the previous section here.

#define _firstArg_( a, ... ) a

#define _empty_()

#define eval( ... ) eval1024(__VA_ARGS__)
#define eval1024( ... ) eval512( eval512( __VA_ARGS__ ))
#define eval512( ... ) eval256( eval256( __VA_ARGS__ ))
#define eval256( ... ) eval128( eval128( __VA_ARGS__ ))
#define eval128( ... ) eval64( eval64( __VA_ARGS__ ))
#define eval64( ... ) eval32( eval32( __VA_ARGS__ ))
#define eval32( ... ) eval16( eval16( __VA_ARGS__ ))
#define eval16( ... ) eval8( eval8( __VA_ARGS__ ))
```

```
#define eval8( ... ) eval4( eval4( __VA_ARGS__ ))
#define eval4( ... ) eval2( eval2( __VA_ARGS__ ))
#define eval2( ... ) eval1( eval1( __VA_ARGS__ ))
#define eval1( ... ) __VA_ARGS__

#define _defer1(m) m _empty_()
#define _defer2(m) m _empty_ _empty_()()
#define _defer3(m) m _empty_ _empty_ _empty_()()()
#define _defer4(m) m _empty_ _empty_ _empty_ _empty_()()()()

#define _has_args(...) _bool_(_firstArg_(_end_of_args __VA_ARGS__)())
#define _end_of_args() 0

#define map( m, _firstArg_, ... )       \
  m( _firstArg_ )                       \
  if_else( _has_args(__VA_ARGS__ ))(    \
    _defer2( _map )()( m, __VA_ARGS__ ) \
  )(                                    \
    /* Do nothing, just terminate */    \
  )
#define _map() map
```

The `eval` macro provides a limited amount of recursion on the macro you pass as an argument (up to 1,024 levels of recursion, which will allow for a varying parameter listing containing up to 1,024 entries). In order for the `map` macro to recurse, you must enclose it in an `eval` invocation. Consider the following example:

```
#define inc(x) (x+1),
     .
     .
     .
    .byte eval( map( inc, 1, 2, 3, 4, 5, 6, 7)) 9
```

The comma at the end of the `inc` macro is required because the invocations of `inc` will emit expressions of the form $(1 + 1), (2 + 1), \ldots, (7 + 1)$ on the same line. The `.byte` directive requires commas between these expressions.

Also note that the value `9` appears after the `eval` invocation. This is because the last expression, `(7+1)`, will have a comma after it, so this statement must supply the last value manually. It would be possible to modify the `map` macro to emit this comma for all but the last argument, or you could modify the `inc` macro to check for a special sentinel value (such as a negative number) to terminate the list (without emitting the value or the comma).

13.2.5.10 Command Line Defines

If you look back at the *build* shell script in section 1.10.1, "Assembling Programs Under Multiple OSes," on page 36, you'll see that GCC has a command line parameter that specifies the OS in use:

```
-D isLinux=1    (for Linux and Pi OS)
-D isMacOS=1    (for macOS)
```

These command line parameters are roughly equivalent to the following two statements:

```
#define isLinux 1
#define isMacOS 1
```

In many instances, you can refer to these definitions in your source file exactly as though you had placed these `#define` statements at the beginning of the file.

As noted, these parameters are *roughly* equivalent—not exactly equivalent—to the `#define` statements. Within macro bodies, these symbols might not be expanded as normal defined symbols would be. To use these symbols in macro bodies, it's generally a good idea to explicitly create some `#define` symbols by using code such as the following, then refer to `myIsLinux` and `myIsMacOS` within your macros:

```
#ifdef isLinux
  #define myIsLinux 1
  #define myIsMacOS 0
#else
  #define myIsLinux 0
  #define myIsMacOS 1
#endif
```

If you are compiling your Gas source files directly from the command line via `gcc`, you can define other symbols by using the `-D` command line option. See the GCC documentation for details. Note that the *build* script will not pass `-D` arguments to GCC, but you can easily modify *build* if you want to define other symbols.

13.2.5.11 The CPP #undef Statement

The CPP allows you to forget a defined symbol by using the `#undef` statement

```
#undef identifier
```

where *`identifier`* is a symbol previously defined with a `#define` statement. (If the symbol is undefined upon `#undef` execution, CPP will simply ignore the statement.)

Undefining a symbol allows you to redefine it (for example, to give it another value). If you attempt to redefine a symbol without first undefining it, CPP will report an error.

13.3 Components of the Gas CTL

Gas's CTL facilities are closer (than CPP) to what most assembly language programmers expect from a macro expansion system. Although CPP's macro facilities are useful for certain purposes, macro assembly programming generally requires a much more powerful macro system. Therefore, learning Gas's macro facilities is essential. This section covers the components of the Gas CTL.

13.3.1 Errors and Warnings During Assembly

The Gas `.err` directive is similar to CPP `#error`. During assembly, Gas will display an error message (printing the source line containing the `.err` statement). Gas will not generate an object file if it encounters an `.err` directive. For example, the following generates an error message at runtime:

```
.err
```

Gas also supports the `.error` directive. Its syntax is shown here:

```
.error "String error message"
```

The operand must be a string constant surrounded by quotes. During assembly, Gas will display the specified error message and will not generate an object file if it encounters an `.error` directive in the source file.

Gas also supports a `.warning` directive similar to the CPP `#warning` statement, with the following syntax:

```
.warning "String warning message"
```

Again, the operand must be a string constant surrounded by quotes. If Gas encounters the `.warning` directive during assembly, it will display the specified warning message. If no errors are in the source files, only warnings, Gas will generate an object file. You can therefore use the `.warning` directive as an assembly-time print statement.

Keep in mind the difference between the CPP `#warning` and `#error` statements and the Gas `.warning` and `.error` directives: the CPP statements execute during the preprocessor pass, prior to the assembly process, while the Gas directives execute after the preprocessor pass, during assembly. If there are any `#error` statements that execute, CPP will terminate the assembly process without running the assembler (so the Gas directives won't execute in that situation).

13.3.2 Conditional Assembly

Gas also supports a set of conditional assembly (or conditional compilation) directives similar to CPP's `#if`/`#endif` statements. The basic directives are as follows:

```
.if expression1

  Statements to assemble if expression is nonzero

.elseif expression2  // Optional, may appear multiple times

  Statements to assemble if expression2 is nonzero

.else  // Optional, but only one instance is legal

  Statements to assemble if expression1, expression2, ...
  and so on, were all 0

.endif
```

As a general rule, you should prefer Gas's conditional assembly directives in your source file over the CPP conditional compilation statements. Only use CPP's conditional compilation statements when conditionally processing source code that contains other CPP statements (such as `#define` statements), or testing whether a symbol has been defined in CPP with the `#ifdef` or `#ifndef` statement. You cannot test whether a CPP symbol has been defined by using Gas's conditional assembly statements because all CPP symbols will be expanded (and won't be present) when Gas is assembling the source file.

The Boolean expression appearing after `.if` or `.elseif` must be an absolute expression (see "Relocatable and Absolute Expressions" on page 176). Within conditional assembly expressions, false is a zero result and true is anything else.

Gas supports several variants of the `.if` directive:

`.ifdef` *`symbol`* Assembles the following section if *`symbol`* is defined prior to that point in the source file. CPP symbols (created with `#define`) are expanded prior to assembly, so their use may not work as expected in this directive.

`.ifb` *`text`* Assembles the following section if the operand field is blank. You generally use this directive to test for a blank macro parameter (*`text`* is typically a macro parameter name that could expand to the empty string).

`.ifc` *`text1`*`,` *`text2`* Compares *`text1`* to *`text2`* and assembles the following section if they are equal. The string comparison ignores any whitespace around the text. The string *`text1`* comprises all characters from the first non-whitespace character after `.ifc` up to the first comma. The string *`text2`* is all text after the comma (ignoring leading whitespace) up to the end of the line (also ignoring whitespace at the end of the line). If you need to include whitespace in the string, you may optionally surround the strings with apostrophes (single quotes). Generally, you would use this statement to compare two macro parameter expansions to see if the parameters are equal.

.ifeq *expression* Assembles the following code if *expression* is equal to 0.

.ifeqs "*string1*", "*string2*" Assembles the following code if the two strings are equal. The strings must be surrounded by double quotes.

.ifge *expression* Assembles the following code if *expression* is greater than or equal to 0.

.ifgt *expression* Assembles the following code if *expression* is greater than 0.

.ifle *expression* Assembles the following code if *expression* is less than or equal to 0.

.iflt *expression* Assembles the following code if *expression* is less than 0.

.ifnb *text* Assembles the following section if the operand field is not blank. You generally use this directive to test for a nonblank macro parameter (*text* is typically a macro parameter name).

.ifnc *text1*, *text2* Compares *text1* to *text2* and assembles the following section if they are not equal. The string comparison ignores any whitespace around the text. The string *text1* comprises all characters from the first non-whitespace character after `.ifnc` up to the first comma. The string *text2* is all text after the comma (ignoring leading whitespace) up to the end of the line (also ignoring whitespace at the end of the line). If you need to include whitespace in the string, you may optionally surround the strings with apostrophes (single quotes). Generally, you would use this statement to compare two macro parameter expansions to see if the parameters are not equal.

.ifndef *symbol*, .ifnotdef *symbol* Assembles the following section if *symbol* is not defined prior to that point in the source file. Note that CPP symbols (created with `#define`) get expanded prior to assembly, so their use may not work as expected in this directive. The `.ifnotdef` directive is a synonym for `.ifndef`.

.ifne *expression* Assembles the following code if *expression* is not equal to 0. This directive is a synonym for `.if`.

.ifnes "*string1*", "*string2*" Assembles the following code if the two strings are not equal. The strings must be surrounded by double quotes.

Conditional assembly statements in Gas may appear anywhere an instruction mnemonic is legal. Typically, they appear on a line by themselves, though it is legal (if unusual) for a label to appear on the same line. In that case, the label will be associated with the next instruction or directive that emits code.

As you can see, Gas provides a wide variety of conditional assembly statements that are quite a bit more powerful and flexible than the CPP conditional compilation statements. This is another reason to use Gas's conditional assembly statements over CPP's.

13.3.3 Compile-Time Loops

Unlike CPP, Gas's CTL provides three looping constructs to easily generate data and unroll loops: `.rept`, `.irp`, and `.irpc`. The following subsections describe these directives.

13.3.3.1 .rept . . . .endr

The `.rept` directive repeats a block of statements a fixed number of times. The syntax for this compile-time loop is the following:

```
.rept expression

  Statements to repeat

.endr
```

Gas will evaluate *expression* and repeat the block of statements between the `.rept` and `.endr` directives the specified number of times. If *expression* evaluates to 0, Gas will ignore all the statements up to the `.endr`, generating no code.

The following example will initialize a 32-element byte array with the values 0 through 31:

```
.set  i, 0     // Initialize array element value.
.rept 32
.byte i
.set  i, i + 1 // Increment array element value.
.endr
```

You aren't limited to data values in a `.rept` loop but can use `.rept` to unroll loops as well:

```
// Zero out an eight-dword array:

    .set  ofs, 0
    .rept 8
    str   xzr, [x0, #ofs]
    .set  ofs, ofs + 8
    .endr
```

This is equivalent to the following code

```
    str xzr, [x0, #0]
    str xzr, [x0, #8]
    str xzr, [x0, #16]
    str xzr, [x0, #24]
    str xzr, [x0, #32]
    str xzr, [x0, #40]
    str xzr, [x0, #48]
    str xzr, [x0, #56]
```

which unrolls the loop eight times.

13.3.3.2 .irp . . . .endr

The .irp (indefinite repeat) looping directive takes the following form:

```
.irp identifier, comma-separated-list-of-values
 .
 .
 .
.endr
```

This loop repeats for each item in *comma-separated-list-of-values*. In the body of the loop, you can refer to the current value by using *identifier*; the following example

```
.irp  i, 0, 1, 2, 3
.byte \i
.endr
```

is equivalent to

```
.byte 0
.byte 1
.byte 2
.byte 3
```

which unrolls the loop for each .irp argument.

13.3.3.3 .irpc . . . .endr

The third compile-time looping construct, .irpc, is similar to .irp but processes a string of text rather than a list of values:

```
.irpc identifier, text
 .
 .
 .
.endr
```

Here, *identifier* is a symbol that will take on the value of each character in the string specified by *text*. Note that *text* is a bare sequence of characters; do not surround it by double or single quotes unless you want the .irpc loop to process those punctuation marks along with the other characters in the string. The .irpc loop will execute once for each character in the string, and *identifier* will expand to that character on each iteration. For example

```
.irpc x, acde
.byte '\x'
.endr
```

expands to this:

```
.byte 'a'
.byte 'c'
.byte 'd'
.byte 'e'
```

Note that *identifier* expands even within character and string constants (as '\x' did in this example). In this example, had you not enclosed \x in single quotes, the .irpc loop would have expanded to

```
.byte a
.byte c
.byte d
.byte e
```

which would have generated an error if the symbols a, c, d, and e were not defined somewhere in the program (I conveniently skipped b, which would have expanded to a branch instruction mnemonic).

13.3.4 Gas Macros

Gas provides macro facilities via the .macro and .endm directives. The syntax for a macro definition is as follows

```
.macro identifier { parameter_list }

  Statements to expand on macro invocation

.endm
```

where the { and } characters mean the parameter list is optional (you don't include these characters in the macro definition). The following subsections describe the various components of a Gas macro, along with important semantic information concerning macros.

13.3.4.1 Macro Parameters

Macro parameters can take one of the following four forms:

identifier This first form, just a simple identifier, is the most common. Unless you supply one of the suffixes appearing in the other three options, this syntax tells Gas that the parameter is optional. If you do not supply an appropriate actual parameter value when invoking the macro, Gas will substitute the empty string (a blank parameter) when expanding the parameter in the macro's body.

identifier=expression Like the first form, this specifies a parameter that can be optional, except that Gas will give the identifier the value of the expression rather than an empty string if the macro invocation doesn't supply a parameter value.

***identifier*:req** Specifies that the macro argument must be supplied when invoking the macro; if it is missing, Gas will respond with an error message.

***identifier*:vararg** Allows for a varying parameter list (zero or more arguments separated by commas). Gas will expand this macro parameter to the entire list of values, including the commas separating the values.

In standard Gas syntax, a space separates the macro name and the first parameter (if any). I've found that with the ARM assembler, sneaking in a comma works fine too (your mileage may vary).

Within a macro body, use a token of the form `\`*`identifier`*—where *`identifier`* is one of the macro's declared formal parameters—to expand a parameter. For example, the following macro demonstrates the expansion of the `value` argument:

```
.macro oneByte value
.byte  \value
.endm
 .
 .
 .
oneByte 1  // Expands to .byte 1
```

A macro definition can have zero or more parameters. If you supply more than one parameter, you must separate each formal parameter with a comma. Additionally, if you specify a `vararg` parameter, it must be the last parameter declared in the `.macro` statement. Here's an example of a slightly complex macro:

```
.macro bytes yy:req, zz=0, tt, ss:vararg
.byte  \yy
.byte  \zz
.byte  \tt
.byte  \ss
.endm
```

When you invoke this macro, you must supply at least one actual parameter (because `yy` is a required parameter). For example

```
bytes 5
```

expands to this:

```
.byte 5
.byte 0  // Because zz expands to 0 by default
.byte    // Argument tt expands to the empty string.
.byte    // Argument ss also expands to the empty string.
```

Note that if a data directive such as .byte does not have any operands, Gas will ignore that statement and not generate any code to the object file.

Here's another invocation of bytes that demonstrates full argument expansion

```
bytes 5, 4, 3, 2, 1, 0
```

which expands to:

```
.byte 5        // yy expansion
.byte 4        // zz expansion
.byte 3        // tt expansion
.byte 2, 1, 0  // ss expansion
```

This example worked out well because the .byte directive allows comma-separated operands. However, what if you want to expand a vararg parameter where comma-separated operands are not legal? Consider the following macro:

```
.macro  addVals theVals:vararg
add     x0, x0, #\theVals
.endm
```

An invocation of addVals such as

```
addVals 1, 2
```

will generate an error because

```
add x0, x0, #1, 2
```

is syntactically incorrect. You can solve this problem by using an .irp loop inside the macro to process a vararg parameter:

```
.macro  addVals theVals:vararg
.irp    valToAdd, \theVals
add     x0, x0, #\valToAdd
.endr
.endm
```

The addVals 1, 2 invocation will now emit the following:

```
add x0, x0, #1
add x0, x0, #2
```

Gas will expand a macro just about anywhere it appears within a macro body. Consider the following macro:

```
.macro select which
lea    x1, var\which
```

```
ldr    w1, [x1]
.endm
```

Assuming you've defined the `var0` and `var1` symbols somewhere, the invocation `select 0` generates the following

```
lea x1, var0
ldr w1, [x1]
```

while the invocation `select 1` generates this:

```
lea x1, var1
ldr w1, [x1]
```

Suppose you want to supply the prefix of the name, rather than the suffix, as the macro argument in this example. A first attempt at this won't work:

```
.macro select2 which
lea    x1, \whichvar
ldr    w1, [x1]
.endm
```

The problem, of course, is that Gas interprets `\whichvar` as the expansion of a parameter named `whichvar`. To separate a parameter name from the following text, use the `\()` token:

```
.macro select2 which
lea    x1, \which\()var
ldr    w1, [x1]
.endm
```

An invocation such as `select2 my` will now properly expand to

```
lea    x1, myvar
ldr    w1, [x1]
```

which creates the name `myvar`, as intended.

13.3.4.2 Macro Parameters with String Constants

Gas's macros have a "feature" that can bite you if you're not careful: if you pass a string constant as a formal parameter, Gas will strip the quotes from the string when expanding that parameter. For example, consider the following macro and invocation:

```
.macro myStr theStr
.asciz \theStr
```

```
.endm
 .
 .
 .
myStr "hello"
```

This expands to the following:

```
.asciz hello
```

Unless you've defined the symbol `hello` with an appropriate value, this will generate an error. The correct way to do this is as follows:

```
.macro myStr theStr
.asciz "\theStr"
.endm
 .
 .
 .
myStr "hello"
```

This code properly generates the following:

```
.asciz "hello"
```

One possible use for this is passing parameters that contain commas and whitespace as a single argument to a macro. I will leave it up to you to figure out other abuses of this "feature" in Gas. Personally, I consider it a bug, and I'd be afraid to use this feature because Gas could remove this behavior in a future version of the assembler.

13.3.4.3 Recursive Macros

Unlike CPP, Gas fully supports recursive macros. Consider the following example (adapted from the Gas manual):

```
.macro  sum from=0, to=5
.long   \from
.ifgt   \to-\from
sum     "(\from+1)",\to
.endif
.endm
```

A macro invocation of the form `sum 0, 5` generates the following code:

```
.long   0
.long   1
.long   2
.long   3
.long   4
.long   5
```

The `sum` macro uses conditional assembly statements to prevent endless recursion. Though you could more easily iterate over five values by using the `.rept` (or `.irp`) directives, sometimes recursion is a better solution than iteration.

The `.irp` and `.rept` directives are a better fit for simple iteration. Recursion is better for processing recursive data structures, such as lists and trees, passed as macro arguments, or if you need to reverse the arguments passed to a macro (I give an example of this in the next section).

13.3.4.4 The .exitm Directive

The `.exitm` directive allows you to prematurely terminate the expansion of a macro. Its syntax is as follows:

```
.exitm
```

When Gas encounters `.exitm` during a macro expansion, it immediately stops the expansion and ignores the rest of the macro's body. Of course, just placing an `.exitm` directive in the middle of a macro body (other than for testing purposes) is not especially useful—why write the rest of the macro body if it's going to be ignored? Instead, you'll generally find an `.exitm` macro inside a conditional assembly block like the following:

```
.macro reverse first, args:vararg
.ifb    \first
.exitm  // Quit recursion if no more arguments.
.endif
reverse \args
.byte   \first
.endm
```

The `.exitm` directive terminates recursion when the argument list is empty. The recursive invocation passes all but the first argument to reverse. As you may already have guessed, this macro will generate the bytes specified as parameters in reverse order to the file. For example, `reverse 0, 1, 2, 3` generates

```
.byte 3
.byte 2
.byte 1
.byte 0
```

reversing the arguments passed to the `reverse` macro.

13.3.4.5 The \@ Operator

Within a macro, Gas will convert the token `\@` to a string of digits specifying the total number of macros it has expanded during assembly. You can use this operator, typically along with the `\()` token, to create macro-local symbols. The following macro provides a trivial example of this usage:

```
        .macro  lclsym sym
        b       a\()\@
a\()\@:
        .endm
```

Multiple expansions of this macro generate a unique symbol by suffixing a string of digits to the end of a.

13.3.4.6 The .purgem Directive

The .purgem directive deletes a previously defined Gas macro. It is similar to the CPP #undef statement. Normally, if you try to redefine a Gas macro, Gas will generate an error. Use the .purgem directive to delete the macro if you want to redefine it.

Note that Gas will generate an error if you attempt to purge a macro that has not already been defined. Unfortunately, the .ifdef (and comparable) conditional assembly statement does not recognize macro symbols; so there is no way to check whether a macro has been defined before using the .purgem directive; you have to ensure that the macro symbol exists prior to using this directive.

13.4 The aoaa.inc Header File

Throughout this book, I've used the *aoaa.inc* header file in examples without discussing what it contains. Now that you've been introduced to CPP and Gas macro and CTL facilities, the time has come to fulfill my promise in Chapter 1 that I would explain, section by section, how this header file works.

I'll go over the source code for *aoaa.inc* piece by piece, in order to annotate and explain each of its components. The first section is the usual header that appears at the beginning of an include file:

```
// aoaa.inc
//
// "Magic" header file for The Art of ARM Assembly
// that smooths over the differences between Linux
// and macOS
//
// Assertions:
//
// Either isMacOS or isLinux has been
// defined in the source file (using #define) prior
// to including this source file. This source file
// must be included using #include "aoaa.inc"
// NOT using the Gas .include directive.
```

To begin, the *aoaa.inc* header file assumes that the source file that includes *aoaa.inc* is being assembled using the *build* shell script. Among other things, the *build* script will include one of the following command

line options on the gcc command line that assembles the source file, as appropriate for the OS under which you're running GCC (and Gas):

```
-D isMacOS=1
-D isLinux=1
```

Because these symbols are defined for use by CPP (not Gas), the source file must have a *.S* suffix, and you must run the CPP on this file, meaning you need to assemble the file by running the gcc executable rather than the as (Gas) executable.

The next section of the source file handles multiple inclusions:

```
// aoaa.inc (cont.)
//
#ifndef aoaa_inc
#define aoaa_inc 0
```

This #ifndef and #define sequence is the standard way to prevent problems if a program includes a header file more than once. The first time CPP includes this file, the symbol aoaa_inc will not be defined; therefore, CPP will process the text after the #ifndef statement. The very next statement defines the aoaa_inc symbol. Should the assembly source file that included *aoaa.inc* include it a second time, the aoaa_inc symbol will be defined, so CPP will ignore everything up to the matching #endif (which happens to be at the end of the source file).

As the earlier comment states, you must include the *aoaa.inc* header file by using the CPP statement #include, rather than Gas's .include directive. This is because the *aoaa.inc* file contains several CPP statements (including #ifndef), and CPP will never see the *aoaa.inc* file if you include it via .include. Remember, Gas statements are processed long after CPP has executed and quit.

Next, the *aoaa.inc* header file sets up defines for various symbols to handle macOS- and Linux-specific code:

```
// aoaa.inc (cont.)
//
// Make sure all OS symbols are
// defined and only one of them
// is set to 1:

#ifndef isMacOS
    #define isMacOS (0)
#else
    #undef isMacOS
    #define isMacOS (1)
#endif

#ifndef isLinux
    #define isLinux (0)
#else
    #undef isLinux
```

```
    #define isLinux (1)
#endif

// Make sure exactly one of the OS symbols is set to 1:

#if (isMacOS+isLinux) != 1
    #error "Exactly one of isMacOS or isLinux," \
                        " must be 1"
#endif
```

This block of conditional compilation statements ensures that both `isLinux` and `isMacOS` are defined and are given appropriate values for the OS. The command line parameters supplied by the *build* script will define only one of these two symbols. These statements ensure that both are defined and are assigned appropriate Boolean values (0 for false, 1 for true).

Next up, the *aoaa.inc* header file defines some symbols required under macOS:

```
// aoaa.inc (cont.)
//
// Do macOS-specific stuff here:

#if isMacOS

    // Change all the C global function
    // names to include a leading underscore
    // character, as required by macOS (these
    // definitions allow you to use all the
    // same names in example programs in
    // macOS and Linux). This list includes
    // all the C stdlib functions used by
    // AoAA example code.

    #define asmMain  _asmMain
    #define acos     _acos
    #define asin     _asin
    #define atan     _atan
    #define cos      _cos
    #define exp      _exp
    #define exp2     _exp2
    #define getTitle _getTitle
    #define free     _free
    #define log      _log
    #define log2     _log2
    #define log10    _log10
    #define malloc   _malloc
    #define pow      _pow
    #define printf   _printf
    #define readLine _readLine
    #define sin      _sin
    #define sqrt     _sqrt
    #define strcat   _strcat
    #define strchr   _strchr
```

```
    #define strcmp   _strcmp
    #define strcpy   _strcpy
    #define strlen   _strlen
    #define strncat  _strncat
    #define strncpy  _strncpy
    #define strstr   _strstr
    #define strtol   _strtol
    #define tan      _tan
    #define write    _write

    #define __errno_location ___error
```

Under macOS, external symbols like the C stdlib function names have a leading underscore. Under Linux, the symbols appear without the underscore. The `#define` statements in the previous code snippet replace several common C stdlib function names with the underscore-prefixed version. This allows calls in this book to use consistent names under both macOS and Linux.

These defines work only for the C stdlib functions appearing in this list. If you decide to call other stdlib functions (or use other external symbols), you'll have to explicitly supply the underscore prefix character or add additional `#define` statements to this list.

The `lea` macro also has a macOS-specific implementation:

```
// aoaa.inc (cont.)
//
// lea (Load Effective Address) macro.
// Correctly loads the address of
// a memory object into a register, even
// on machines that use position-independent
// executables (PIE):

.macro  lea, reg, mem
    adrp    \reg,\mem@PAGE
    add     \reg, \reg, \mem@PAGEOFF
.endm
```

As noted in Chapters 1 and 7, the `lea` macro expands to two instructions that load the address of a symbol into a 64-bit register. The main reason for including this macro (rather than explicitly writing these two instructions everywhere `lea` appears in this book) is that the two instructions are slightly different, depending on whether the code is being assembled for macOS or Linux. This version of the `lea` macro generates the code for macOS.

Next up are the `mstr`, `mstrb`, and `mstrh` macros, which also have macOS-specific implementations:

```
// aoaa.inc (cont.)
//
// mstr Assembles to a str instruction under macOS
// mstrb
// mstrh
```

```
.macro      mstr, operands:vararg
str         \operands
.endm

.macro      mstrb, operands:vararg
strb        \operands
.endm

.macro      mstrh, operands:vararg
strh        \operands
```

Linux and macOS handle varying parameter lists differently. Under Linux, you continue to pass the first eight parameters in X0 through X7, while under macOS, you pass them both in the registers and on the stack. The `mstr`, `mstrb`, and `mstrh` macros expand to code that stores a register onto the stack when operating under macOS (as you'll see shortly, the Linux versions expand to nothing).

The Clang assembler (the macOS version of Gas) does not support the `.dword` directive; the following macro implements this for macOS. Under macOS, the *aoaa.inc* header file therefore includes a macro to supply this missing directive, mapping it to the equivalent `.quad` directive:

```
// aoaa.inc (cont.)
//
// macOS's assembler doesn't have .dword,
// define it here:

.macro  .dword, value:vararg
    .quad   \value
.endm
```

Earlier chapters used the `vparm`*n* macros to pass variables to the `printf()` function. Because the API for varying parameters differs between macOS and Linux, there are separate definitions for the two OSes. Here is their macOS implementation:

```
// aoaa.inc (cont.)
//
// Macros to load parameters 2..8 onto
// the stack for macOS when calling
// a variadic (variable parameter list)
// function, such as printf().
//
// Note that parameter 1 still goes into X0.

.macro  vparm2, mem
lea     x1, \mem
ldr     x1, [x1]
str     x1, [sp]
.endm
```

```
.macro  vparm3, mem
lea     x2, \mem
ldr     x2, [x2]
str     x2, [sp, #8]
.endm

.macro  vparm4, mem
lea     x3, \mem
ldr     x3, [x3]
str     x3, [sp, #16]
.endm

.macro  vparm5, mem
lea     x4, \mem
ldr     x4, [x4]
str     x4, [sp, #24]
.endm

.macro  vparm6, mem
lea     x5, \mem
ldr     x5, [x5]
str     x5, [sp, #32]
.endm

.macro  vparm7, mem
lea     x6, \mem
ldr     x6, [x6]
str     x6, [sp, #40]
.endm

.macro  vparm8, mem
lea     x7, \mem
ldr     x7, [x7]
str     x7, [sp, #48]
.endm        .endm
```

Next come the Linux-specific implementations of these macros:

```
// aoaa.inc (cont.)

#elif isLinux == 1

    // Do Linux (no-PIE)-specific stuff here:

    .macro  lea, reg, mem
        adrp    \reg,\mem
        add     \reg, \reg, :lo12:\mem
    .endm

    // mstr  assembles to nothing under Linux
    // mstrb
    // mstrh

    .macro      mstr, operands:vararg
    .endm
```

```
    .macro      mstrb, operands:vararg
    .endm

    .macro      mstrh, operands:vararg
    .endm

    .macro  vparm2, mem
    lea     x1, \mem
    ldr     x1, [x1]
    .endm

    .macro  vparm3, mem
    lea     x2, \mem
    ldr     x2, [x2]
    .endm

    .macro  vparm4, mem
    lea     x3, \mem
    ldr     x3, [x3]
    .endm

    .macro  vparm5, mem
    lea     x4, \mem
    ldr     x4, [x4]
    .endm

    .macro  vparm6, mem
    lea     x5, \mem
    ldr     x5, [x5]
    .endm

    .macro  vparm7, mem
    lea     x6, \mem
    ldr     x6, [x6]
    .endm

    .macro  vparm8, mem
    lea     x7, \mem
    ldr     x7, [x7]
    .endm

#endif      #endif
```

There are no `#define` statements for stdlib functions because Linux doesn't require the underscore prefix character on external names. As for the parameter-related functions, Linux passes the first eight arguments of a varying parameter list only in the registers, not on the stack. The `mstr`, `mstrb`, and `mstrh` macros thus expand to nothing, while the `vparm`*n* macros expand to code without storing data on the stack.

The remainder of the source file is common to both macOS and Linux. First, the *aoaa.inc* header file contains a few `.global` directives to specify public names for use by the C/C++ program that calls the assembly file:

```
// aoaa.inc (cont.)
//
// Global declarations:

.global asmMain
.global getTitle
.global readLine
.global printf
```

The `printf` definition, strictly speaking, isn't necessary; it's really just an external declaration, and undefined symbols are external by default. I added it simply because almost every sample program in this book calls the `printf()` function.

Gas doesn't actually provide a `.qword` directive. The `.qword` macro renames `.octa` to `.qword` to be consistent with `.word` and `.dword`:

```
// aoaa.inc (cont.)
//
// Generic code for all OSes:

// Gas doesn't have a .qword
// directive. Map .qword to .octa:

.macro  .qword, value:vararg
    .octa   \value
.endm
```

Next up in *aoaa.inc* are the definitions needed for the structure definition macros:

```
// aoaa.inc (cont.)
//
// Macros for structure definitions:

__inStruct          = 0
__inArgs            = 0
__inLocals          = 0
__dir               = 1
```

The `__inStruct`, `__inArgs`, `__inLocals`, and `__dir` compile-time variables maintain information needed to declare structure fields, parameters, and local variables by using the `struct`, `args`, and `locals` macros. The `__in*` variables are Booleans that track whether the program is currently defining a structure, a parameter list, or a set of local variables. Only one of these fields may contain true (nonzero) at a time, though they can all be 0 if you're not declaring the field of any of these objects.

The __dir variable is either 1 or –1. This determines whether successive declarations in these objects have increasing (when __dir is +1) or decreasing (when __dir is –1) offsets. Structures and parameters have increasing offsets, while locals have decreasing offsets.

With those compile-time constants out of the way, here are the actual struct, args, and locals macros:

```
// aoaa.inc (cont.)

               ❶ .macro  struct  name, initialOffset=0
__inStruct          = 1
__inLocals          = 0
__inArgs            = 0
__struct_offset     = \initialOffset
\name\().base       = \initialOffset
__dir               = 1
                    .if     \initialOffset > 0
                    .err
                    error   struct offset must be negative or 0
                    .endif
                    .endm

               ❷ .macro  args  name, initialOffset=16
__inStruct          = 0
__inLocals          = 0
__inArgs            = 1
__struct_offset     = \initialOffset
\name\().base       = \initialOffset
__dir               = 1
                    .endm


               ❸ .macro  locals  name
__inStruct          = 0
__inLocals          = 1
__inArgs            = 0
__struct_offset     = 0
__dir               = -1
                    .endm
```

The struct, args, and locals macros allow you to define structures (records), parameter lists (arguments), and local variables. These macros set up some compile-time variables that track the base address of the object, as well as the direction, positive or negative, by which offsets are assigned to fields of the object.

The struct macro ❶ creates structures (records) by associating a field offset with each member of the structure. The struct macro itself simply initializes the __inStruct, __dir, and *name*.base compile-time variables that maintain information needed when declaring fields of the structure (where *name* is the user-supplied structure name). The __struct_offset CTL maintains a "location counter" within the structure. By default, the struct macro initializes this with 0. However, the person invoking struct can specify a negative

value if they would like to specify that the first fields of the structure appear in memory before the structure's base address. The __dir CTL is initialized with 1 because successive fields in a structure have increasing offsets within the structure.

The args macro ❷ declares parameter lists for a function or procedure, fundamentally the same operation as creating a structure; you are defining part of the activation record, after all. The only real difference is that the starting offset is 16 (this is the offset of the first parameter in the activation record, using the base address specified by the FP register; the saved FP value and the return address consume the double words at offsets 0 and 8, respectively). Because parameters follow in higher addresses, the __dir field is initialized with 1.

The locals macro ❸ declares local variables allocated on the stack below the address held in the FP register. Because successive declarations appear at lower addresses in memory, this macro initializes the __dir field with –1.

The macros for the matching ends, enda, and endl statements appear later in the listing. The following sections describe the data declaration macros that can appear inside a structure:

```
// aoaa.inc (cont.)

                    .macro  salign, size
__salign        = 0xFFFFFFFFFFFFFFFF - ((1 << \size)-1)
__struct_offset = (__struct_offset + (1 << \size)-1) & __salign
                    .endm
```

The salign macro, which should appear only in a struct, args, or locals declaration, adjusts the __struct_offset value (the location counter) so that it is aligned at an offset that is a power of 2 (the power of 2 is specified by the parameter). This macro achieves its purpose by creating a bitmask containing size 0s in the LO bits and 1s in the remaining HO bits. Logically ANDing __struct_offset with this value produces an offset that is aligned to the designed value.

The following macros provide the byte, hword, word, dword, qword, oword, single, and double directives for use in structures:

```
// aoaa.inc (cont.)

                    .macro  byte, name, elements=1
                    .if     __dir > 0
\name               =       __struct_offset
__struct_offset     =       __struct_offset + \elements
                    .else
__struct_offset     =       __struct_offset + \elements
\name               =       -__struct_offset
                    .endif
                    .endm

                    .macro  hword, name, elements=1
```

```
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*2)
                .else
__struct_offset =       __struct_offset + ((\elements)*2)
\name           =       -__struct_offset
                .endif
                .endm

                .macro  word, name, elements=1
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*4)
                .else
__struct_offset =       __struct_offset + ((\elements)*4)
\name           =       -__struct_offset
                .endif
                .endm

                .macro  dword, name, elements=1
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*8)
                .else
__struct_offset =       __struct_offset + ((\elements)*8)
\name           =       -__struct_offset
                .endif
                .endm

                .macro  qword, name, elements=1
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*16)
                .else
__struct_offset =       __struct_offset + ((\elements)*16)
\name           =       -__struct_offset
                .endif
                .endm

                .macro  oword, name, elements=1
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*32)
                .else
__struct_offset =       __struct_offset + ((\elements)*32)
\name           =       -__struct_offset
                .endif
                .endm

                .macro  single, name, elements=1
                .if     __dir > 0
\name           =       __struct_offset
__struct_offset =       __struct_offset + ((\elements)*4)
                .else
```

```
__struct_offset     =       __struct_offset + ((\elements)*4)
\name               =       -__struct_offset
                    .endif
                    .endm

                    .macro  double, name, elements=1
                    .if     __dir > 0
\name               =       __struct_offset
__struct_offset     =       __struct_offset + ((\elements)*8)
                    .else
__struct_offset     =       __struct_offset + ((\elements)*8)
\name               =       -__struct_offset
                    .endif
                    .endm
```

Each macro declares a single scalar or array variable of the specified type (you can specify an array by providing a second argument with the number of elements).

These macros will bump the current location counter, `__struct_offset`, by the size of the variable and assign that offset to the declared name. If `__dir` is negative (locals declarations), the macro first decrements the location counter, then assigns the offset to the name; if `__dir` is positive, the macro assigns the offset and increments the location counter value.

Here are the `ends`, `enda`, and `endl` macros:

```
// aoaa.inc (cont.)
//
// Generate name.size and name.offset constants
// specifying total structure size and the offset
// just beyond the last field.
//
// Also create a macro to be used to declare
// structure variables.

                    .macro ends, name
__inStruct          =      0
\name\().size       =      __struct_offset-\name\().base
\name\().offset = __struct_offset
                    .macro  \name, varName
                    .if     \name\().base < 0
                    .space  __struct_offset-(\name\().base)
                    .endif
\varName:
                    .if     __struct_offset > 0
                    .fill   __struct_offset
                    .endif

                    .endm
                    .endm


                    .macro enda, name
__inArgs            =      0
```

```
\name\().size       =       __struct_offset-\name\().base
                    .endm

                    .macro endl, name
__inLocal           =       0
\name\().size       =       __struct_offset
                    .endm
```

The `ends`, `enda`, and `endl` macros complete a declaration begun by `struct`, `args`, or `locals`. They set to false the Boolean variable that is tracking an open structure, parameter list, or local variables declaration, then set the *name*`.size` equate to the total size of the declarations. The `ends` macro also defines a macro that you can use to declare structure objects in your code.

The `wastr` macro emits a word-aligned string to memory. Here is its implementation:

```
// aoaa.inc (cont.)
//
// Macro to emit a string that is padded with bytes
// so that it consumes a multiple of 4 bytes in memory:

                    .macro   wastr, theStr
                    .asciz   "\theStr"
                    .p2align 2
                    .endm
```

This macro is mainly used in `.text` sections because you must keep all code and labels word-aligned within those sections. Quotes must surround the macro parameter expansion because Gas will strip off the quotes you supply in the actual parameter (see section 13.3.4.2, "Macro Parameters with String Constants," on page 768).

The `proc` and `endp` macros provide syntactical sugar for declaring procedures in an assembly language source file. Here is their implementation:

```
// aoaa.inc (cont.)
//
// Macros for declaration procedures/functions:

public              =       1
                    .macro  proc pName:req, isPublic=0

// If "public" argument is present, emit
// global statement.

                    .if     \isPublic
                    .global _\pName
                    .global \pName
                    .endif

\pName\().isOpenProcDCL = 1
\pName:
_\pName:
                    .endm
```

```
                    .macro  endp pName:req
                    .ifndef \pName\().isOpenProcDCL
                    .err
                    .err    "Not an open procedure"
                    .else
                    .if     \pName\().isOpenProcDCL
                    .else
                    .err
                    .err    "endp name does not match last proc name"
                    .endif
                    .endif
\pName\().isOpenProcDCL = 0
                    .endm
```

Other than emitting the procedure name and (if `isPublic` is 1) the `.global` directive, this macro doesn't really do much.

The `public` equate allows you to specify `public` as a second argument to this macro to tell the assembler to make the symbol global (that is, public). Technically, you could just pass `1` as the second argument, but `public` is more readable.

The `.code` macro simply expands to `.text` and ensures that the location counter is aligned on a word (4-byte) boundary:

```
// aoaa.inc (cont.)
//
// Sanity for ARM code:

                    .macro  .code
                    .text
                    .align  2
                    .endm
```

The `enter` and `leave` macros provide the standard entry and standard exit sequences for a procedure (see section 5.4.4, "Standard Entry Sequence," on page 248 and section 5.4.5, "Standard Exit Sequence," on page 250 for more details):

```
// aoaa.inc (cont.)
//
// Assembly standard entry sequence:

    .macro  enter, localsSize
    stp     fp, lr, [sp, #-16]!
    mov     fp, sp
    .if     \localsSize > 0
    sub     sp, sp, #((\localsSize)+15) & 0xFFFFFFFFFFFFFFF0
    .endif
    .endm

// Assembly standard exit sequence:

    .macro  leave
    mov     sp, fp
```

```
    ldp     fp, lr, [sp], #16
    ret
    .endm
```

In rare circumstances, the `b` (branch) and `b.al` (branch always) instructions may generate an out-of-range error when the target location is too far away from the instruction. In those situations, you can use the `goto` macro to transfer control anywhere in the 64-bit address range of the ARM CPU:

```
// aoaa.inc (cont.)
//
// goto
//
// Transfers control to the specified label
// anywhere in the 64-bit address space:

            .macro  goto, destination
            adr     x16, 0f
            ldr     x17, 0f
            add     x16, x16, x17
            br      x16
0:
            .dword  \destination-0b
            .endm
```

The `goto` macro modifies the values held in the X16 and X17 registers. The ARM API reserves these registers for exactly this purpose, so this is permissible. However, you should always remember that this macro modifies X16 and X17.

The C stdlib provides the magic pointer `__errno_location` to return a pointer to C's `errno` variable in X0. The `getErrno` macro expands to a function call that retrieves this value and returns it in W0:

```
// aoaa.inc (cont.)
//
// getErrno
//
// Retrieves C errno value and returns
// it in X0:

            .extern __errno_location
            .macro  getErrno
            bl      __errno_location
            ldr     w0, [x0]
            .endm
```

The ccne through ccnle equates define useful bit patterns for use by the ccmp instruction:

```
// aoaa.inc (cont.)
//
// Constants to use in the immediate field of
// ccmp:

//          NZCV
    .equ    ccne,   0b0000          // Z = 0
    .equ    cceq,   0b0100          // Z = 1
    .equ    cchi,   0b0010          // C = 1
    .equ    cchs,   0b0110          // Z = 1, C = 1
    .equ    cclo,   0b0000          // Z = 0, C = 0
    .equ    ccls,   0b0100          // Z = 1, C = 0
    .equ    ccgt,   0b0000          // Z = 0, N = V
    .equ    ccge,   0b0100          // Z = 1, N = V
    .equ    cclt,   0b0001          // Z = 0, N! = V
    .equ    ccle,   0b0101          // Z = 1, N! = V

    .equ    cccs,   0b0010          // C = 1
    .equ    cccc,   0b0000          // C = 0
    .equ    ccvs,   0b0001          // V = 1
    .equ    ccvc,   0b0000          // V = 0
    .equ    ccmi,   0b1000          // N = 1
    .equ    ccpl,   0b0000          // N = 0

    .equ    ccnhi,  0b0100          // Not HI = LS, Z = 1, C = 0
    .equ    ccnhs,  0b0000          // Not HS = LO, Z = 0, C = 0
    .equ    ccnlo,  0b0110          // Not LO = HS, Z = 1, C = 1
    .equ    ccnls,  0b0010          // Not LS = HI, C = 1

    .equ    ccngt,  0b0101          // Not GT = LE, Z = 1, N! = V
    .equ    ccnge,  0b0001          // Not GE = LT, Z = 0, N! = V
    .equ    ccnlt,  0b0100          // Not LT = GE, Z = 1, N = V
    .equ    ccnle,  0b0000          // Not LE = GT, Z = 0, N = V
```

The opposite branches are useful when writing code to simulate HLL-like control structures such as if/then/else statements:

```
// aoaa.inc (cont.)
//
// Opposite conditions (useful with all conditional instructions)

#define nhi ls
#define nhs lo
#define nlo hs
#define nls hi
#define ngt le
#define nge lt
#define nlt ge
#define nle gt
```

```
// Opposite branches

        .macro  bnlt, dest
        bge     \dest
        .endm

        .macro  bnle, dest
        bgt     \dest
        .endm

        .macro  bnge, dest
        blt     \dest
        .endm

        .macro  bngt, dest
        ble     \dest
        .endm

        .macro  bnlo, dest
        bhs     \dest
        .endm

        .macro  bnls, dest
        bhi     \dest
        .endm

        .macro  bnhs, dest
        blo     \dest
        .endm

        .macro  bnhi, dest
        bls     \dest
        .endm

#endif // aoaa_inc
```

The `#endif` statement terminates the `#ifndef` statement appearing at the very beginning of the source file.

13.5 Generating Macros by Another Macro

You can use one macro to write another, as the following code demonstrates:

```
// Variant of the proc macro that deals
// with procedures that have varying
// parameter lists. This macro creates
// a macro named "_name" (where name is
// the procedure name) that loads all
// but the first parameters into registers
// X1..X7 and stores those values onto
// the stack.
//
// Limitation: maximum of seven arguments

        .macro  varProc pName:req
```

```
// Create a macro specifically for this func:

        .macro  _\pName parms:vararg
reg     =       1
        .irp    parm, \parms
        .irpc   rnum, 1234567
        .if     reg==\rnum
        lea     x\rnum, \parm
        ldr     x\rnum, [x\rnum]
        mstr    x\rnum, [sp, #(reg-1)*8]
        .endif
        .endr
reg     =       reg + 1
        .endr
        bl      \pName
        .endm

// Finish off the varProc macro (just like
// the proc macro from aoaa.inc):

\pName\().isOpenProcDCL = 1
\pName:
        .endm
```

As the comment states, this variant of the `proc` macro creates a new `varProc` macro you can use to invoke the procedure with HLL-like syntax. Consider the following invocation of this macro:

```
// Demonstrate the varProc macro
// (creates a macro name _someFunc
// that will load parameters and
// then branch to printf):

        varProc someFunc
        b       printf
        endp    someFunc
```

This expands to the following code:

```
        .macro _someFunc, parms:vararg
reg     =       1
        .irp    parm, \parms
        .irpc   rnum, 1234567
        .if     reg==\rnum
        lea     x\rnum, \parm
        ldr     x\rnum, [x\rnum]
        mstr    x\rnum, [sp, #(reg-1)*8]
        .endif
        .endr
reg     =       reg + 1
        .endr
        bl      someFunc
        .endm
```

Invoke the _someFunc macro as follows:

```
    lea x0, fmtStr
    _someFunc i, j
```

This generates the following code

```
    lea x0, fmtStr

// Macro expansion:

    lea x1, i
    ldr x1, [x1]
    lea x2, j
    ldr x2, [x2]
    bl  someFunc
```

and then generates someFunc branches to printf, essentially making this a call to the printf() function.

You could have written a macro to invoke printf() directly (handling the arguments), but you'd have to write such a macro for every function you want to call using HLL-like syntax. Having the varProc macro automatically write this macro spares you this repetitive task.

The varProc macro has the severe limitation that its parameters must be global memory locations (no register, local variables, or other types of memory operands). While this macro may not be especially useful, it serves to demonstrate how one macro can write another. I'll leave it as an exercise for you to expand this macro to handle other types of operands.

Note that if a macro creates another macro, it must use an undefined name when creating the new macro. The examples in this section achieved this by having the invoker supply the new macro name as an argument to the macro that creates the new macro. It is also possible to use the .purgem directive to delete the new macro's name prior to creating it. However, keep in mind that the macro name must already exist when using .purgem to delete it. On the first invocation of the creating macro, this could be a problem since the macro to create might not exist on the first invocation. This is easily remedied by providing an empty macro prior to the first invocation of the creating macro:

```
.macro  createdMacro
  Empty body
.endm

.macro  createMacro

.purgem createdMacro
.macro  createdMacro

  Macro body
```

```
.endm // createdMacro
  .
  .
  .
.endm
```

Because `createdMacro` already exists on the first invocation of `createMacro`, the `.purgem` statement will not generate an error message. After the first invocation, future invocations of `createMacro` will delete the version of `createdMacro` created in the previous invocation.

13.6 Choosing Between Gas Macros and CPP Macros

Glancing at the GNU CPP documentation (*https://gcc.gnu.org/onlinedocs/cpp/*), you'll find that the GNU folks suggest using the Gas built-in macro facilities rather than the CPP. If the people who wrote CPP and Gas suggest using the Gas macro processor rather than CPP, shouldn't you take that suggestion seriously?

If you could use only one macro processor, you could make a strong case for using the Gas macro processor rather than CPP. CPP is not a very powerful macro processor, and macro abuse by C/C++ programmers has given it a bad reputation. Gas's macro processor, in many respects, is clearly superior. Gas's macro facilities would make the better choice between the two.

However, who says you have to use only one or the other? Why not both? CPP and Gas each have strengths and weaknesses that tend to be complementary. Although Gas's macro facilities are more powerful than CPP's, compared with other assemblers out there—such as the Microsoft Macro Assembler (MASM) or the High-Level Assembler (HLA)—Gas's macro facilities aren't particularly impressive. Anything you can use to boost the power of Gas's macros is a good thing. CPP also has some neat features that Gas lacks (such as functional-style macro invocations), and Gas, of course, has many features that CPP lacks (such as multiline macro definitions). If you're careful, using both macro processors gives you abilities above and beyond those of either. That's a good thing, so combining the power of CPP and Gas is something I wholeheartedly recommend.

Given that you have two CTLs available to you when compiling a Gas source file (at least when using a *.S* suffix), which CTL constructs should you use? Most of the time, it doesn't matter much; if either CTL would work, the choice is up to you, though sticking with Gas's CTL is probably the safest choice if all other factors are equal. However, because the two have differing capabilities, at times you might need to pick one over the other.

For example, CPP's macro definitions look like functions and can appear almost anywhere (outside of comments) in the source file. They are great for writing address expression functions. The following code demonstrates the use of functional-style CPP macros:

```
mov x0, #cppMacro(0, x)
```

In this example, a Gas macro wouldn't work because Gas macros don't support functional-style invocations.

On the other hand, CPP macros are limited to producing a single line of text. Therefore, Gas macros are necessary when you want to emit a sequence of instructions:

```
lea x0, someLabel  // Expands to two instructions!
```

Also note that CPP macros can cause you problems if you attempt to use the # symbol (stringify in CPP, immediate operand in Gas) in your macro expansion.

Gas also supports a richer set of conditional assembly statements, along with CTL looping statements. This makes Gas more appropriate for macros that emit a large amount of data or a large number of statements. I generally prefer CPP macros for simple address expression functions and use Gas macros when I need to expand the macro to actual statements.

Deciding whether to use CPP statements versus Gas's `.set`, `.equ`, and `=` directives for simple constant declarations is less clear-cut. For simple integer constants, Gas's equate directives work fine. For nonintegral values, CPP works better. The following example demonstrates defining string constants with CPP macros:

```
#define aStr "Hello, World!"       // This works.
//      .set aStr, "Hello, World!" // This does not.

        .asciz aStr
```

For integer expressions, Gas's equates tend to work better:

```
// #define a (a+1)  // Generally doesn't work as expected
    .set a, a+1     // Works fine (assuming some previous definition of a)
```

Finally, always keep in mind that CPP processes its CTL statements in a preprocessing pass before assembly takes place, meaning CPP is blissfully unaware of symbols and other tokens specific to the assembly language source file. For example:

```
a:   .byte 0
      .
      .
      .
#ifdef a    // a is undefined to CPP; it's a Gas symbol.
 .
 .
 .
#endif
```

```
.ifdef a  // a is defined to Gas's conditional assembly.
 .
 .
 .
.endif
```

The `#ifdef` symbol would believe that the symbol `a` is undefined (even though it was defined earlier in the Gas source file). Remember, CPP conditional compilation statements know only about symbols created with `#define` statements.

13.7 Moving On

The Gas and CPP CTLs greatly expand the capabilities of the Gas assembler. Using these facilities, including constant definitions, macro definitions, conditional compilation and assembly, and so on, can reduce the effort you need to write assembly language source code.

This chapter covered the basic information you need to employ the Gas and CPP CTLs in your assembly language source files, beginning with a discussion of CPP. The second half of this chapter discussed the Gas CTL, including the error and warning directives, conditional assembly, compile-time looping directives, and Gas macros. Next, this chapter described the internal source code for the *aoaa.inc* header file that you've used extensively since Chapter 1. The chapter concluded by contrasting the CPP CTL and the Gas macro facilities, discussing when you should pick one system over the other.

Now that this book has described the Gas CTL, its example code will begin to use the macro facilities, starting in the next chapter.

13.8 For More Information

- You can review the GNU CPP documentation at *https://gcc.gnu.org/onlinedocs/cpp/*.
- Find the Gas documentation (including macros) at *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_node/as_toc.html*. For more on Gas macros in particular, see *https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_node/as_107.html*.
- For information on advanced CPP macros, check out "C Pre-Processor Magic" by Jonathan Heathcote at *http://jhnet.co.uk/articles/cpp_magic* and *https://github.com/18sg/uSHET/blob/master/lib/cpp_magic.h*.
- You can check the C Preprocessor Tricks, Tips, and Idioms GitHub site to find CPP tricks: *https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms*.
- Find the Boost CPP library at *https://www.boost.org/doc/libs/1_57_0/libs/preprocessor/doc/index.html*.

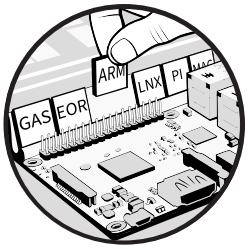
- Embedded Artistry has an article on "Exploiting the Preprocessor for Fun and Profit" by Klemens Morgenstern: *https://embeddedartistry.com/blog/2020/07/27/exploiting-the-preprocessor-for-fun-and-profit/*.
- Learn more about macro metaprogramming from Thomas Mailund's blog at *https://mailund.dk/posts/macro-metaprogramming/*.

TEST YOURSELF

1. What does *CTL* stand for?
2. When do CTL programs execute?
3. What directive would you use to print a message (not an error) during preprocessing (CPP)?
4. What directive would you use to print a message (not an error) during assembly?
5. What directive would you use to print an error message during CPP preprocessing?
6. What directive would you use to print an error message during assembly?
7. What directive would you use to create a CPP CTL variable?
8. What directive(s) could you use to create a Gas CTL variable?
9. What are the CPP conditional compilation directives?
10. What are the Gas conditional assembly directives?
11. What macro could you use to create preprocessor-time loops?
12. What directive(s) could you use to create assembly-time loops?
13. What directive would you use to extract the characters from a macro parameter object in a loop?
14. What directive(s) do you use to define a CPP macro?
15. What directive(s) do you use to define a Gas macro?
16. How do you invoke a macro in a Gas source file?
17. How do you specify macro parameters in a CPP macro declaration?
18. How do you specify macro parameters in a Gas macro declaration?
19. How do you specify that a Gas macro parameter is required?
20. How do you specify that a Gas macro parameter is optional?
21. How do you specify a variable number of CPP macro arguments?
22. How do you specify a variable number of Gas macro arguments?
23. Explain how you can manually test whether a Gas macro parameter is present (without using the `:req` suffix).
24. What Gas directive would you use (generally inside a conditional assembly sequence) to immediately terminate macro expansion without processing any additional statements in the macro?

14

STRING OPERATIONS



A *string* is a collection of values stored in contiguous memory locations. Strings are usually arrays of bytes, half words, words, dwords, or quad words. Some CISC CPUs, such as the Intel x86-64, support instructions that operate directly on strings of data. However, the ARM does not provide instructions for this purpose, as string operations tend to be complex and violate RISC design guidelines. Nevertheless, it is possible to operate on string data structures by using discrete ARM instructions.

Although the ARM CPU does not support string-specific instructions, string operations are still an important function that CPUs must perform. This chapter discusses how to work with strings when using ARM assembly language. First off, this chapter describes how to call functions in the C stdlib to implement string operations. These functions are well written

(typically in assembly language) and provide a high-performance implementation, as long as you are using zero-terminated strings. As noted throughout this book, however, zero-terminated strings are not the most suitable string data structure for high-performance string operations. Therefore, this chapter describes a better string format that allows you to write faster string functions.

Of course, the problem with a new string implementation is that the C stdlib functions don't support it, so this chapter also describes how to implement various string functions that support the new string format. Finally, this chapter concludes by discussing Unicode strings and string functions.

14.1 Zero-Terminated Strings and Functions

Chapter 4 briefly introduced the string data type, discussing zero-terminated strings (commonly used in C/C++ and languages derived from them), length-prefixed strings, string descriptors, and other string forms. As noted there, the zero-terminated string is the most common string form in use today. In particular, Linux (Pi OS) and macOS generally use zero-terminated strings for API functions to which you pass or receive string data. Therefore, you'll often use zero-terminated strings in your ARM assembly language programs running under these OSes. This section describes the issues with zero-terminated strings and how to call functions in the C stdlib that support zero-terminated strings.

The main problem with zero-terminated strings is performance. Such strings often require scanning over every character in the string to perform simple operations such as computing the string length. For example, the following code computes the length of a string named `longZString`:

```
          lea   x1, longZString
          mov   x2, x1          // Save pointer to string.
whileLp:  ldrb  w0, [x1], #1    // Fetch next char and inc X1.
          cmp   w0, #0          // See if 0 byte.
          bne   whileLp         // Repeat while not 0.
          sub   x0, x1, x2      // X0 = X1 - X2
          sub   x0, x0, #1      // Adjust for extra increment.

// String length is now in X0.
```

If `longZString` is indeed very long, this code sequence can take a long time to execute.

Length-prefixed strings (see section 4.6.2, "Length-Prefixed Strings," on page 188) solve this problem by including the string's current length along with the data. Any string function that uses the string's length will operate much more efficiently because it doesn't have to first scan the entire string to determine its length. If you get to choose the string format to use in your assembly language code, choosing a data type that includes the length as part of the string's data can dramatically improve string-processing performance.

Unfortunately, you don't always get to choose the string format to use in your code. Sometimes external datasets, applications, or OSes force the zero-terminated string format on you (for example, OS API calls generally require zero-terminated strings).

It is possible to improve the performance of naive string functions such as the string-length code given earlier. In particular, the code at `whileLp` processes a single byte per iteration of the loop. Because the ARM64 CPU is capable of processing 8 bytes at a time in general-purpose registers (and 16 bytes at a time in vector registers), you might wonder whether it's possible to do better than one character per loop iteration. After all, if you can process 16 bytes per loop iteration (rather than 1), the function should run 16 times faster, right?

The answer is a qualified yes. The first caveat is that processing 16 bytes per iteration is more complex and will require more than three instructions in the loop body. Expecting a 16-times improvement is therefore overly optimistic; 4 to 8 times faster is probably a more reasonable expectation but still worth achieving.

The second caveat is that processing 16 characters at a time requires loading 16 bytes from memory on each iteration, meaning that for many strings you'll have to read data from memory beyond the end of the string. Thus, it's possible to read beyond the end of an MMU page in memory containing the string, which could lead to a memory protection fault (see section 3.1.7, "Memory Access and MMU Pages," on page 127). While such a situation is rare, it nevertheless represents a defect in your code that could crash your application.

One last issue, while not as lethal as an illegal memory access, is that loading 16 bytes of data from memory into a vector register works best if the data is aligned on a 16-byte boundary. Unfortunately, a zero-terminated string is not guaranteed to begin on such a boundary in memory.

If you have complete control of string placement in memory, you can arrange for strings to always begin on a 16-byte boundary. Likewise, you can always include padding at the end of the strings so you're guaranteed to be able to read at least 15 bytes beyond the end of the string's data, thereby avoiding the MMU page-boundary problem. Unfortunately, few programs have such tight control over their strings that they can guarantee this arrangement in memory. For example, if an OS returns a pointer to a string, it may violate alignment and padding requirements.

As a general rule, I recommend calling C stdlib functions if you're going to manipulate zero-terminated (C) strings. In the past, C stdlib functions were written in C, and even with the best optimizing compilers it was easy enough for a good assembly language programmer to write faster code than the compiler produced. However, modern C stdlib string code is typically written in assembly language (by an *expert* programmer) and is often much better than any code you would write yourself. The GNU C stdlib for AARCH64, for example, has the following functions written in handcoded

assembly language (see section 14.6, "For More Information," on page 859 for more on these):

```
strcpy
strchr
strchrnul
strcmp
strcpy
strlen
strncmp
strnlen
strrchr
```

The following is the GNU C stdlib *strlen.S* source file (slightly modified for formatting and comments):

```
// strlen.S
//
// Copyright (C) 2012-2022 Free Software Foundation, Inc.
// This file is part of the GNU C Library. The GNU C
// Library is free software; you can redistribute it
// and/or modify it under the terms of the GNU Lesser
// General Public License as published by the Free
// Software Foundation; either version 2.1 of the License,
// or (at your option) any later version. The GNU C
// Library is distributed in the hope that it will be
// useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU Lesser General Public
// License for more details. You should have received a
// copy of the GNU Lesser General Public License along
// with the GNU C Library. If not, see
// <https://www.gnu.org/licenses/>.

#include <sysdep.h>

// Assumptions:
//
// ARMv8-a, AArch64, Advanced SIMD
// MTE compatible

❶ #define srcin       x0
  #define result      x0

  #define src         x1
  #define synd        x2
  #define tmp         x3
  #define shift       x4

  #define data        q0
  #define vdata       v0
  #define vhas_nul    v1
```

```
#define vend        v2
#define dend        d2

// Core algorithm: For each 16-byte chunk, calculate a
// 64-bit nibble mask value with 4 bits per byte. This code
// take 4 bits of every comparison byte with shift right
// and narrow by 4 instruction. Since the bits in the
// nibble mask reflect the order in which things occur in
// the original string, counting trailing 0s identifies
// exactly which byte matched.

// On input, X0 contains a pointer to a zero-terminated string.
// On return, X0 contains the string length.

STRLEN:
      ❷ bic     src, srcin, 15
        ld1     {vdata.16b}, [src]
        cmeq    vhas_nul.16b, vdata.16b, 0
        lsl     shift, srcin, 2
        shrn    vend.8b, vhas_nul.8h, 4     /* 128 -> 64 */
        fmov    synd, dend
        lsr     synd, synd, shift
        cbz     synd, zloop

      ❸ rbit    synd, synd
        clz     result, synd
        lsr     result, result, 2
        ret

        .p2align 5
zloop:
      ❹ ldr     data, [src, 16]!
        cmeq    vhas_nul.16b, vdata.16b, 0
        umaxp   vend.16b, vhas_nul.16b, vhas_nul.16b
        fmov    synd, dend
        cbz     synd, zloop

      ❺ shrn    vend.8b, vhas_nul.8h, 4     /* 128 -> 64 */
        sub     result, src, srcin
        fmov    synd, dend
        rbit    synd, synd
        clz     tmp, synd
        add     result, result, tmp, lsr 2
        ret
```

The defines ❶ give the registers meaningful names within the function. Personally, I prefer to see the register names with comments describing their contents rather than redefined registers, to make it easier to avoid reusing registers, but this is certainly an acceptable style, especially for C library code that uses the ARM ABI.

As noted earlier, code that operates on 16 bytes at a time (as this version of `strlen()` does) has to deal with two issues: aligning data on a 16-byte boundary and preventing data access beyond the end of the MMU page containing the string. This code achieves both of those requirements by

using the `bic` instruction ❷ to set the LO 4 bits of the string pointer to 0. Setting the LO 4 bits to 0 will align the pointer on the 16-byte boundary on or before the beginning of the string. Note that `src` (`X1`) might now point at up to 15 characters before the actual start of the string (which might contain some 0 bytes; this function will deal with that issue later). Because MMU pages always begin on a 4,096-byte boundary (which are also 16-byte boundaries), adjusting the pointer to the beginning of the 16-byte boundary will never produce a memory access outside the MMU page containing the start of the string.

Another advantage to aligning the pointer on a 16-byte boundary is that you don't have to worry about accidental illegal memory accesses at the end of the MMU page. Because 4,096 is divisible by 16, loading 16 bytes at a time on 16-byte boundaries will never produce a memory access that crosses a page boundary. If the zero-terminating byte is anywhere within those 16 bytes, reading data to the end of the 16-byte block is safe. Clearing the LO 4 bits of the pointer thus allows safe memory accesses within an MMU page.

The problem with aligning the `src` pointer to a 16-byte boundary is that doing so may change the pointer to point at memory *before* the start of the string. While this won't create MMU page fault problems, it might cause the code to incorrectly compute the string's length. At the very least, you don't want to count any extra bytes before the start of the string, nor do you want to terminate the string-length calculation, because 0s appear in those earlier bytes.

Fortunately, the code handles this in a clever way ❷. First, the `ld1` instruction starts the process by loading 16 bytes from the aligned address in `src`. The `cmeq` instruction then locates every 0 byte in those 16, then stores `0xFF` in the corresponding bytes of V1 (`vhas_null`) and 0s everywhere else.

The `shrn` instruction shifts the compare mask bits to the right 4 bit positions. The even bytes now contain two comparison masks in the LO and HO nibbles, and the "narrowing" component of the instruction extracts those even bytes and packs them into the LO 8 bytes of V2 (for a total of 16 nibbles), which the `fmov` instruction copies into X2.

The `lsl` instruction (which I skipped discussing in the previous paragraph) is part of the code that deals with extra bytes appearing in the 16-byte block prior to the string. This multiplies the original address by 4; this will be an index into the nibbles held in X2, where the string will actually start (using only the LO 6 bits of the shifted value). The `lsr` instruction shifts the nibble mask to the right by the number of bits held in the LO 6 bits of X4 (`shift`). This removes the `cmeq` nibble masks from X2 for the bytes that appear before the start of the string. These `lsl` and `lsr` instructions allow the algorithm to ignore the (possible) extra bytes in the 16-byte block before the string.

The resulting value in X2 will contain `0b1111` nibbles everywhere a 0 byte appears in the portion of the string held in V0 (`vdata`), and `0b0000` in all the other nibbles. In particular, X2 will contain 0 if there were no 0 bytes in the string in V0. If X2 contains 0 (meaning no 0 bytes), the string's terminating 0 byte must appear later in the string; in that case, the `cbz` instruction transfers control to label `zloop`.

If X2 does not contain 0, a `0b1111` nibble indicates the position of a 0 byte in the string. The `rbit` (reverse bits) instruction ❸ reverses all the bits in X2, and the `clz` instruction counts the number of leading 0s. Because each byte in the string is marked by 4 bits in X2, the count in X0 (result) is four times the length of the string. The `lsr` instruction shifts this count right by 2, which divides the bit count by 4, producing the string length.

The function then returns this length to the caller in X0. This code handles the case where the string appears in the first block of bytes loaded into V0 (`vdata`). If the string is sufficiently long that the code must fetch another block of 16 bytes from memory, the function transfers control to the code ❹. The code at `zloop` is responsible for processing blocks of 16 characters, where the first byte read from memory is part of the string (unlike the previous code, where the first 1–15 bytes might not be part of the string). This loop rapidly scans through blocks of 16 bytes, looking for the first one that contains a 0. As this loop takes five instructions to process 16 bytes (versus three instructions to process a single byte in the original string-length example), you would expect it to run approximately eight times faster than the single-byte-at-a-time code.

Once the loop finds a 0 byte somewhere in a block of 16 bytes ❺, it determines the position of the 0 byte (using the same technique as earlier ❷ ❸), adds the distance from the start of the string to the current 16-byte block, then adds the number of nonzero bytes in the current 16-byte block.

Although this code is tricky and complex, computing the length of a zero-terminated string is a common operation, so it's worth the optimization work. You would be hard-pressed to improve on this algorithm when writing your own code.

Again, because the *glibc* (GNU Library for C) authors have spent considerable time optimizing their ARM string functions, I highly recommend calling the C stdlib functions if they are appropriate for your application.

14.2 A String Format for Assembly Language Programmers

As mentioned many times in this book, the zero-terminated string is not the best string data type to use if you want to write the highest-performing code. Choosing a string format that incorporates the length, and possibly other information, as well as providing data alignment for the string data, can improve performance in many cases. This section introduces a sample format that provides these improvements.

The string format discussed in this section is based on the HLA string format (see section 4.6.2, "Length-Prefixed Strings," on page 188). HLA strings consist of two 32-bit length values followed by the character data and a zero-terminating byte. A string "variable" is just a pointer object that points to the first character of the string (or to a zero-terminating byte, if the string is empty).

The current length of the string (character count, not including the zero-terminating byte) appears at address `ptr-4`, and a maximum allocated

space (for characters) value appears at address `ptr-8` (where `ptr` is the pointer to the character data). Both length values are unsigned 32-bit values, supporting strings up to 4GB in length. For HLA, string objects are always aligned on a 4-byte boundary and the storage allocated for the string (and 0 byte) is always a multiple of 4 bytes.

For strings on a 64-bit CPU, a few changes are appropriate. First, the 4-byte maximum length and current length fields can remain. You probably don't need strings that can hold more than four billion characters. The alignment should be on a 16-byte boundary, allowing the use of Neon vector registers to efficiently process 16 bytes at a time from memory. Finally, storage allocated for strings should always be a multiple of 16 bytes (to prevent problems when reading bytes beyond the end of the string). Here's a first pass at a structure that defines this string type:

```
struct  string, -8
word    string.maxlen
word    string.len
byte    string.chars  // Note: up to 4GB chars
ends    string
```

With this struct declaration and a pointer to a string object in X0, you can access the fields of the string as follows:

```
ldr  w1, [x0, #string.maxlen]  // Fetch the maxlen field.
ldr  w2, [x0, #string.len]     // Fetch the current length.
```

Note that X0 points directly at the first character of the string (if it isn't empty), so you can reference the character data by using `[x0]` directly (you don't need to use the `string.chars` field name, which turns out to be 0, anyway).

If you would actually like to allocate storage for string data, or initialize some string storage with character data, the following two macros are useful (as a first approximation, with a minor modification to appear in the next section):

```
            .macro  str.buf strName, maxSize
            .align  4   // Align on 16-byte boundary.
            .word   \maxSize
            .word   0
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len

            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
```

```
        .byte   0   // Zero-terminating byte

        // Ensure object is multiple of 16 bytes:

        .align  4
        .endm
```

The `str.buf` macro will allocate storage for a string that can hold a maximum of `maxSize` characters (plus a zero-terminating byte). The `.align` directive ensures that the object begins on a 16-byte boundary (2^4). It initializes the first word of the structure (`string.maxlen`) with `maxSize` passed as the argument. It creates an empty string by initializing the second 4 bytes (the `string.len` field) with 0. Finally, it allocates sufficient storage for `maxSize + 1` characters (for the string data and a zero-terminating byte, initializing them to zeros) and additional storage to ensure that the whole data structure (including the `string.maxlen` and `string.len` fields) consumes a multiple of 16 bytes.

Here's an example use of the `str.buf` macro:

```
str.buf  myString, 100  // Maximum of 100 chars in string
```

The `str.literal` macro also creates string buffers, but instead of initializing it with the empty string, you can specify a string literal in the macro:

```
str.literal hwStr, "Hello, World!\n"
```

Note that `str.literal` will initialize both the `string.maxlen` and `string.len` fields with the actual size of the string literal you supply.

This string data type has one small issue. Although the entire structure is aligned on a 16-byte boundary—and the whole structure is a multiple of 16 bytes long, at least when you create the buffers with the `str.buf` and `str.literal` macros—the first character of string data is actually at an address that is not a multiple of 16 (though it is a multiple of 8). To process string data 16 bytes at a time, you must either make a special case of the first 8 bytes or add another 8 bytes to the beginning of the structure (some additional fields or just 8 padding bytes). In the next section, you'll see a modification of these two macros that adds an extra field for this purpose.

14.2.1 Dynamic String Allocation

As long as you use only the `str.buf` and `str.literal` macros to allocate storage for string variables, you don't have to worry about alignment and MMU page issues; your string data structures will always be allocated on a 16-byte boundary (because of the `.align 4` statement) and will always be a multiple of 16 bytes long. However, if you want to dynamically allocate storage for your strings (using the C stdlib `malloc()` function, for example), you must deal with data alignment and padding issues yourself.

The C stdlib `malloc()` function makes no promises about the storage it allocates, other than it will return a pointer to *at least* the amount of storage you've requested if the function succeeds. In particular, the C stdlib doesn't

make any guarantees about the alignment of the storage you've requested. Also, `malloc()` may allocate a few bytes more than you've requested, but you can never count on this. If you want your storage to be allocated on a certain byte boundary (such as a 16-byte boundary), you'll have to handle this yourself.

MALLOC() AND MEMORY ALIGNMENT

Although the C stdlib doesn't address data alignment of the storage you've requested, I've run some experiments on macOS and Linux (64-bit), and it appears that both OSes always allocate storage on 16-byte boundaries. I tested this by calling `malloc()` 256 times, allocating 1, 2, 3, . . . , 256 bytes of storage. I logically OR'd all the resulting pointers together. The result had 0s in the LO 4 bits (meaning all return pointers were 16-byte aligned). It's therefore probably justified to assume that all calls to `malloc()` on these machines return a 16-byte-aligned block memory.

That being said, there is no guarantee that this will be true for every version of `malloc()` you might call. If you want to live dangerously, you can assume 16-byte alignment on the returned blocks, but I'd recommend you always check the return value to verify this.

If you can't guarantee that `malloc()` returns a properly aligned block, you can create a `str.alloc` function that does this for you:

1. On entry, add 16 to the requested storage size to make room for any needed padding bytes.
2. Call the `malloc()` function with the new allocation size.
3. Save the pointer that `malloc()` returns (you will need it to free the storage later).
4. Add 15 to the pointer and clear the LO 4 bits of the sum; then add 16 so the pointer contains the address of the first character position in the string.
5. Set the `maxlen` field as appropriate.
6. Initialize the `len` field to 0.
7. Store a zero-terminating byte at the first character position (to create an empty string).
8. Return the pointer to the first character position as the `str.malloc` result.

A function that deallocates the string storage is much simpler: all you need to do is fetch the allocated pointer (saved during the `str.alloc` call), then call the C stdlib `free()` function to free the storage. From where do you retrieve the allocated pointer value? The best place to keep it is within the

string object data structure itself, as accomplished in the following modification of the `string` structure:

```
struct  string, -16
dword   string.allocPtr // At offset -16
word    string.maxlen   // At offset -8
word    string.len      // At offset -4
byte    string.chars    // At offset 0

// Note: characters in string occupy offsets
// 0 ... in this structure.

ends    string
```

The header fields now consume 16 bytes, so the `string.chars` field will start on a 16-byte aligned boundary (assuming the whole structure is on a 16-byte boundary).

Before providing the code to implement `str.alloc` and `str.free`, I'll introduce one other useful string constructor function that `str.alloc` will use: `str.bufInit`. Its purpose is similar to the `str.buf` macro insofar as it initializes a memory buffer to hold a string object, but while you use `str.buf` to declare a static object in memory during assembly, `str.bufInit` allows you to initialize a block of memory at runtime. The `str.bufInit` function does the following:

- Adjusts the pointer passed to it so that the address held in the pointer is 16-byte aligned (adding 0 to 15 to the pointer's value if it is not already 16-byte aligned).
- Initializes the `string.allocPtr` field to 0 (`NULL`) to differentiate the buffer from one created by `str.alloc`.
- Computes the `string.maxlen` field value based on the buffer size passed to the function, subtracting any padding bytes needed to achieve 16-byte alignment, as well as the 16 bytes required by the header field and any additional bytes needed to ensure that the whole structure is a multiple of 16 bytes long.
- Initializes the `string.len` field to 0 and stores a zero-terminating byte at the beginning of the character buffer area.

Before presenting the implementation of these string functions, a quick sidetrack is necessary to present the `volatile_save` structure the code will use to preserve registers. This structure appears in the *aoaa.inc* header file and takes the following form:

```
// Structure to hold the volatile registers saved by functions
// that call C stdlib funcs
//
// Note: size of this structure must be a multiple of 16 bytes!

            struct  volatile_save
            qword   volatile_save.xOx1
```

```
            qword   volatile_save.x2x3
            qword   volatile_save.x4x5
            qword   volatile_save.x6x7
            qword   volatile_save.x8x9
            qword   volatile_save.x10x11
            qword   volatile_save.x12x13
            qword   volatile_save.x14x15
            qword   volatile_save.v0
            qword   volatile_save.v1
            qword   volatile_save.v2
            qword   volatile_save.v3
            qword   volatile_save.v4
            qword   volatile_save.v5
            qword   volatile_save.v6
            qword   volatile_save.v7
            qword   volatile_save.v8
            qword   volatile_save.v9
            qword   volatile_save.v10
            qword   volatile_save.v11
            qword   volatile_save.v12
            qword   volatile_save.v13
            qword   volatile_save.v14
            qword   volatile_save.v15
            ends    volatile_save
```

Listing 14-1 contains the `str.alloc` and `str.free` functions, as well as updates to the `str.buf` and `str.literal` macros (to handle the `string.allocPtr` field). The listing uses the `string` structure (I gave earlier in this section) that includes the `string.allocPtr` field. For strings whose storage is allocated dynamically, this field will contain the allocation pointer that `str.free` will use when deallocating the string's storage. For string objects that were not created on the heap, this field will contain `NULL` (0). This structure is the first major piece of code appearing in the listing.

```
// Listing14-1.S
//
// String initialization, allocation, and deallocation functions and macros

            #include    "aoaa.inc"

// Assembly language string data structure:

            struct  string, -16
            dword   string.allocPtr // At offset -16
            word    string.maxlen   // At offset -8
            word    string.len      // At offset -4
            byte    string.chars    // At offset 0

            // Note: characters in string occupy offsets
            // 0 ... in this structure.

            ends    string
```

The str.buf and str.literal macros contain minor modifications (to the macros with the same name given earlier in this chapter) that include storage for the allocPtr field:

```
// Listing14-1.S (cont.)
//
// str.buf
//
// Allocate storage for an empty string
// with the specified maximum size:

            .macro  str.buf strName, maxSize
            .align  4         // Align on 16-byte boundary.
            .dword  0         // NULL ptr for allocation ptr
            .word   \maxSize  // Maximum string size
            .word   0         // Current string length
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

// str.literal
//
// Allocate storage for a string buffer and initialize
// it with a string literal:

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len

            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
            .byte   0   // Zero-terminating byte

            // Ensure object is multiple of 16 bytes:

            .align  4
            .endm
```

Note that both of these macros will initialize this field to NULL (0).

The next section of the listing is the code section, beginning with the usual getTitle function:

```
// Listing14-1.S (cont.)

            .code
            .global malloc
            .global free

ttlStr:     wastr  "Listing14-1"
```

```
// Standard getTitle function
// Returns pointer to program name in X0

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle
```

Next, the `str.bufInit` function initializes a memory buffer for use as a string variable:

```
// Listing14-1.S (cont.)
//
// str.bufInit
//
// Initializes a raw memory buffer for use as an assembly
// language string object
//
// On entry:
//
//  X0- Pointer to the first byte of a buffer
//  W1- Buffer length
//
// On return:
//
//  X0- Pointer to string variable object
//  X1- Maximum string length
//
//  Carry flag clear if success, set if error

            proc    str.bufInit

            locals  str_bufInit_l
            dword   str_bufInit_l.saveX2
            byte    str_bufInit_l.stkSpace, 64
            endl    str_bufInit_l

            enter   str_bufInit_l.size
            str     x2, [fp, #str_bufInit_l.saveX2]

            // Clear HO 32 bits of X1:

          ❶ and     x1, x1, #0xFFFFFFFF

            // Ensure that the pointer is aligned
            // on a 16-byte boundary:

          ❷ add     x2, x0, #15
            bic     x2, x2, #0xf

            // Point X2 at the start of the
            // character data:
```

```
            ❸ add     x2, x2, #string.chars-string.allocPtr

              // Compute the new maxlen value:

            ❹ sub     x0, x2, x0
              subs    x1, x1, x0
              bmi     str.bufInit.bad

              // Force maxlen to be a multiple of 16:

            ❺ bic     x1, x1, #0xf

              // Error if maxlen is 0:

              cbz     x1, str.bufInit.bad

              // Initialize the string struct fields:

            ❻ str     xzr, [x2, #string.allocPtr] // NULL
              str     w1,  [x2, #string.maxlen]
              str     wzr, [x2, #string.len]      // Empty str
              strb    wzr, [x2, #string.chars]    // 0 byte

              mov     x0, x2      // Return str ptr in X0.

              ldr     x2, [fp, #str_bufInit_l.saveX2]
              adds    xzr, xzr, xzr     // Clear the carry flag.
              leave

// Error return (sets the carry flag):

str.bufInit.bad:
              ldr     x2, [fp, #str_bufInit_l.saveX2]
              cmp     x2, #0  // Set the carry flag.
              leave
              endp    str.bufInit
```

This function expects a pointer to the buffer in X0 along with the buffer length in W1. It initializes the fields of the string object and returns a pointer to the string object in X0. The code begins by clearing the HO 32 bits of X1, so the code can work with 64-bit values ❶. It then adjusts the pointer passed in X0 to make it 16-byte aligned, by adding 16 and clearing the LO 4 bits of the sum ❷. This adjusts X0 to point at the next higher 16-byte aligned address if it wasn't already so aligned.

Next, the code adjusts the pointer to contain the address of the first byte of character data in the string (so that the other fields have negative offsets from the pointer) ❸. It then computes the new `maxlen` value by subtracting out the padding bytes (for 16-byte alignment) and the size of the fields preceding the character data ❹. The function returns an error if this difference is a negative value.

The code ensures that the length of the character data is a multiple of 16 bytes (possibly further reducing the `maxlen` size) by clearing the LO 4 bits

of the length value ❺. The function returns an error if the maxlen value turns out to be 0.

Finally, the code initializes the fields of the string object (producing an empty string) ❻. Note that

```
adds    xzr, xzr, xzr
```

clears the carry flag (successful return) because adding 0 to anything never produces an unsigned overflow (carry). Also note that

```
cmp     x2, #0
```

always sets the carry flag because the carry will be set after a comparison if the left value is greater than or equal to (higher than or the same as) the right value. Of course, for unsigned values, any value will always be greater than or equal to 0.

Two functions in Listing 14-1, str.alloc and str.free, will call the C stdlib malloc() and free() functions. The str.alloc and str.free functions preserve all registers they modify that don't contain explicit return values. However, because the malloc() and free() functions follow the ARM ABI, they are allowed to overwrite values in the volatile register set. To preserve the register values, the str.alloc and str.free functions must preserve the volatile registers by using the volatile_save structure.

Up next is the str.alloc function:

```
// Listing14-1.S (cont.)
//
// str.alloc
//
// Allocates storage for an assembly language string
// object on the heap (C stdlib malloc heap)
//
// On entry:
//
//  W0- Maximum string length for string object
//
// On exit:
//
//  X0- Pointer to string object (NULL if error)
//
//  Carry clear if successful, set if error

            proc    str.alloc

            locals  str_alloc
            dword   str_alloc.maxlen    // Really only a word
            dword   str_alloc.saveX1
            salign  4   // 16-byte align vsave
         ❶  byte    str_alloc.vsave, volatile_save.size
            byte    str_alloc.stkSpace, 64
            endl    str_alloc
```

```
    enter   str_alloc.size

    // Preserve X1 and point it at the
    // volatile_save.xOx1 entry in str_alloc.vsave:

    str     x1, [fp, #str_alloc.saveX1]

    // Load X1 with the effective address of
    // str_alloc.vsave (which will be the
    // volatile_save.xOx1 element):

❷   add     x1, fp, #str_alloc.vsave

    // Preserve all the volatile registers (call to
    // malloc may change these). Note that X1 is
    // currently pointing at volatile_save.xOx1 in
    // str_alloc.vsave (volatile_save). You don't know
    // that you *have* to save all the registers (it's
    // unlikely malloc will modify them all), but just
    // to be safe ...

    // The following code stores away X2, ..., X15 and
    // VO..V15 in successive memory locations in the
    // volatile_save structure. X1 was already preserved,
    // and it returns the result in XO.

❸   stp     x2,  x3,  [x1, #16]!
    stp     x4,  x5,  [x1, #16]!
    stp     x6,  x7,  [x1, #16]!
    stp     x8,  x9,  [x1, #16]!
    stp     x10, x11, [x1, #16]!
    stp     x12, x13, [x1, #16]!
    stp     x14, x15, [x1, #16]!

    str     q0,  [x1, #16]!
    str     q1,  [x1, #16]!
    str     q2,  [x1, #16]!
    str     q3,  [x1, #16]!
    str     q4,  [x1, #16]!
    str     q5,  [x1, #16]!
    str     q6,  [x1, #16]!
    str     q7,  [x1, #16]!
    str     q8,  [x1, #16]!
    str     q9,  [x1, #16]!
    str     q10, [x1, #16]!
    str     q11, [x1, #16]!
    str     q12, [x1, #16]!
    str     q13, [x1, #16]!
    str     q14, [x1, #16]!
    str     q15, [x1, #16]!

    // Save maxlen value for now:

    str     w0, [fp, #str_alloc.maxlen]
```

```
                // Force maxlen to be a multiple of 16 and
                // add in 16 extra bytes so you can ensure
                // that the storage is 16-byte aligned.
                // Also add in the size of the string.struct
                // fields:

             ❹  add     x0, x0, #31 + (string.chars-string.allocPtr)
                and     x0, x0, #0xffffffff // Fix at 32 bits.
                bic     x0, x0, #0xf        // Force to multiple of 16.

                // Call C stdlib malloc function to allocate the
                // storage:

             ❺  bl      malloc
                cmp     x0, x0              // Set carry flag on error.
                cbz     x0, str.alloc.bad   // Error if NULL return.

                mov     x1, x0              // Save allocation pointer.

                // Adjust pointer to point at start of characters
                // in string struct and 16-byte align the pointer:

             ❻  add     x0, x0, #15+(string.chars-string.allocPtr)
                bic     x0, x0, #0xf

                // Initialize the string struct fields:

                str     x1,  [x0, #string.allocPtr] // Save alloc ptr.
                ldr     w2,  [fp, #str_alloc.maxlen]
                str     w2,  [x0, #string.maxlen]   // Save maxlen.
                str     wzr, [x0, #string.len]      // Empty string.
                strb    wzr, [x0, #string.chars]    // Zero terminator

                // Restore all the volatile general-
                // purpose registers:

                adds    xzr, xzr, xzr   // Clear carry for success.

str.alloc.bad:

                // Restore all the volatile registers.
                // From this point forward, the code must
                // not change the carry flag.

             ❼  add     x1, fp, #str_alloc.vsave
                ldp     x2,  x3,  [x1, #16]!
                ldp     x4,  x5,  [x1, #16]!
                ldp     x6,  x7,  [x1, #16]!
                ldp     x8,  x9,  [x1, #16]!
                ldp     x10, x11, [x1, #16]!
                ldp     x12, x13, [x1, #16]!
                ldp     x14, x15, [x1, #16]!
```

```
        ldr     q0,  [x1, #16]!
        ldr     q1,  [x1, #16]!
        ldr     q2,  [x1, #16]!
        ldr     q3,  [x1, #16]!
        ldr     q4,  [x1, #16]!
        ldr     q5,  [x1, #16]!
        ldr     q6,  [x1, #16]!
        ldr     q7,  [x1, #16]!
        ldr     q8,  [x1, #16]!
        ldr     q9,  [x1, #16]!
        ldr     q10, [x1, #16]!
        ldr     q11, [x1, #16]!
        ldr     q12, [x1, #16]!
        ldr     q13, [x1, #16]!
        ldr     q14, [x1, #16]!
        ldr     q15, [x1, #16]!

        ldr     x1, [fp, #str_alloc.saveX1]

        leave
        endp    str.alloc
```

The local variable declaration `str_alloc.vsave` (type `volatile_save` ❶) will hold the preserved values of the volatile registers. Unfortunately, this structure is so large that you cannot directly access fields by using the `[FP, #offset]` addressing mode. Therefore, the code computes the address of the `volatile_save.x0x1` field into X1 and stores successive registers into the block pointed at by X1 ❷. This code must initialize X1 prior to storing anything in `str_alloc.vsave`, so it first preserves X1 in a different local variable. Because the function returns the result in X0 and has to save X1 in a different location, this code doesn't actually use the `volatile_save.x0x1` field of `str_alloc.vsave`.

The code saves all the volatile registers except X0 and X1 ❸. It uses the pre-increment addressing mode, so it skips over the `volatile_save.x0x1` field when writing the X2 and X3 registers to the structure.

Next, the code computes a string allocation size by adding 16 to `maxlen` (to cover the extra fields in the string data structure) ❹; it also adjusts the allocation size to be a multiple of 16 (greater than or equal to the requested size plus 16). This ensures that the character data area is a multiple of 16 bytes long, so string-handling code can manipulate 16 bytes at a time without worrying about accessing data beyond the allocated storage.

The call to `malloc()` ❺ allocates the storage for the string object. This code checks for a `NULL` (0) return result and returns an error if `malloc()` fails. On success, the code initializes the fields of the string object and then returns a pointer to the object in X0 (with the carry clear on a successful call) ❻. Finally, the code restores all the volatile registers (except X0, which contains the function result) ❼.

Next, the code includes the `str.free` function:

```
// Listing14-1.S (cont.)
//
// str.free
//
// Deallocates storage for an assembly language string
// object that was previously allocated via str.alloc
//
// On entry:
//
//  WO- Pointer to string object to deallocate

            proc    str.free

            locals  str_free
            dword   str_free.maxlen // Really a word
            dword   str_free.saveX1
            salign  4   // 16-byte align vsave
            byte    str_free.vsave, volatile_save.size
            byte    str_free.stkSpace,64
            endl    str_free

            enter   str_free.size

            // Preserve X1:

            str     x1, [fp, #str_free.saveX1]

            // Load X1 with the effective address of
            // str_alloc.vsave (which will be the
            // volatile_save.xOx1 element):

            add     x1, fp, #str_free.vsave

            // Preserve all the volatile registers (call to free
            // may change these):

          ❶ stp     x2,  x3,  [x1, #16]!
            stp     x4,  x5,  [x1, #16]!
            stp     x6,  x7,  [x1, #16]!
            stp     x8,  x9,  [x1, #16]!
            stp     x10, x11, [x1, #16]!
            stp     x12, x13, [x1, #16]!
            stp     x14, x15, [x1, #16]!

            str     q0,  [x1, #16]!
            str     q1,  [x1, #16]!
            str     q2,  [x1, #16]!
            str     q3,  [x1, #16]!
            str     q4,  [x1, #16]!
            str     q5,  [x1, #16]!
            str     q6,  [x1, #16]!
            str     q7,  [x1, #16]!
```

```
            str     q8,  [x1, #16]!
            str     q9,  [x1, #16]!
            str     q10, [x1, #16]!
            str     q11, [x1, #16]!
            str     q12, [x1, #16]!
            str     q13, [x1, #16]!
            str     q14, [x1, #16]!
            str     q15, [x1, #16]!

            // Fetch the allocation pointer from the
            // string struct data type:

❷           ldr     x1, [x0, #string.allocPtr]

            // Make sure it's not NULL (non-allocated
            // pointer):

❸           cbz     x1, str.free.done

            // Defensive code, set the allocPtr field to
            // NULL:

            str     xzr, [x0, #string.allocPtr]

            // Deallocate the storage:

❹           mov     x0, x1
            bl      free

str.free.done:

            // Restore the volatile register before
            // returning:

            add     x1, fp, #str_free.vsave
❺           ldp     x2,  x3,  [x1, #16]!
            ldp     x4,  x5,  [x1, #16]!
            ldp     x6,  x7,  [x1, #16]!
            ldp     x8,  x9,  [x1, #16]!
            ldp     x10, x11, [x1, #16]!
            ldp     x12, x13, [x1, #16]!
            ldp     x14, x15, [x1, #16]!

            ldr     q0,  [x1, #16]!
            ldr     q1,  [x1, #16]!
            ldr     q2,  [x1, #16]!
            ldr     q3,  [x1, #16]!
            ldr     q4,  [x1, #16]!
            ldr     q5,  [x1, #16]!
            ldr     q6,  [x1, #16]!
            ldr     q7,  [x1, #16]!
            ldr     q8,  [x1, #16]!
            ldr     q9,  [x1, #16]!
            ldr     q10, [x1, #16]!
```

```
            ldr     q11, [x1, #16]!
            ldr     q12, [x1, #16]!
            ldr     q13, [x1, #16]!
            ldr     q14, [x1, #16]!
            ldr     q15, [x1, #16]!

            ldr     x1, [fp, #str_free.saveX1]
            leave
            endp    str.free
```

The `str.free` function also calls a C stdlib function and therefore must preserve all the volatile registers. In fact, the preservation code ❶ ❺ makes up the bulk of the statements in this function.

The caller passes the address of an assembly string object in the X0 register to this function. However, this is not the address that the code passes to the C stdlib `free()` function; instead, this code fetches the address found in the `string.allocPtr` field to pass on to `free()` ❷.

Before actually calling `free()`, the code first checks whether this pointer value is `NULL` ❸. A `NULL string.allocPtr` value means that the string wasn't originally allocated with a call to `str.alloc`. If that's the case, `str.free` simply returns (without registering an error), allowing code to call this function on dynamically and statically allocated objects. This is sometimes convenient when an arbitrary string pointer has been passed to a function that frees the storage without knowing how the storage was originally created.

Finally, the `str.free` function calls the `free()` function ❹ to return the storage to the heap.

Here's a main program example (along with some data) that tests the functions appearing in Listing 14-1:

```
// Listing14-1.S (cont.)
//
// Some read-only strings:

fmtStr:     wastr   "hwStr=%s"
fmtStr2:    wastr   "hwDynamic=%s"
fmtStr3:    wastr   "strBufInit error\n"

            str.literal hwLiteral, "Hello, world!\n"

////////////////////////////////////////////////////////
//
// Main program to test the code:

            proc    asmMain, public

            locals  lcl
            qword   hwStr
            qword   hwDynamic
            byte    hwBuffer, 256
            byte    stkSpace, 64
            endl    lcl
```

```
enter   lcl.size      // Reserve space for locals.

// Demonstrate call to str.bufInit:

// Initialize hwBuffer as a string object and
// save pointer in hwStr:

add     x0, fp, #hwBuffer
mov     x1, #256    // Buffer size
bl      str.bufInit
str     x0, [fp, #hwStr]

// Force copy of hwLiteral into hwStr:

lea     x2, hwLiteral
ldr     w3, [x2, #string.len]   // Get length.
str     w3, [x0, #string.len]   // Save hwStr len.

// Cheesy string copy. You know the length is less
// than 16 bytes and both string objects have a
// minimum of 16 character locations available.

ldr     q0, [x2]    // Copy "Hello, world!\n" string.
str     q0, [x0]

// Now, hwStr contains a copy of hwLiteral.
// Print hwStr (because the assembly language
// string format always includes a zero-terminating
// byte, you can just call printf to print the string).
// Note that X0 still contains the hwStr pointer.

mov     x1, x0
lea     x0, fmtStr
mstr    x1, [sp]
bl      printf

// Demonstrate call to str.alloc and str.free:

mov     x0, #256    // String size
bl      str.alloc
bcs     badAlloc
str     x0, [fp, #hwDynamic]

// Force copy of hwLiteral into hwDynamic:

lea     x2, hwLiteral
ldr     w3, [x2, #string.len]   // Get length.
str     w3, [x0, #string.len]   // Save hwDynamic len.

// Cheesy string copy. You know the length is less
// than 16 bytes and both string objects have a
// minimum of 16 character locations available.
```

```
            ldr     q0, [x2]    // Copy "Hello, world!\n" string.
            str     q0, [x0]

            // Now hwDynamic contains a copy of hwLiteral.
            // Print hwDynamic (because the assembly language
            // string format always includes a zero-terminating
            // byte, you can just call printf to print the string).
            // Note that X0 still contains the hwDynamic pointer.

            mov     x1, x0
            lea     x0, fmtStr2
            mstr    x1, [sp]
            bl      printf

            // Free the string storage:

            ldr     x0, [fp, #hwDynamic]
            bl      str.free

AllDone:    leave

badAlloc:   lea     x0, fmtStr3
            bl      printf
            leave
            endp    asmMain
```

The `asmMain` function provides a few simple examples of calls to the `str.alloc`, `str.free`, and `str.bufInit` functions.

Here's the `build` command for Listing 14-1 and the sample program output:

```
% ./build Listing14-1
% ./Listing14-1
Calling Listing14-1:
hwStr=Hello, world!
hwDynamic=Hello, world!
Listing14-1 terminated
```

As you can see, this code properly copied the static string to the dynamically allocated string.

14.2.2 String Copy Function

Listing 14-1 demonstrates the lack of perhaps the most important string function of all: one that copies character data from one string to another. This section presents `str.cpy`, the second-most-used string function (after string length, in my experience), which makes a copy of the data in one string variable and stores that data in a second string variable.

The `str.cpy` function must do the following:

- Compare the length of the source string against the maximum length of the destination string and return an error if the source string will not fit in the destination string variable.
- Copy the `len` field from the source string to the destination string.
- Copy `len + 1` characters from the source string to the destination string, which will also copy the zero-terminating byte.

Listing 14-2 provides the implementation of this function.

```
// Listing14-2.S
//
// A str.cpy string copy function

            #include    "aoaa.inc"

// Assembly language string data structure:

            struct  string, -16
            dword   string.allocPtr // At offset -16
            word    string.maxlen   // At offset -8
            word    string.len      // At offset -4
            byte    string.chars    // At offset 0

            // Note: characters in string occupy offsets
            // 0 ... in this structure

            ends    string

// str.buf
//
// Allocate storage for an empty string
// with the specified maximum size:

            .macro  str.buf strName, maxSize
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   \maxSize
            .word   0
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

// str.literal:
//
// Allocate storage for a string buffer and initialize
// it with a string literal:

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len
```

```
            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
            .byte   0   // Zero-terminating byte

            // Ensure object is multiple of 16 bytes:

            .align  4
            .endm

//////////////////////////////////////////////////////////

            .data
            str.buf     destination, 256
            str.literal source, "String to copy"

//////////////////////////////////////////////////////////

            .code
            .global malloc
            .global free

ttlStr:     wastr  "Listing14-2"

// Standard getTitle function
// Returns pointer to program name in X0

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

//////////////////////////////////////////////////////////
//
// str.cpy
//
// Copies the data from one string variable to another.
//
// On entry:
//
//  X0- Pointer to source string (string struct variable)
//  X1- Pointer to destination string
//
// On exit:
//
//  Carry flag clear if no errors; carry is set if
//  the source string will not fit in the destination.

            proc    str.cpy

            locals  str_cpy
            qword   str_cpy.saveV0
```

```
            qword   str_cpy.saveX2X3
            dword   str_cpy.saveX4
            byte    str_cpy.stkSpace,64 // Not actually needed
            endl    str_cpy

            enter   str_cpy.size

            // Preserve X2 ... X4 and V0:

            str     q0,     [fp, #str_cpy.saveV0]
            stp     x2, x3, [fp, #str_cpy.saveX2X3]
            str     x4,     [fp, #str_cpy.saveX4]

            // Ensure the source will fit in the destination
            // string object:

          ❶ ldr     w4, [x0, #string.len]
            ldr     w3, [x1, #string.maxlen]
            cmp     w4, w3
            bhi     str.cpy.done    // Note: carry is set.

            // Set the length of the destination string
            // to the length of the source string:

          ❷ str     w4, [x1, #string.len]

            // X4 contains the number of characters to copy.
            // While this is greater than 16, copy 16 bytes
            // at a time from source to dest:

          ❸ mov     x2, x0  // Preserve X0 and X1.
            mov     x3, x1
cpy16:      ldr     q0, [x2], #16
            str     q0, [x3], #16
            subs    w4, w4, #16
            bhi     cpy16

// At this point, you have fewer than 16 bytes to copy. If
// W4 is not 0, just copy 16 remaining bytes (you know,
// because of the string data structure, that if you have at
// least 1 byte left to copy, you can safely copy
// 16 bytes):

          ❹ beq     setZByte    // Skip if 0 bytes.

            ldr     q0, [x2]
            str     q0, [x3]

// Need to add a zero-terminating byte to the end of
// the string. Note that maxlen does not include the
// 0 byte, so it's always safe to append the 0
// byte to the end of the string.
```

```
setZByte:   ldr     w4,  [x0, #string.len]
          ❺ strb    wzr, [x1, w4, uxtw]

            adds    wzr, wzr, wzr   // Clears the carry

str.cpy.done:
            ldr     q0,     [fp, #str_cpy.saveV0]
            ldp     x2, x3, [fp, #str_cpy.saveX2X3]
            ldr     x4,     [fp, #str_cpy.saveX4]
            leave
            endp    str.cpy

//////////////////////////////////////////////////////////
//
// A read-only format string:

fmtStr:     wastr   "source='%s', destination='%s'\n"

//////////////////////////////////////////////////////////
//
// Main program to test the code:

            proc    asmMain, public

            locals  lcl
            byte    stkSpace, 64
            endl    lcl

            enter   lcl.size      // Reserve space for locals.

            lea     x0, source
            lea     x1, destination
            bl      str.cpy

            mov     x2, x1
            mov     x1, x0
            lea     x0, fmtStr
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

AllDone:    leave
            endp    asmMain
```

The `str.cpy` function is straightforward and efficient, almost entirely because of the design of the string data type (in particular, the alignment and padding requirements of the string). The code first checks to ensure that the current length of the source string is less than or equal to the maximum length allowed for the destination string ❶. If the source string's length is too large, control transfers to the end of the function and it returns. The comparison will set the carry flag if the source length is "higher or same" as the destination maximum length. Therefore, this comparison automatically sets the carry flag to indicate a *string overflow* error

if it branches because the `string.len` field is higher than the `string.maxlen` field. Because the new destination string will be a copy of the source string, the code then sets the destination `string.len` field to the source string's length ❷.

The code is responsible for copying the character data from the source string to the destination string ❸. This is a `repeat...until` loop, so it always copies 16 bytes, even if the string length is 0. That's okay because the string data type always ensures that the character storage area is a multiple of 16 bytes long (including space for a zero-terminating byte). This loop may end up copying just the zero-terminating byte and 15 bytes of garbage data, but it will not access memory beyond the end of the string object's storage.

For each of the 16 bytes the loop copies ❸, the code decrements the length counter (W4) by 16. The `subs` instruction sets the flags exactly as a `cmp` instruction would, so the `bhi` instruction repeats the loop as long as the value in W4 is greater than 16 (prior to the `subs` instruction). If the string's length is a multiple of 16 bytes long, this loop will terminate after copying the last 16 bytes of the string (when W4 decrements to 0). In this situation, the `beq` instruction ❹ transfers control to the code that will append the zero-terminating byte.

If the string's length is not an integral multiple of 16, subtracting 16 will produce a result greater than 0 but less than 16 (meaning some characters remain left to copy from the source to the destination). Therefore, the code will fall through to the `ldr/str` instructions and copy the remaining bytes of the string (plus some garbage bytes).

Finally, the code will store a zero-terminating byte to the end of the string ❺ in the event the previous `ldr/str` instructions didn't copy that byte along with the character data.

NOTE *Technically, the `beq` instruction in Listing 14-2 is unnecessary. If the string's length is an exact multiple of 16 bytes long, at least 1 additional byte must be copied: the zero-terminating byte. Therefore, the data structure is guaranteed to contain at least 16 additional bytes, so falling through to the next pair of load and store instructions won't create a problem. As an interesting experiment, you might determine whether removing the `beq` instruction improves or hurts the algorithm's performance.*

Here's the `build` command and sample program output for the code in Listing 14-2:

```
% ./build Listing14-2
% ./Listing14-2
Calling Listing14-2:
source='String to copy', destination='String to copy'
Listing14-2 terminated
```

Although this string is shorter than 16 characters long and doesn't fully test `str.cpy`, I've run this program with different source strings to verify that it works for larger strings.

14.2.3 String Comparison Function

After copying strings, comparing strings is the string function you'll likely use most often. To compare two character strings, use the following steps:

1. Extract a character at corresponding indices from both strings.
2. Compare the two characters. If they are not equal, the comparison is complete and the result of the string comparison is the result of this character comparison (not equal, less than, or greater than). If they are equal and not zero, repeat step 1.
3. If the two characters are both 0 bytes, the comparison is finished, and the two strings are equal.

This algorithm works for zero-terminated strings (and, because they are also zero-terminated, for the assembly language string format given in this chapter). Note that the comparison algorithm does not use the string-length value.

Here's a naive version of this string comparison in ARM64 assembly language that assumes X0 and X1 point at the string data to compare:

```
cmpLp:
    ldrb w2, [x0], #1
    ldrb w3, [x1], #1
    cmp  w2, w3
    bne  strNE
    cbnz w2, cmpLp

// At this point, the strings are equal.
    .
    .
    .
strNE:
    // At this point, the strings are not equal.
```

As you saw with the `strlen()` function, processing multiple bytes at once using 64- or 128-bit registers is usually much faster. Can you improve performance by using vector registers? The big problem with this is that vector comparisons check for a specific comparison (`lt`, `le`, `eq`, `ne`, `gt`, or `ge`). They don't set the condition code flags, so you can use the conditional branches, which is what most programmers would prefer. That being the case, comparing eight characters at a time using 64-bit general-purpose registers is probably the best solution.

Given the efficiency of the glibc `strlen()` function, you might wonder whether its `strcmp()` function is also good. Listing 14-3 presents this function, with its operation explained in the comments.

```
// Listing14-3.S
//
// GNU glibc strcmp function
//
```

```
// Copyright (C) 2013 ARM Ltd.
// Copyright (C) 2013 Linaro.
//
// This code is based on glibc cortex strings work originally
// authored by Linaro and relicensed under GPLv2 for the
// Linux kernel. The original code can be found @
//
// http://bazaar.launchpad.net/~linaro-toolchain-dev/
// cortex-strings/trunk/
//
// files/head:/src/aarch64/
//
// This program is free software; you can redistribute it
// and/or modify it under the terms of the GNU General Public
// License version 2 as published by the Free Software
// Foundation.
//
// This program is distributed in the hope that it will be
// useful, but WITHOUT ANY WARRANTY; without even the implied
// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
// PURPOSE. See the GNU General Public License for more
// details.
//
// You should have received a copy of the GNU General Public
// License along with this program. If not, see
// <http://www.gnu.org/licenses/>.

#include <linux/linkage.h>
#include <asm/assembler.h>

// Compare two strings
//
// Parameters:
//  X0 - Const string 1 pointer
//  X1 - Const string 2 pointer
//
// Returns:
//  X0 - An integer less than, equal to, or greater
//       than zero if S1 is found, respectively, to be
//       less than, to match, or to be greater than S2

#define REP8_01 0x0101010101010101
#define REP8_7f 0x7f7f7f7f7f7f7f7f
#define REP8_80 0x8080808080808080

// Parameters and result

src1        .req    x0
src2        .req    x1
result      .req    x0
```

```
// Internal variables

data1       .req    x2
data1w      .req    w2
data2       .req    x3
data2w      .req    w3
has_nul     .req    x4
diff        .req    x5
syndrome    .req    x6
tmp1        .req    x7
tmp2        .req    x8
tmp3        .req    x9
zeroones    .req    x10
pos         .req    x11

strcmp:
    eor tmp1, src1, src2
    mov zeroones, #REP8_01
    tst tmp1, #7
    b.ne    .Lmisaligned8
    ands    tmp1, src1, #7
    b.ne    .Lmutual_align

// NUL detection works on the principle that (X - 1) &
// (~X) & 0x80 (=> (X - 1) & ~(X | 0x7f)) is nonzero if
// a byte is 0, and can be done in parallel across the
// entire word.

.Lloop_aligned:
    ldr data1, [src1], #8
    ldr data2, [src2], #8
.Lstart_realigned:
    sub tmp1, data1, zeroones
    orr tmp2, data1, #REP8_7f
    eor diff, data1, data2  // Nonzero if differences found
    bic has_nul, tmp1, tmp2 // Nonzero if NUL terminator
    orr syndrome, diff, has_nul
    cbz syndrome, .Lloop_aligned
    b   .Lcal_cmpresult
.Lmutual_align:

// Sources are mutually aligned but are not currently at
// an alignment boundary. Round down the addresses and
// then mask off the bytes that precede the start point:

    bic src1, src1, #7
    bic src2, src2, #7
    lsl tmp1, tmp1, #3  // Bytes beyond alignment -> bits
    ldr data1, [src1], #8
    neg tmp1, tmp1      // (Bits to align) - 64
    ldr data2, [src2], #8
    mov tmp2, #~0

    lsr tmp2, tmp2, tmp1 // Shift (tmp1 & 63)
    orr data1, data1, tmp2
    orr data2, data2, tmp2
```

```
    b   .Lstart_realigned
.Lmisaligned8:

// Get the align offset length to compare per byte first.
// After this process, one string's address will be
// aligned.

    and     tmp1, src1, #7
    neg     tmp1, tmp1
    add     tmp1, tmp1, #8
    and     tmp2, src2, #7
    neg     tmp2, tmp2
    add     tmp2, tmp2, #8
    subs    tmp3, tmp1, tmp2
    csel    pos, tmp1, tmp2, hi // Choose the maximum.
.Ltinycmp:
    ldrb    data1w, [src1], #1
    ldrb    data2w, [src2], #1
    subs    pos, pos, #1
    ccmp    data1w, #1, #0, ne      // NZCV = 0b0000
    ccmp    data1w, data2w, #0, cs  // NZCV = 0b0000
    b.eq    .Ltinycmp
    cbnz    pos, 1f  // Find the null or unequal ...
    cmp     data1w, #1
    ccmp    data1w, data2w, #0, cs
    b.eq    .Lstart_align  // The last bytes are equal.
1:
    sub result, data1, data2
    ret
.Lstart_align:
    ands    xzr, src1, #7
    b.eq    .Lrecal_offset

    // Process more leading bytes to make str1 aligned:

    add src1, src1, tmp3
    add src2, src2, tmp3

    // Load 8 bytes from aligned str1 and nonaligned str2:

    ldr data1, [src1], #8
    ldr data2, [src2], #8
    sub tmp1, data1, zeroones
    orr tmp2, data1, #REP8_7f
    bic has_nul, tmp1, tmp2
    eor diff, data1, data2 // Nonzero if differences found
    orr syndrome, diff, has_nul
    cbnz    syndrome, .Lcal_cmpresult

    // How far is the current str2 from the alignment boundary?

    and tmp3, tmp3, #7
.Lrecal_offset:
    neg pos, tmp3
```

```
.Lloopcmp_proc:

// Divide the 8 bytes into two parts. First, adjust the src
// to the previous alignment boundary, load 8 bytes from
// from the SRC2 alignment boundary, then compare with the
// relative bytes from SRC1. If all 8 bytes are equal,
// start the second part's comparison. Otherwise, finish
// the comparison. This special handle can guarantee all
// the accesses are in the thread/task space in order to
// avoid overrange access.

    ldr data1, [src1,pos]
    ldr data2, [src2,pos]
    sub tmp1, data1, zeroones
    orr tmp2, data1, #REP8_7f
    bic has_nul, tmp1, tmp2
    eor diff, data1, data2  // Nonzero if differences found
    orr syndrome, diff, has_nul
    cbnz    syndrome, .Lcal_cmpresult

    // The second part of the process:

    ldr data1, [src1], #8
    ldr data2, [src2], #8
    sub tmp1, data1, zeroones
    orr tmp2, data1, #REP8_7f
    bic has_nul, tmp1, tmp2
    eor diff, data1, data2  // Nonzero if differences found
    orr syndrome, diff, has_nul
    cbz syndrome, .Lloopcmp_proc
.Lcal_cmpresult:

// Reverse the byte order as big-endian, so CLZ can find
// the most significant 0 bits:

    rev syndrome, syndrome
    rev data1, data1
    rev data2, data2

    clz pos, syndrome

// The MS-nonzero bit of the syndrome marks either the
// first bit that is different or the top bit of the
// first 0 byte. Shifting left now will bring the
// critical information into the top bits.

    lsl data1, data1, pos
    lsl data2, data2, pos

// But you need to zero-extend (char is unsigned) the value
// and then perform a signed 32-bit subtraction:

    lsr data1, data1, #56
    sub result, data1, data2, lsr #56
    ret
```

The majority of the complexity is due to the code being written to handle string data that is not aligned to an 8-byte boundary. This string-comparison code could be written more simply if it could assume that the source and destination strings were always aligned on an 8-byte boundary. Because the assembly language string object is, by definition, always aligned on a 16-byte boundary, it is possible to write a more efficient comparison function for those strings. Listing 14-4 provides such a `str.cmp` function.

```
// Listing14-4.S
//
// A str.cmp string comparison function

            #include    "aoaa.inc"

// Assembly language string data structure:

            struct  string, -16
            dword   string.allocPtr // At offset -16
            word    string.maxlen   // At offset -8
            word    string.len      // At offset -4
            byte    string.chars    // At offset 0

            // Note: characters in string occupy offsets
            // 0 ... in this structure.

            ends    string

// str.buf
//
// Allocate storage for an empty string
// with the specified maximum size:

            .macro  str.buf strName, maxSize
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   \maxSize
            .word   0
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

// str.literal
//
// Allocate storage for a string buffer and initialize
// it with a string literal:

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len
```

```
            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
            .byte   0   // Zero-terminating byte

            // Ensure object is multiple of 16 bytes:

            .align  4
            .endm

//////////////////////////////////////////////////////////

            .data
            str.buf     destination, 256
            str.literal left,   "some string"
            str.literal right1, "some string"
            str.literal right2, "some string."
            str.literal right3, "some string"
            str.literal right4, ""
            str.literal right5, "t"
            str.literal right6, " "

            str.literal left2,  "some string 16.."
            str.literal right7, "some string 16.."
            str.literal right8, "some string 16."
            str.literal right9, "some string 16..."

//////////////////////////////////////////////////////////

            .code
            .global malloc
            .global free

ttlStr:     wastr   "Listing14-4"

// Standard getTitle function
// Returns pointer to program name in X0

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

//////////////////////////////////////////////////////////
//
// str.cmp
//
// Compares two string objects
//
// On entry:
//
//  X0- Pointer to left string
```

```
//  X1- Pointer to right string
//
//      left op right
//
//  Where op is the string comparison operation
//
// On exit:
//
//  Condition code flags contain state of comparison

            proc    str.cmp

            locals  str_cmp
            qword   str_cmp.saveX2X3
            dword   str_cmp.saveX4X5
            dword   str_cmp.saveX6X7
            byte    str_cmp.stkSpace,64
            endl    str_cmp

            enter   str_cmp.size

            // Preserve X2 ... X7:

          ❶ stp     x2, x3, [fp, #str_cmp.saveX2X3]
            stp     x4, x5, [fp, #str_cmp.saveX4X5]
            stp     x6, x7, [fp, #str_cmp.saveX6X7]

            mov     x2, x0  // Preserve X0 and X1.
            mov     x3, x1

            // Compute the minimum of the string lengths:

          ❷ ldr     w6, [x2, #string.len]
            ldr     w7, [x3, #string.len]
            cmp     w6, w7
            csel    w6, w6, w7, hs
            b.al    cmpLen

cmp8:
          ❸ ldr     x4, [x2], #8
            ldr     x5, [x3], #8
            rev     x4, x4
            rev     x5, x5
            cmp     x4, x5
            bne     str.cmp.done
cmpLen:
          ❹ subs    w6, w6, #8      // Also compares W6 to 8
            bhs     cmp8

            // Fewer than eight characters left (and more
            // than zero). Cheapest to just compare them
            // one at a time:

          ❺ adds    w6, w6, #8
            beq     str.cmp.done  // If lens are equal
```

```
cmp1:
          ❻ ldrb    w4, [x2], #1
            ldrb    w5, [x3], #1
            cmp     w4, w5
            bne     str.cmp.done
            subs    w6, w6, #1
            bne     cmp1

            // At this point, the strings are equal
            // through the length of the shorter
            // string. The comparison is thus based
            // on the result of comparing the lengths
            // of the two strings.

cmpLens:
          ❼ ldr     w6, [x0, #string.len]   // Fetch left len.
            cmp     w6, w7                  // Right len

str.cmp.done:
            ldp     x2, x3, [fp, #str_cmp.saveX2X3]
            ldp     x4, x5, [fp, #str_cmp.saveX4X5]
            ldp     x6, x7, [fp, #str_cmp.saveX6X7]
            leave
            endp    str.cmp

////////////////////////////////////////////////////////
//
// Some read-only strings:

ltFmtStr:   wastr   "Left ('%s') is less than right ('%s')\n"
gtFmtStr:   wastr   "Left ('%s') is greater than right ('%s')\n"
eqFmtStr:   wastr   "Left ('%s') is equal to right ('%s')\n"

////////////////////////////////////////////////////////
//
// prtResult
//
// Utility function to print the result of a string
// comparison

          ❽ proc    prtResult

            mov     x2, x1
            mov     x1, x0
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            beq     strsEQ
            bhi     strGT

            // Must be LT at this point

            lea     x0, ltFmtStr
            b       printf
```

```
strsEQ:     lea     x0, eqFmtStr
            b       printf

strGT:      lea     x0, gtFmtStr
            b       printf

            endp    prtResult

////////////////////////////////////////////////////////
//
// Main program to test the code:

            proc    asmMain, public

            locals  lcl
            byte    stkSpace, 64
            endl    lcl

            enter   lcl.size      // Reserve space for locals.

            lea     x0, left
            lea     x1, right1
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right2
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right3
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right4
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right5
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right6
            bl      str.cmp
            bl      prtResult

            lea     x0, left2
            lea     x1, right7
            bl      str.cmp
            bl      prtResult
```

```
            lea     x0, left2
            lea     x1, right8
            bl      str.cmp
            bl      prtResult

            lea     x0, left2
            lea     x1, right9
            bl      str.cmp
            bl      prtResult

AllDone:    leave
            endp    asmMain
```

The `str.cmp` function does not modify the X0 or X1 registers, but it will modify X2 through X7, so this code begins by preserving these register values ❶. It then copies the values in X0 and X1 into X2 and X3 (respectively), which it will use in the code.

When comparing the strings, `str.cmp` will compare only to the length of the shorter string. The code computes the minimum length of the two strings ❷, leaving the result in W6. If the two strings are equal to the length of the shorter string, the shorter string is considered less than the longer string.

The `cmp8` loop compares the characters in the string 8 bytes at a time ❸. Strings are intrinsically a *big-endian* data structure, meaning the lower-order bytes in the string have the most-significant values. Therefore, you cannot simply load 8 successive bytes into a pair of 64-bit registers and compare those registers; that would produce a little-endian comparison result. To resolve this issue, the code executes two `rev` instructions to swap the bytes in the two 64-bit registers prior to comparing them, resulting in a big-endian comparison.

After comparing the two dwords, the code branches to the return code if those dwords are not equal. At that point, the ARM condition codes will hold the result of the comparison. If the two dwords are equal, the `cmp8` loop must repeat until it has exhausted all the characters or finds a pair of dwords that are not equal. The code subtracts 8 from W6 and repeats if the value prior to the subtraction was greater than or equal to 8 (remember, `subs` and `cmp` set the flags the same way) ❹.

Because this code subtracts 8 from W6 before comparing the corresponding characters, if W6 winds up with 0, eight characters still remain to compare. That's why this code repeats even when subtracting 8 produces a 0 result.

If the code falls down to ❺, the W6 contains a negative result. The code adds 8 to this value to determine the number of characters it must still process. If the result is 0, the strings are the same length and all characters in the string are equal; in that case, the code exits (with the flags already containing appropriate values). If the result is nonzero, the code processes the remaining characters in the two strings one character at a time ❻. (Four instructions per character, for an average of four characters per string,

assuming random string lengths, is usually faster than attempting to zero out the excess bytes and compare 8 bytes at a time.)

If the code transfers or drops through to `cmpLens` ❼, the strings were equal to the length of the shorter string. At this point, the code determines the result of the comparison by comparing the strings' lengths.

The main program compares several strings to test the `str.cmp` function. The `prtResult` function ❽ is a short utility function that prints the result of the comparisons.

NOTE *This code would be slightly more efficient if it preserved X0 and X1 rather than X2 and X3, then used X0 and X1 rather than X2 and X3. However, I left X0 and X1 alone because during development I used `printf()` to print some debugging messages. Feel free to change this code to use X0/X1 rather than X2/X3 if the two extra instructions (those that move X0 and X1 into X2 and X3) bother you.*

As was the case for the glibc `strcmp()` function, the `str.cmp` function expects pointers to the two strings to compare in X0 and X1. The *left* string is X0, and the *right* string is X1. Left and right have to do with their position in a comparison expression. The following example demonstrates the positions of the two strings in an `if` statement:

```
if( leftStr <= rightStr ) then ...
```

The `strcmp()` function returns a result in X0 indicating the result of the comparison:

- If X0 is negative, the left string (X0) is less than the right string (X1).
- If X0 is 0, the two strings are equal.
- If X0 is positive, the left string is greater than the right string.

The `str.cmp` function, on the other hand, returns the comparison result in the condition code flags, so you can use the conditional branch instructions upon return to test the result.

Here's the `build` command and sample output for Listing 14-4:

```
% ./build Listing14-4
% ./Listing14-4
Calling Listing14-4:
Left ('some string') is equal to right ('some string')
Left ('some string') is less than right ('some string.')
Left ('some string') is greater than right ('some string')
Left ('some string') is greater than right ('')
Left ('some string') is less than right ('t')
Left ('some string') is greater than right (' ')
Left ('some string 16..') is equal to right ('some string 16..')
Left ('some string 16..') is greater than right ('some string 16.')
Left ('some string 16..') is less than right ('some string 16..')
Listing14-4 terminated
```

As you can see, `str.cmp` returned the proper results for the test strings.

14.2.4 Substring Function

The last ASCII example I'll provide in this chapter is the substring function `str.substr`. A typical substring function extracts a portion of the characters from a string, creating a new string from the substring. It typically has four parameters: a pointer to a source string, an index into the substring where the extraction is to begin, a length specifying the number of characters to copy from the source, and a pointer to a destination string.

The substring operation has several issues:

- You can't assume that the source characters are aligned on a 16-byte boundary.
- The specified starting index might be beyond the length of the source string.
- The specified substring length might extend beyond the end of the source string.
- The length of the substring might exceed the maximum length of the destination string.

The first issue is often impossible to deal with. Most of the time, either the source characters or the destination characters will have an unaligned address. The `str.substr` code in this section will choose to keep the destination address aligned on a 16-byte boundary (which it gets by default). The function must carefully check the length while copying data to ensure that it doesn't read any data beyond the end of the source-string data structure.

You can handle the second issue in two ways: either return an error code without copying any data or simply store an empty string into the destination. This latter solution is often the most convenient, and I rely on it in this section's code.

Likewise, there are two ways to handle the third issue: either return an error code without copying any data or copy all the characters from the starting index to the end of the source string into the destination string. Once again, the latter solution is often the most convenient, and the `str.substr` code relies on it.

The fourth issue is a bit more problematic. The `str.substr` code could truncate the string it copies, but this situation usually indicates a serious error on the part of the application (string overflow). Therefore, `str.substr` will return a flag in the carry to indicate success or failure.

NOTE *If you prefer to return an error status for the second and third issues listed here, you can easily modify `str.substr` to accomplish this.*

Listing 14-5 provides the `str.substr` function and a sample main program that tests it.

```
// Listing14-5.S
//
// A str.substr substring function
```

```
            #include    "aoaa.inc"

// Assembly language string data structure:

            struct  string, -16
            dword   string.allocPtr // At offset -16
            word    string.maxlen   // At offset -8
            word    string.len      // At offset -4
            byte    string.chars    // At offset 0

            // Note: characters in string occupy offsets
            // 0 ... in this structure

            ends    string

// str.buf
//
// Allocate storage for an empty string
// with the specified maximum size:

            .macro  str.buf strName, maxSize
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   \maxSize
            .word   0
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

// str.literal
//
// Allocate storage for a string buffer and initialize
// it with a string literal:

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len

            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
            .byte   0   // Zero-terminating byte

            // Ensure object is multiple of 16 bytes:

            .align  4
            .endm

////////////////////////////////////////////////////////////
```

```
            .data
fmtStr:     .ascii      "Source string:\n\n"
            .ascii      "          111111111122222222233333\n"
            .ascii      "012345678901234567890123456789012345\n"
            .asciz      "%s\n\n"

            str.buf     smallDest, 32
            str.literal dest,   "Initial destination string"

//                            111111111122222222233333
//                   012345678901234567890123456789012345
str.literal source, "Hello there, world! How's it going?"

////////////////////////////////////////////////////////

            .code

ttlStr:     wastr  "listing14-5"

// Standard getTitle function
// Returns pointer to program name in X0

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

////////////////////////////////////////////////////////
//
// str.substr
//
// Extracts a substring
//
// On entry:
//
//  X0- Pointer to source string
//  W1- Starting index into source string
//  W2- Length of substring
//  X3- Destination string
//
// On exit:
//
//  Carry clear on success and result stored at X3
//
//  If the substring will not fit in X3, return with
//  the carry set (and no data copied).

            proc    str.substr

            locals  str_substr
            qword   str_substr.saveV0
            qword   str_substr.saveX0X1
            qword   str_substr.saveX2X3
            qword   str_substr.saveX6X7
```

```
    byte    str_substr.stkSpace,64  // Not needed
    endl    str_substr

    enter   str_substr.size

    // Preserve X0 ... X7 and V0:

    str     q0,     [fp, #str_substr.saveV0]
    stp     x0, x1, [fp, #str_substr.saveX0X1]
    stp     x2, x3, [fp, #str_substr.saveX2X3]
    stp     x6, x7, [fp, #str_substr.saveX6X7]

    // Handle the exceptional conditions:
    //
    // 1. Index >= source.len   (return empty string)

❶   ldr     w6, [x0, #string.len]
    cmp     w1, w6
    bhs     returnEmpty

    // 2. Index + substr length > source length
    // If so, reduce the length to match the end
    // of the string:

❷   add     w7, w1, w2      // W7 = index + substr length
    cmp     w6, w7
    csel    w6, w6, w7, ls  // W6 = min(source len, sum)
    sub     w6, w6, w1      // W6 = actual length

    // 3. Substr length > destination maxlen
    //    (fail):

❸   ldr     w7, [x3, #string.maxlen]
    cmp     w6, w7          // Carry set if
    bhi     str.sub.exit    // W6 >= W7.

    // At this point, W6 contains the actual number of
    // characters to copy from the source
    // to the destination. This could be less than the
    // length passed in W2 if the index + substr length
    // exceeded the length of the source string.

❹   str     w6, [x3, #string.len]   // Save as dest len.

    // Point X0 at the first character of the substring
    // to copy to the destination string (base address
    // plus starting index):

❺   add     x0, x0, w1, uxtw
    b.al    test16
```

```
                // Copy the substring 16 bytes at a time:

copy16:
            ❻ ldr     q0, [x0], #16   // Get bytes to copy.
                str     q0, [x3], #16   // Store into dest.

                // Decrement the number of characters to copy by
                // 16. Quit if the result is negative (meaning
                // fewer than 16 characters were left to
                // copy). Remember, subs sets the flags the same
                // as cmp, so the following compares the value in
                // W6 against 16 and branches to copy16 if
                // 16 or more characters are left to copy:
test16:
                subs    w6, w6, #16
                bhs     copy16

                // W6 has gone negative. Need to add 16 to determine
                // the number of bytes left to copy:

            ❼ add     w6, w6, #16     // Now W6 contains 0 to 15.

                // Switch statement based on the number of characters
                // left to copy in the substring. Handle as a special
                // case each of the 0 ... 15 bytes to copy:

                and     x6, x6, #0xFFFFFFFF  // Zero-extend to 64 bits.
                adr     x7, JmpTbl
                ldr     w6, [x7, x6, lsl #2] // *4 for 32-bit entries
                add     x7, x7, w6, sxtw     // Sign-extend to 64 bits.
                br      x7

JmpTbl:     .word   str.sub.success-JmpTbl  // _0bytesToCopy
                .word   _1byteToCopy-JmpTbl
                .word   _2bytesToCopy-JmpTbl
                .word   _3bytesToCopy-JmpTbl
                .word   _4bytesToCopy-JmpTbl
                .word   _5bytesToCopy-JmpTbl
                .word   _6bytesToCopy-JmpTbl
                .word   _7bytesToCopy-JmpTbl
                .word   _8bytesToCopy-JmpTbl
                .word   _9bytesToCopy-JmpTbl
                .word   _10bytesToCopy-JmpTbl
                .word   _11bytesToCopy-JmpTbl
                .word   _12bytesToCopy-JmpTbl
                .word   _13bytesToCopy-JmpTbl
                .word   _14bytesToCopy-JmpTbl
                .word   _15bytesToCopy-JmpTbl

// Special case copying 1-15 bytes:

            ❽ _14bytesToCopy:
                ldr     x7, [x0], #8
                str     x7, [x3], #8
```

```
_6bytesToCopy:
            ldr     w7, [x0], #4
            str     w7, [x3], #4

_2bytesToCopy:
            ldrh    w7, [x0], #2
            strh    w7, [x3], #2
            b.al    str.sub.success

_13bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8

_5bytesToCopy:
            ldr     w7, [x0], #4
            str     w7, [x3], #4
            ldrb    w7, [x0], #1
            strb    w7, [x3], #1
            b.al    str.sub.success

_12bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8

_4bytesToCopy:
            ldr     w7, [x0], #4
            str     w7, [x3], #4
            b.al    str.sub.success

_11bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8
            ldrh    w7, [x0], #2
            strh    w7, [x3], #2
            ldrb    w7, [x0], #1
            strb    w7, [x3], #1
            b.al    str.sub.success

_10bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8
            ldrh    w7, [x0], #2
            strh    w7, [x3], #2
            b.al    str.sub.success

_9bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8
            ldrb    w7, [x0], #1
            strb    w7, [x3], #1
            b.al    str.sub.success

_8bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8
            b.al    str.sub.success
```

```
_15bytesToCopy:
            ldr     x7, [x0], #8
            str     x7, [x3], #8

_7bytesToCopy:
            ldr     w7, [x0], #4
            str     w7, [x3], #4

_3bytesToCopy:
            ldrh    w7, [x0], #2
            strh    w7, [x3], #2

_1byteToCopy:
            ldrb    w7, [x0], #1
            strb    w7, [x3], #1

// Branch here after copying all string data.
// Need to add a zero-terminating byte to the
// end of the destination string:

str.sub.success:
          ❾ strb    wzr, [x3]       // Zero-terminating byte
            adds    wzr, wzr, wzr   // Clear carry for success.

str.sub.exit:
            ldr     q0,     [fp, #str_substr.saveV0]
            ldp     x0, x1, [fp, #str_substr.saveX0X1]
            ldp     x2, x3, [fp, #str_substr.saveX2X3]
            ldp     x6, x7, [fp, #str_substr.saveX6X7]
            leave

// Special case where the code just returns an empty string:

returnEmpty:
          ❿ strh    wzr, [x3, #string.len]
            b.al    str.sub.success

            endp    str.substr

////////////////////////////////////////////////////////
//
// testSubstr
//
//  Utility function to test call to str.substr
//
// On entry:
//  X0, X1, X2, X3 -- str.substr parameters

successStr: wastr   "substr( '%s', %2d, %3d )= '%s'\n"
failureStr: wastr   "substr( '%s', %2d, %3d ) failed\n"

            proc    testSubstr

            locals  testSS
            byte    testSS.stkspace, 64
            endl    testSS
```

```
            enter   testSS.size

            lea     x5, successStr
            bl      str.substr
            bcc     success
            lea     x5, failureStr

success:
            mov     x4, x3
            mov     x3, x2
            mov     x2, x1
            mov     x1, x0
            mov     x0, x5
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            mstr    x3, [sp, #16]
            mstr    x4, [sp, #24]
            bl      printf
            leave
            endp    testSubstr

////////////////////////////////////////////////////////
//
// Main program to test the code:

            proc    asmMain, public

            locals  lcl
            byte    stkSpace, 64
            endl    lcl

            enter   lcl.size      // Reserve space for locals.

            lea     x0, fmtStr
            lea     x1, source
            mstr    x1, [sp]
            bl      printf

            lea     x0, source
            mov     x1, #0
            mov     x2, #11
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #20
            mov     x2, #15
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #20
            mov     x2, #20
            lea     x3, dest
            bl      testSubstr
```

```
            lea     x0, source
            mov     x1, #40
            mov     x2, #20
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #0
            mov     x2, #100
            lea     x3, smallDest
            bl      testSubstr

AllDone:    leave
            endp    asmMain
```

The `str.substr` function begins by finding any special cases it must handle. It first checks whether the starting index value is beyond the end of the source string; if so, the function returns the empty string as the result ❶. Next, the code checks whether the starting index plus the substring length would extend beyond the end of the source string; if so, it adjusts the length to reach to the end of the source string (and no farther) ❷. Finally, if the substring length is greater than the maximum length of the destination string, `str.substr` immediately returns with the carry flag set to indicate an error condition ❸.

If none of the special cases exist, the code can successfully copy a substring into the destination string. `str.substr` begins this process by setting the length of the destination string to the length of the substring ❹. The code then copies the substring data starting at the index position, beginning by adding the value of the index to the string pointer ❺.

The loop copies 16 bytes at a time using the V0 vector register (Q0) ❻, as long as there are 16 or more bytes left to copy. When fewer than 16 bytes are left to copy, the code drops down to ❼ and adds 16 to the remaining length value (because the loop subtracted 16 one too many times).

After adding 16, W6 will contain a value in the range 0 to 15, the number of bytes left to copy. The code could have executed a simple loop to copy the remaining bytes one at a time to the destination, but that would be somewhat slow. Instead, I chose to execute a (simulated) switch statement to transfer control to one of 16 labels ❽ where the code exists to carry out a straight-line, brute-force copy of the necessary bytes. (To reduce code size, I've interleaved these sections as much as possible, sharing various code sequences.)

Once they copy the necessary number of bytes, all these code sequences converge ❾ (also the location where the switch code transfers if 0 bytes are left to copy). This code appends a zero-terminating byte to the end of the string, clears the carry flag, and returns to the caller.

The code handles the special case in which `str.substr` returns an empty string because the index value was greater than the length of the source string ❿. This code sets the length of the destination string to 0, then transfers to ❾ to zero-terminate the string and return success. The `asmMain` function calls a special helper function (`testSubstr`) to perform various tests and print the results.

It is very unlikely that the start of the substring will lie on a 16-byte boundary. Therefore, when the function in Listing 14-5 fetches 16 bytes at a time from the source string, it will probably be an unaligned memory access (which is slower). Without writing a lot of code, you can't do much about this other than accept that execution will be slightly slower. Because the accesses may not be aligned on a 16-byte boundary, it is important that this code copies only the specified number of bytes (never reading beyond the end of the source string) to ensure it doesn't access an inappropriate memory page.

Here is the `build` command and sample output for the program in Listing 14-5:

```
% ./build Listing14-5
% ./Listing14-5
Calling Listing14-5:
Source string:

          111111111122222222233333
012345678901234567890123456789012345
Hello there, world! How's it going?

substr( 'Hello there, world! How's it going?',  0,  11 )= 'Hello there'
substr( 'Hello there, world! How's it going?', 20,  15 )= 'How's it going?'
substr( 'Hello there, world! How's it going?', 20,  20 )= 'How's it going?'
substr( 'Hello there, world! How's it going?', 40,  20 )= ''
substr( 'Hello there, world! How's it going?',  0, 100 ) failed
Listing14-5 terminated
```

While this isn't an exhaustive test by any means, this output is sufficient to show the basic operation of `str.substr`.

14.2.5 More String Functions

Of course, any decent string library has many additional string functions. A `str.len` function is the most obvious function missing thus far. The implementation of this function should be fairly obvious: just fetch the `string.len` field from the string data structure. Even ignoring this oversight, however, there are dozens of other string functions you might want to use (the HLA standard library, for example, provides over 200 string functions).

Unfortunately, this book doesn't have room to describe a complete set of string library functions. After reading this chapter, you should possess the skills needed to implement any string functions you need on your own. See section 14.6, "For More Information," on page 859 for further resources to help you do so.

14.3 The Unicode Character Set

All code examples up to this point in this book have assumed that strings in assembly language consist of sequences of ASCII characters, largely because

Gas doesn't directly support Unicode. However, Linux and macOS systems generally work with Unicode (though ASCII is a subset of Unicode). Now that you've seen how to implement string functions for ASCII characters, it's time to expand on Chapter 2's cursory introduction to Unicode and discuss string functions for Unicode strings.

14.3.1 Unicode History

A few decades back, engineers at Aldus Corporation, NeXT, Sun Microsystems, Apple Computer, IBM, Microsoft, the Research Libraries Group, and Xerox realized that their new computer systems with bitmaps and user-selectable fonts could display far more than 256 characters at one time. At the time, *double-byte character sets (DBCSs)* were the most common solution.

DBCSs had a couple of issues, however. First, as they were typically variable-length encodings, they required special library code; common character or string algorithms that depended on fixed-length character encodings would not work properly with them. Second, no consistent standard existed; different DBCSs used the same encoding for different characters.

To avoid these compatibility problems, the engineers sought a different solution. They came up with the Unicode character set, which originally used a 2-byte character size. Like DBCSs, this approach still required special library code (existing single-byte string functions would not always work with 2-byte characters). Other than changing the size of a character, however, most existing string algorithms would still work with 2-byte characters. The Unicode definition included all the (known or living) character sets at the time, giving each character a unique encoding, to avoid the consistency problems that plagued differing DBCSs.

The original Unicode standard used a 16-bit word to represent each character. Therefore, Unicode supported up to 65,536 character codes—a huge advance over the 256 possible codes representable with an 8-bit byte. Furthermore, Unicode is upward compatible from ASCII. If the HO 9 bits of a Unicode character's binary representation contain 0, the LO 7 bits use the standard ASCII code. (ASCII is a 7-bit code, so if the HO 9 bits of a 16-bit Unicode value are all 0, the remaining 7 bits are an ASCII encoding for a character.) If the HO 9 bits contain a nonzero value, the 16 bits form an extended character code, above and beyond the ASCII character set.

You may be wondering why so many character codes are necessary. When Unicode was first being developed, certain Asian character sets contained 4,096 characters. The Unicode character set even provided codes you could use to create an application-defined character set. Approximately half of the 65,536 possible character codes have been defined, and the remaining character encodings are reserved for future expansion.

Today, Unicode is a universal character set, long replacing ASCII and older DBCSs. All modern OSes (including macOS, Windows, Linux, Pi OS, Android, and Unix), all web browsers, and most modern applications provide Unicode support. The Unicode Consortium, a nonprofit corporation, maintains the Unicode standard. By maintaining the standard, the

consortium helps guarantee that a character you write on one system will display as you expect on a different system or application.

14.3.2 Code Points and Code Planes

Alas, as well-thought-out as the original Unicode standard was, its creators couldn't have anticipated the subsequent explosion in characters. Emojis, astrological symbols, arrows, pointers, and a wide variety of symbols introduced for the internet, mobile devices, and web browsers—along with a desire to support historic, obsolete, and rare scripts—have greatly expanded the Unicode symbol repertoire.

In 1996, systems engineers discovered that 65,536 symbols were insufficient. Rather than require 3 or 4 bytes for each Unicode character, those in charge of the Unicode definition gave up on trying to create a fixed-size representation of characters and allowed for opaque (and multiple) encodings of Unicode characters. Today, Unicode defines 1,112,064 code points, far exceeding the 2-byte capacity originally set aside for Unicode characters.

A Unicode *code point* is simply an integer value associated with a particular character symbol; you can think of it as the Unicode equivalent of the ASCII code for a character. The convention for Unicode code points is to specify the value in hexadecimal with a U+ prefix. For example, U+0041 is the Unicode code point for the letter A.

Blocks of 65,536 characters are known as a *multilingual plane* in Unicode. The first multilingual plane, U+000000 to U+00FFFF, roughly corresponds to the original 16-bit Unicode definition; the Unicode standard calls this the *Basic Multilingual Plane (BMP)*. Planes 1 (U+010000 to U+01FFFF), 2 (U+020000 to U+02FFFF), and 14 (U+0E0000 to U+0EFFFF) are supplementary planes. Plane 3 (U+030000 to U+03FFFF) is the *Tertiary Ideographic Plane* (see *https://unicode.org/roadmaps/tip/*). Unicode reserves planes 4 through 13 for future expansion and planes 15 and 16 for user-defined character sets.

The Unicode standard defines code points in the range U+000000 to U+10FFFF. Note that 0x10ffff is 1,114,111, which is where most of the 1,112,064 characters in the Unicode character set come from; the remaining 2,048 values form the *surrogate code points*.

14.3.3 Surrogate Code Points

As noted earlier, Unicode began life as a 16-bit (2-byte) character set encoding. When it became apparent that 16 bits were insufficient to handle all the possible characters that existed at the time, an expansion was necessary. As of Unicode v2.0, the Unicode Consortium extended the definition of Unicode to include multiword characters. Now Unicode uses surrogate code points (U+D800 through U+DFFF) to encode values larger than U+FFFF, as shown in Figure 14-1.

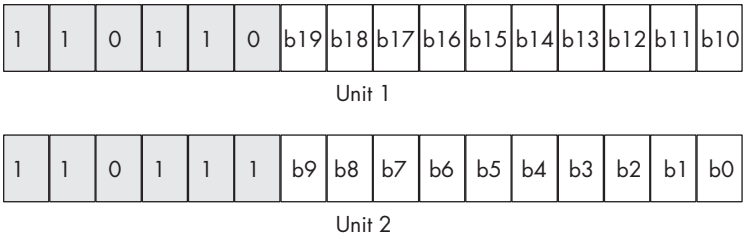


Figure 14-1: Surrogate code-point encoding for Unicode planes 1–16

The two words, unit 1 (high surrogate) and unit 2 (low surrogate), always appear together. The unit 1 value with HO bits 0b110110 specifies the upper 10 bits (b10 through b19) of the Unicode scalar, and the unit 2 value with HO bits 0b110111 specifies the lower 10 bits (b0 through b9) of the Unicode scalar. Therefore, the value of bits b16 to b19 plus 1 specifies Unicode planes 1 through 16. Bits b0 through b15 specify the Unicode scalar value within the plane.

Note that surrogate codes appear only in the BMP. None of the other multilingual planes contain surrogate codes. Bits b0 through b19 extracted from the unit 1 and 2 values always specify a Unicode scalar value (even if the values fall in the range U+D800 through U+DFFF).

14.3.4 Glyphs, Characters, and Grapheme Clusters

Each Unicode code point has a unique name. For example, U+0045 has the name LATIN CAPITAL LETTER A. The symbol A is *not* the name of the character. A is a *glyph*, a series of strokes (one horizontal and two slanted strokes) that a device draws in order to represent the character.

Many glyphs exist for the single Unicode character LATIN CAPITAL LETTER A. For example, a Times Roman A and a Times Roman Italic A have different glyphs, but Unicode doesn't differentiate between them (or between the A character in any two different fonts). The character LATIN CAPITAL LETTER A remains U+0045 regardless of the font or style you use to draw it.

The term *character* has a simple meaning when working with ASCII. The character code 0x41 corresponds to the LATIN CAPITAL LETTER A, which has a consistent representation when it appears on a display screen; in particular, a one-to-one correspondence exists between ASCII character codes and the symbol a user expects to see on the display. The situation is dramatically more complex when working with Unicode. There's a difference between what you'd normally call a character and the definition of a Unicode character (*scalar* in Unicode terminology). For example, consider the following Swift code:

```
import Foundation
let eAccent  :String = "e\u{301}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

This section uses the Swift programming language for examples because it is one of the first programming languages that attempts to do Unicode right (though a huge performance hit results from it). Assembly, on the other hand, requires the programmer to handle everything manually and isn't the best tool for many Unicode examples. I explain how to translate this to assembly code in section 14.4, "Unicode in Assembly Language," on page 853.

The Swift syntax for specifying a Unicode scalar value within a string is `"\u{hex}"`, where *hex* is a hexadecimal value, as in `"\u{301}"`. In this example, `301` is the hexadecimal code for the *combining acute accent* character. The first `print()` statement prints the character, producing `é` on the output, as you expect. The second `print()` statement prints the number of characters Swift determines are present in the string—in this case, `1`. The third `print()` statement prints the number of elements (UTF-16 elements, discussed further later in this section) in the string. In this case, that number is `2`, because the string holds 2 words of UTF-16 data.

In this example, is é one character or two? Internally (assuming UTF-16 encoding), the computer sets aside 4 bytes of memory for this single character (two 16-bit Unicode scalar values). On the screen, however, the output takes only one character position and looks like a single character to the user. When this character appears within a text editor and the cursor is immediately to the right of the character, the user expects that pressing BACKSPACE will delete it. From the user's perspective, then, this is a single character (as Swift reports when you print the count attribute of the string).

In Unicode, however, a *character* is largely equivalent to a code point. In Unicode terminology, when you talk about symbols that an application displays to an end user, you refer to them not as characters but as *grapheme clusters*. These are sequences of one or more Unicode code points that combine to form a single language element (that is, something that appears as a single character to the user on the display, such as é).

14.3.5 Normal Forms and Canonical Equivalence

The Unicode character é actually existed on personal computers long before Unicode came along: it's part of the original IBM PC character set and of the Latin-1 character set (used, for example, on old DEC terminals). Unicode uses the Latin-1 character set for the code points in the range U+00A0 to U+00FF, and U+00E9 just happens to correspond to the é character. Therefore, you can modify the earlier program as follows:

```
import Foundation
let eAccent  :String = "\u{E9}"
print( eAccent )
print( "eAccent.count=\(eAccent.count)" )
print( "eAccent.utf16.count=\(eAccent.utf16.count)" )
```

The outputs from this program are as follows:

```
é
eAccent.count=1
eAccent.utf16.count=1
```

Ouch! You now have a couple of strings that all produce é but contain a different number of code points. Imagine how this complicates programming strings containing Unicode characters. For example, if you try to compare the following three strings (Swift syntax), what will the result be?

```
let eAccent1 :String = "\u{E9}"
let eAccent2 :String = "e\u{301}"
```

To the user, both strings look the same on the screen. However, they clearly contain different values. If you compare them to see whether they are equal, will the result be true or false?

Ultimately, that depends on which string libraries you're using. Most current string libraries would return false if you compared these strings for equality. Many languages' string libraries simply report that both strings are unequal.

The two Unicode/Swift strings `"\{E9}"` and `"e\{301}"` should produce the same output on the display. Therefore, they are canonically equivalent according to the Unicode standard. Some string libraries won't report any of these strings as being equivalent. Some, like the one accompanying Swift, will handle small canonical equivalences (such as `"\{E9}" == "e\{301}"`) but not arbitrary sequences that should be equivalent. (This is probably a good balance of correctness versus efficiency; it can be computationally expensive to handle all the weird cases that won't normally happen, such as `"e\{301}\{301}"`.)

Unicode defines *normal forms* for Unicode strings. One aspect of normal form is to replace canonically equivalent sequences with an equivalent sequence—for example, replace `"e\u{309}"` with `"\u{E9}"` or vice versa (the shorter form is usually preferable). Some Unicode sequences allow multiple combining characters. Often, the order of the combining characters is irrelevant to producing the desired grapheme cluster. However, comparing two such characters is easier if the combining characters are in a specified order. Normalizing Unicode strings may also produce results whose combining characters always appear in a fixed order, thereby improving the efficiency of string comparisons.

14.3.6 Encodings

As of Unicode 2.0, the standard supports a 21-bit character space capable of handling over a million characters (though most of the code points remain reserved for future use). Rather than use a fixed-size 3-byte (or worse, 4-byte) encoding to allow the larger character set, the Unicode Consortium allows different encodings: UTF-32, UTF-16, and UTF-8 (*UTF* stands for

Unicode Transformation Format). Each of these three encodings comes with advantages and disadvantages.

UTF-32 uses 32-bit integers to hold Unicode scalars. The advantage to this scheme is that a 32-bit integer can represent every Unicode scalar value in only 21 bits. Programs that require random access to characters in strings—without having to search for surrogate pairs—and other constant-time operations are usually possible when using UTF-32. The obvious drawback to UTF-32 is that each Unicode scalar value requires 4 bytes of storage—twice that of the original Unicode definition and four times that of ASCII characters.

It may seem that using two or four times as much storage (over ASCII and the original Unicode) is a small price to pay. After all, modern machines have several orders of magnitude more storage than they did when Unicode first appeared. However, that extra storage has a huge impact on performance, because those additional bytes quickly consume cache storage. Furthermore, modern string-processing libraries often operate on character strings 8 bytes at a time (on 64-bit machines). With ASCII characters, that means a given string function can process up to eight characters concurrently; with UTF-32, that same string function can operate on only two characters concurrently. As a result, the UTF-32 version will run four times slower than the ASCII version. Ultimately, even Unicode scalar values are insufficient to represent all Unicode characters (that is, many Unicode characters require a sequence of Unicode scalars), so using UTF-32 doesn't solve the problem.

As the name suggests, the second encoding format that Unicode supports, *UTF-16*, uses 16-bit (unsigned) integers to represent Unicode values. To handle scalar values greater than 0xFFFF, UTF-16 uses the surrogate-pair scheme to represent values in the range 0x010000 to 0x10FFFF. Because the vast majority of useful characters fit into 16 bits, most UTF-16 characters require only 2 bytes. For those rare cases where surrogates are necessary, UTF-16 requires two words (32 bits) to represent the character.

The last encoding, and unquestionably the most popular, is *UTF-8*. The UTF-8 encoding is forward-compatible from the ASCII character set. In particular, all ASCII characters have a single-byte representation (their original ASCII code, where the HO bit of the byte containing the character contains a 0 bit). If the UTF-8 HO bit is 1, UTF-8 requires 1 to 3 additional bytes to represent the Unicode code point.

Table 14-1 provides the UTF-8 encoding schema, where the *x* bits are the Unicode point bits.

Table 14-1: UTF-8 Encoding

Bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+00	U+7F	0*xxxxxxx*			
2	11	U+80	U+7FF	110*xxxxx*	10*xxxxxx*		
3	16	U+800	U+FFFF	1110*xxxx*	10*xxxxxx*	10*xxxxxx*	
4	21	U+10000	U+10FFFF	11110*xxx*	10*xxxxxx*	10*xxxxxx*	10*xxxxxx*

For multibyte sequences, byte 1 contains the HO bits, byte 2 contains the next HO bits (LO bits compared to byte 1), and so on. For example, the 2-byte sequence (0b11011111, 0b10000001) corresponds to the Unicode scalar 0b0000_0111_1100_0001 (U+07C1).

UTF-8 encoding is probably the most common encoding in use, since most web pages use it. Most C stdlib string functions will operate on UTF-8 text without modification (although some can produce malformed UTF-8 strings if the programmer isn't careful).

Different languages and OSes default to using different encodings. For example, macOS and Windows tend to use UTF-16 encoding, whereas most Unix systems use UTF-8. Some variants of Python use UTF-32 as their native character format. By and large, though, most programming languages use UTF-8 because they can continue to use older ASCII-based character-processing libraries to process UTF-8 characters.

14.3.7 Combining Characters

Although UTF-8 and UTF-16 encodings are much more compact than UTF-32, the CPU overhead and algorithmic complexities of dealing with multibyte (or multiword) character sets complicates their use, introducing bugs and performance issues. Despite the issues of wasting memory, especially in the cache, why not simply define characters as 32-bit entities and be done with it? This seems like it would simplify string-processing algorithms, improving performance and reducing the likelihood of defects in the code.

The problem with this theory is that you cannot represent all possible grapheme clusters with only 21 bits (or even 32 bits) of storage. Many grapheme clusters consist of several concatenated Unicode code points. Here's an example from Chris Eidhof and Ole Begemann's *Advanced Swift*, version 3.0 (CreateSpace, 2017):

```
let chars: [Character] = [
    "\u{1ECD}\u{300}",
    "\u{F2}\u{323}",
    "\u{6F}\u{323}\u{300}",
]
print( chars )
```

Each of these Unicode grapheme clusters produces the same output: ọ̀, (a character from the Yoruba character set). The character sequence (`U+1ECD, U+300`) is an ọ followed by a combining acute. The character sequence (`U+F2, U+323`) is an ò followed by a combining dot. The character sequence (`U+6F, U+323, U+300`) is an o followed by a combining dot, followed by a combining acute.

The Swift string comparisons treat all four strings as equal:

```
print("\u{1ECD} + \u{300} = \u{1ECD}\u{300}")
print("\u{F2} + \u{323} = \u{F2}\u{323}")
print("\u{6F} + \u{323} + \u{300} = \u{6F}\u{323}\u{300}")
print("\u{6F} + \u{300} + \u{323} = \u{6F}\u{300}\u{323}")
print( chars[0] == chars[1] ) // Outputs true.
```

```
Print( chars[0] == chars[2] ) // Outputs true.
print( chars[0] == chars[3] ) // Outputs true.
Print( chars[1] == chars[2] ) // Outputs true.
print( chars[1] == chars[3] ) // Outputs true.
Print( chars[2] == chars[3] ) // Outputs true.
```

No single Unicode scalar value will produce this character. You must combine at least two Unicode scalars (or as many as three) to produce this grapheme cluster on the output device. Even UTF-32 encoding would still require two (32-bit) scalars to produce this particular output.

Emojis present another challenge that can't be solved using UTF-32. Consider the Unicode scalar U+1F471. This prints an emoji of a person with blond hair. If you add a skin-color modifier, you obtain (`U+1F471, U+1F3FF`), which produces a person with a dark skin tone and blond hair. In both cases, a single character displays on the screen. The first example uses a single Unicode scalar value, but the second example requires two. There is no way to encode this with a single UTF-32 value.

The bottom line is that certain Unicode grapheme clusters require multiple scalars, no matter how many bits you assign to the scalar (it's possible to combine 30 or 40 scalars into a single grapheme cluster, for example). That means you're stuck dealing with multiword sequences to represent a single "character," regardless of how hard you try to avoid it. This is why UTF-32 has never really taken off: it doesn't solve the problem of random access into a string of Unicode characters. When normalizing and combining Unicode scalars, using UTF-8 or UTF-16 encodings is more efficient.

Again, most languages and OSes today support Unicode in one form or another (typically using UTF-8 or UTF-16 encoding). Despite the obvious problems with dealing with multibyte character sets, modern programs need to deal with Unicode strings rather than simple ASCII strings.

14.4 Unicode in Assembly Language

As noted in section 2.17, "Gas Support for the Unicode Character Set," on page 102), Gas doesn't provide especially good support for Unicode strings. If you have a text editor that allows you to enter Unicode text into a source file, you might be able to type non-ASCII UTF-8 characters into a string constant and have Gas accept them. In general, though, the safest way to insert non-ASCII Unicode characters into an assembly language source file is to use hexadecimal constants. This section describes how to output Unicode characters from a console application and provides a brief introduction to Unicode string functions.

14.4.1 Writing Console Applications with UTF-8 Characters

To be able to print strings containing UTF-8 characters, you must ensure that your OS is able to accept them. This is generally accomplished by using

the C stdlib `setlocale()` function. Unfortunately, the parameter list varies by locale, so I can't provide a universal example that works everywhere. For US English, I typically use the following function call:

```
setlocale(LC_ALL, "en_US.UTF-8");
```

The exact string for the second parameter will vary based on the country and language. You can search online for a description of the `setlocale()` function for more details about calling this function (or see section 14.6, "For More Information," on page 859). The following Linux/macOS command will list the available locale strings for your system:

```
locale -a
```

Here are some of the strings produced by this command under macOS (on a Mac mini M1):

```
C                   en_NZ.ISO8859-1     it_IT
POSIX               en_NZ.ISO8859-15    it_IT.ISO8859-1
af_ZA               en_NZ.US-ASCII      it_IT.ISO8859-15
af_ZA.ISO8859-1     en_NZ.UTF-8         it_IT.UTF-8
af_ZA.ISO8859-15    en_US               ja_JP

Many entries snipped ...

zh_TW.Big5
zh_TW.UTF-8
```

See "For More Information" for an explanation of the locale string format.

You could call `setlocale()` from your assembly language code, but I've found it more convenient to modify the *c.cpp* program that the *build* script uses. The following shows this modification:

```
// c-utf8.cpp
//
// (Rename to c.cpp to use with build script.)
//
// Generic C++ driver program to demonstrate returning function
// results from assembly language to C++. Also includes a
// "readLine" function that reads a string from the user and
// passes it on to the assembly language code.
//
// Need to include stdio.h so this program can call "printf"
// and stdio.h so this program can call strlen.

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
```

```
// extern "C" namespace prevents "name mangling" by the C++
// compiler.

extern "C"
{
    // asmMain is the assembly language code's "main program":

    void asmMain( void );

    // getTitle returns a pointer to a string of characters
    // from the assembly code that specifies the title of that
    // program (which makes this program generic and usable
    // with a large number of sample programs in "The Art of
    // ARM Assembly Language").

    char *getTitle( void );

    // C++ function that the assembly
    // language program can call:

    int readLine( char *dest, int maxLen );

};

// readLine reads a line of text from the user (from the
// console device) and stores that string into the destination
// buffer the first argument specifies. Strings are limited in
// length to the value specified by the second argument
// (minus 1).
//
// This function returns the number of characters actually
// read, or -1 if there was an error.
//
// Note that if the user enters too many characters (maxlen or
// more), this function returns only the first maxlen - 1
// characters. This is not considered an error.

int readLine( char *dest, int maxLen )
{
    // Note: fgets returns NULL if there was an error, else
    // it returns a pointer to the string data read (which
    // will be the value of the dest pointer).

    char *result = fgets( dest, maxLen, stdin );
    if( result != NULL )
    {
        // Wipe out the newline character at the
        // end of the string:

        int len = strlen( result );
        if( len > 0 )
        {
            dest[ len - 1 ] = 0;
        }
```

```
        return len;
    }
    return -1; // If there was an error
}

int main(void)
{
    // Get the assembly language program's title:

    char *title = getTitle();

    setlocale(LC_ALL, "en_US.UTF-8");
    asmMain();
    printf( "%s terminated\n", title );
}
```

Listing 14-6 presents a trivial program that demonstrates example text output containing UTF-8, to be compiled and linked with *c-utf8.cpp*. This example prints the UTF-8 sequence U+65 (lowercase e) followed by U+301 (combining acute accent character).

```
// Listing14-6.S
//
// Simple program to demonstrate UTF-8 output

            #include    "aoaa.inc"

            .data
fmtStr:     .ascii  "Unicode='"

            // e followed by U+301 (0xCC, 0x81 in UTF-8)

            .ascii  "e"
            .byte   0xCC, 0x81

            .asciz  "'\n"

            .code
ttlStr:     wastr  "Listing14-6.S"

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

            proc    asmMain, public

            locals  lcl
            qword   saveX20_X21
            byte    stkSpace, 64
            endl    lcl

            enter   lcl.size      // Reserve space for locals.
```

```
            lea     x0, fmtStr
            bl      printf

AllDone:    leave
            endp    asmMain
```

Note that the UTF-8 encoding for U+301 requires 2 bytes

$$1\ 1\ 0\ b_{11}\ b_{10}\ b_9\ b_8\ b_7, \quad 1\ 0\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

where B_{11} down to B_0 is 0x301 or 0b011_0000_0001. Therefore, the two UTF-8 bytes are 0b1100_1100 (0xCC) and 0b10000001 (0x81).

Here's the `build` command and sample output for Listing 14-6 (assuming that *c-utf8.cpp* has been renamed to *c.cpp*):

```
% ./build Listing14-6
% ./Listing14-6
Unicode='é'
Listing14-6.S terminated
```

As you can see, the character sequence `e`, `0xcc`, `0x89` produces the accented é character.

14.4.2 Using Unicode String Functions

As long as you stick with UTF-8 encoding, character string functions that operate on ASCII strings will *mostly* work with Unicode strings. You should be aware of a few issues, though:

- Unless you keep strings in a canonical form, some string comparisons may report that two strings are not equal when, in fact, they would appear equal to the reader.
- String comparison for *less than* and *greater than* will likely produce non-intuitive results because ASCII comparisons don't work well in the face of Unicode scalars whose values consume 2 or more bytes.
- A string-length calculation (when using zero-terminated or the assembly language string data type) will report the number of *bytes* in the string, not the number of characters (scalars or glyphs). Unless the string contains only ASCII characters, the length calculation will be wrong. The only reasonable way to count characters in a Unicode string is to process each glyph one at a time and count the glyphs.
- Functions that accept indices into a string generally require glyph indices, not byte indices. For example, the `str.substr` function given earlier could extract a substring containing a portion of a glyph at the beginning of the string, or chop a glyph in half at the end of the string, if the index and length parameters aren't carefully chosen. Functions that insert glyphs into a string or delete characters from a string will suffer from this same problem.

Because of these issues (and more), it's dangerous to use ASCII-based string functions on Unicode and UTF-8 strings. It goes without saying that ASCII-based functions will not work on UTF-16 or UTF-32 encodings of Unicode characters.

Section 14.6, "For More Information," on the next page provides links to several string libraries (mostly written on C/C++) that process Unicode strings. The International Components for Unicode (UCI) library is important to consider, since it's the library provided by the Unicode Consortium. At the time of writing, this library (ICU74.2) claims the following support:

- The latest version of the Unicode standard
- Character set conversions with support for over 220 code pages
- Locale data for more than 300 locales
- Language-sensitive text collation (sorting) and searching based on the Unicode Collation Algorithm (ISO 14651)
- Regular expression matching and Unicode sets
- Transformations for normalization, upper- and lowercase, script transliterations (50+ pairs)
- Resource bundles for storing and accessing localized information
- Date/number/message formatting and parsing of culture-specific input/output formats
- Calendar-specific date and time manipulation
- Text boundary analysis for finding character, word, and sentence boundaries

Although this isn't a complete set of string functions you'd expect in a typical programming language, it does provide all the basic operations needed to correctly implement a full set of functions. Also keep in mind that Unicode string functions aren't particularly fast. Unfortunately, because of the design of the Unicode character set (and multibyte character sets in general), you have to process each and every character in a string to accomplish mundane tasks. Only a few functions, like `str.cpy`, can work without scanning over every character in the string.

14.5 Moving On

This chapter covered string data structures (zero-terminated and special assembly language strings), calling C stdlib string functions from assembly, writing assembly language–based string functions, and using the Unicode character set (and Unicode string functions).

The next chapter discusses managing large projects in assembly language, particularly how to create library modules, which will prove useful for combining several string functions into a single library module.

14.6 For More Information

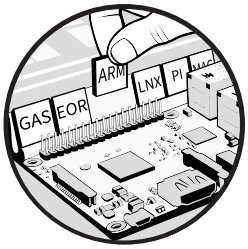
- The official Unicode website is the main source for information about all things concerning the Unicode standard. This is the closest thing to an official Unicode library: *https://icu.unicode.org/home*.
- GNU offers a Unicode string library at *https://www.gnu.org/software/libunistring/manual/libunistring.html*.
- You can find the source code for the functions from the GNU C stdlib for AARCH64 discussed in this chapter at *https://github.com/bminor/glibc/tree/master/sysdeps/aarch64*.
- For the GitHub repository containing the source code for the glibc string functions written in ARM assembly, see *https://github.com/ARM-software/optimized-routines/tree/master/string/aarch64*. You can study this code to learn advanced string-handling tricks in assembly language.
- For help writing ARM string-handling functions in assembly language, you can post questions to a forum I've set up at *https://forums.randallhyde.com*.
- For a list of various Unicode string libraries, see *https://unicodebook.readthedocs.io/libraries.html#libunistring*.
- Wikipedia offers more detail on Unicode code points at *https://en.wikipedia.org/wiki/Unicode#General_Category_property*.
- My website links to information on the HLA string library (x86): *https://www.randallhyde.com/AssemblyLanguage/HighLevelAsm/HLADoc/index.html*.
- ARM Developer assembly language string library source code can be found at *https://developer.arm.com/documentation/102620/0100/Optimized-string-routines---libastring*.
- For more on the `setlocale()` function, see *https://man7.org/linux/man-pages/man3/setlocale.3.html*. For an explanation of the locale string format, see *https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html* (or search for **locale function**).

TEST YOURSELF

1. What is a zero-terminated string?
2. Why are zero-terminated string functions so slow (in general)?
3. How does the string assembly language type presented in this chapter generally improve the performance of string functions? Name at least three reasons.
4. Why can't the `str.substr` function copy the source data on a 16-byte boundary?
5. Why are most Unicode string functions intrinsically slow?

15

MANAGING COMPLEX PROJECTS



Most assembly language source files are not stand-alone programs. In general, you must call various standard library or other routines that are not defined in your main program because attempting to write such code as part of your application would be far too much work (and poor programming practice).

For example, the ARM doesn't provide machine instructions like read, write, or put for doing I/O operations. The functions in this book contain thousands of lines of source code to accomplish these operations. For small programs, working with a single source file is fine, but for large programs, this gets cumbersome. Programming would be formidable if you had to merge these thousands of lines of code into your simple programs, which would then compile slowly. Furthermore, once you've debugged and tested a large section of your code, continuing to assemble that same code when you make a small change to another part of your program is a waste of

time. Imagine having to wait 20 or 30 minutes on a fast PC to assemble a program after making a one-line change!

Programming in the large is the term software engineers have coined to describe the processes, methodologies, and tools for reducing the development time of large software projects. While everyone has their own idea of what "large" is, *separate compilation* is one of the more popular techniques that support programming in the large. First, you break your large source files into manageable chunks. Then you compile the separate files into object code modules. Finally, you link the object modules together to form a complete program. If you need to make a small change to one of the modules, you need to reassemble only that one module rather than the entire program.

This chapter describes the tools that Gas and your OS provide for separate compilation and how to effectively employ these tools in your programs.

15.1 The .include Directive

The `.include` directive, when encountered in a source file, switches program input from the current file to the file specified in the operand field of the include directive. Section 1.5, "The *aoaa.inc* Include File," on page 10 described the `.include` directive as a way to include code from separate source files into the current assembly, allowing you to construct text files containing common constants, types, source code, and other Gas items into the assembly. As noted in that section, the syntax for the `.include` directive is

```
.include "filename"
```

where *filename* must be a valid filename.

By this book's convention, Gas include files have a *.inc* (include) suffix. Gas, however, does not require include files to have this suffix; any filename containing Gas assembly language source will work. Gas merges the specified file into the compilation at the point of the `.include` directive. You can nest `.include` statements inside files you include; that is, a file being included into another file during assembly may itself include a third file.

Using the `.include` directive by itself does not provide separate compilation. You *could* use `.include` to break up a large source file into separate modules and join these modules together when you compile your file. The following example would include the *print.inc* and *getTitle.inc* files during the compilation of your program:

```
.include  "print.inc"
.include  "getTitle.inc"
```

Your program will now benefit from modularity. Alas, you will not save any development time. The `.include` directive inserts the source file at the point of the `.include` during compilation, exactly as though you had typed

that code yourself. Gas still has to compile the code, and that takes time. If you are including a large number of source files (such as a huge library) into your assembly, the compilation process could take *forever*.

In general, you should *not* use the `.include` directive to include source code as shown in the previous example, as this code won't allow you to take advantage of separate compilation. Instead, use the `.include` directive to insert a common set of constants, types, external procedure declarations, and other such items into a program. Typically, an assembly language include file does *not* contain any machine code (outside of a macro; see Chapter 13 for details). The purpose of using `.include` files in this manner will become clearer after you see how the external declarations work (see section 15.3, "Assembly Units and External Directives," on the next page).

If your assembly language source files have a *.S* suffix, you can also use the `#include "filename"` directive to include a source file. This is generally preferable because you can use the CPP directives in such include files (you can't in a standard `.include` file). The rest of this chapter assumes the use of the `#include` directive rather than `.include`.

15.2 Ignoring Duplicate Include Operations

As you begin to develop sophisticated modules and libraries, you will eventually discover a big problem: some header files need to include other header files. Technically, this is fine in and of itself, but issues arise when one header file includes another, and that second header file includes another, and so on, such that the final header file includes the first header file.

There are two problems with a header file indirectly including itself. First, this creates an infinite loop in the compiler. The compiler will happily go on about its business including all these files over and over again until it runs out of memory or another error occurs. Second, when Gas includes a header file for the second time, it starts complaining bitterly about duplicate symbol definitions. After all, the first time it reads the header file, it processes all the declarations in that file; the second time around, it views all those symbols as duplicate symbols.

The standard technique for resolving recursively included files, well-known to C/C++ programmers, is to use conditional assembly to have Gas ignore the content of an include file. (See Chapter 13 for a discussion of conditional assembly for CPP and the Gas CTL.) The trick is to place an `#ifdef` (if defined) statement around all statements in the include file. Specify an undefined symbol as the `#ifdef` operand (I tend to use the include file's filename, substituting underlines for periods). Then, immediately after the `#ifdef` statement, define that symbol; using a numeric equate and assigning the symbol the constant 1 is typical. Here's an example of this `#ifdef` usage in action:

```
#ifdef  myinclude_inc   // Filename: myinclude.inc
#define myinclude_inc 1
```

```
 Put all the source code lines for the include file here.

// The following statement should be the last nonblank line
// in the source file:

#endif  // myinclude_inc
```

If you attempt to include *myinclude.inc* a second time, the `#ifdef` directive will cause Gas (actually, the CPP) to skip over all the text up to the corresponding `#endif` directive, thus avoiding the duplicate definition errors.

15.3 Assembly Units and External Directives

An *assembly unit* is the assembly of a source file plus any files it directly or indirectly includes. An assembly unit produces a single *.o* (object) file after assembly. The linker takes multiple object files (produced by Gas or other compilers, such as GCC) and combines those object files into a single executable file. The main purpose of this section, and, indeed, this whole chapter, is to describe how these assembly units (*.o* files) communicate linkage information to one another during the linking process. Assembly units are the basis for creating modular programs in assembly language.

To use Gas's assembly unit facilities, you must create at least two source files. One file contains a set of variables and procedures used by the second. The second file uses those variables and procedures without knowing how they're implemented.

Technically, the `#include` directive provides you with all the facilities you need to create such modular programs. You can create several modules, each containing a specific routine, and include those modules, as necessary, in your assembly language programs by using the `#include` directive. However, if you use this method, including a routine you've debugged in a compilation will still waste time because Gas must recompile bug-free code whenever you assemble the main program. A much better solution is to preassemble the debugged modules and link the object code modules together, using Gas's `.global` and `.extern` directives, which this section covers.

All the programs appearing in this book up to this point have been separately assembled modules that happen to link with a C/C++ main program rather than another assembly language module. In every program thus far, the assembly language "main program" has been named `asmMain`, which is nothing but a C++-compatible function that the generic *c.cpp* program has called from its main program. For example, consider the body of `asmMain` from Listing 1-3 on page 9 (for Linux and Pi OS systems):

```
// Listing1-3.S
//
// A simple Gas module that contains
// an empty function to be called by
// the C++ code in Listing 1-2.
```

```
        .text

// Here is the asmMain function:

        .global asmMain
        .align  2    // Guarantee 4-byte alignment.
asmMain:

// Empty function just returns to C++ code:

        ret          // Returns to caller
```

This `.global asmMain` statement has been included in every program that has had an `asmMain` function without any definition or explanation. It's now time to deal with that oversight.

Normal symbols in a Gas source file are *private* to that particular source file and are inaccessible from other source files (that don't directly include the file containing those private symbols, of course). That is, the *scope* of most symbols in a source file is limited to those lines of code within that particular source file and any files it includes. The `.global` directive tells Gas to make the specified symbol *global* to the assembly unit—accessible by other assembly units during the link phase. By placing the `.global asmFunc` statement in the example programs appearing throughout this book, these sample programs have made the `asmMain` symbol global to the source file containing them so that the *c.cpp* program can call the `asmMain` function.

As you may recall, macOS requires an underscore prefix in front of the global name. This means you would use `.global _asmMain` and `_asmMain:` if you wanted this source file to assemble under macOS. The *aoaa.inc* header file resolves this issue in a portable fashion, but the code from *Listing1-3.S* does not include *aoaa.inc*.

Simply making a symbol public is insufficient to use that symbol in another source file. The source file that wants to use the symbol must also declare that symbol as *external*. This notifies the linker that it will have to patch in the address of a public symbol whenever the file with the external declaration uses that symbol. For example, the *c.cpp* source file defines the `asmMain` symbol as external in the following lines of code (for what it's worth, this declaration also defines the external symbol `getTitle`):

```
// extern "C" namespace prevents
// "name mangling" by the C++
// compiler.

extern "C"
{
    // Here's the external function,
    // written in assembly language,
    // that this program will call:

    void asmMain( void );
    int readLine( char *dest, int maxLen );
};
```

In this example, `readLine` is actually a C++ function defined in the *c.cpp* source file. C/C++ does not have an explicit public declaration. Instead, if you supply the source code for a function in a source file that declares that function to be external, C/C++ will automatically make that symbol public by virtue of the external declaration.

When you place an `.extern` directive in your program, Gas treats that declaration the same as any other symbol declaration. If the symbol already exists, Gas will generate a symbol redefinition error. Generally, you should place all external declarations near the beginning of the source file to avoid any scoping / forward reference issues.

Technically speaking, using the `.extern` directive is optional, as Gas assumes that any symbol you use that is not defined in a source file is an external symbol. The linker will report actual undefined symbols if it fails to find a symbol when linking in all the other object code modules. However, it is good programming style to explicitly define your external symbols with `.extern` in order to make your intentions clear to others reading your source code.

Because the `.global` directive does not actually define the symbol, its placement is not as critical as the `.extern` directive's is. Some programmers put all the global declarations at the beginning of a source file; others put the global declaration right before the definition of the symbol (as I've done with the `asmMain` symbol in most of the same programs). Either is fine.

Because a public symbol from one source file can be used by many assembly units, a problem develops: you have to replicate the `.extern` directive in all the files that use that symbol. For a small number of symbols, this is not much of an issue. However, as the number of external symbols increases, maintaining all these external symbols across multiple source files becomes burdensome.

The Gas solution is the same as the C/C++ solution: *header files*. These are simply include files containing external (and other) declarations that are common among multiple assembly units. Header files get their name from the fact that the include statement that injects their code into a source file usually appears at the beginning (the "head") of the source file that uses them. This turns out to be the primary use of include files in Gas.

15.4 Creating a String Library with Separate Compilation

Chapter 14 provided several examples of string-handling functions along with macros and the string structure. The problem with these functions and declarations is that they must be cut and pasted into any source file that wants to use them. It would be far better to create a header file containing the macros, structures, and external symbol definitions and then compile the individual functions into *.o* files to link with those programs that want to use those functions. This section describes how to create linkable object modules for these string functions.

The header file for the strings library is *strings.inc*:

```
// strings.inc
//
// String function header file for the assembly
// language string format

#ifndef  strings_inc
#define strings_inc 1

// Assembly language string data structure:

            struct  string, -16
            dword   string.allocPtr // At offset -16
            word    string.maxlen   // At offset -8
            word    string.len      // At offset -4
            byte    string.chars    // At offset 0

            // Note: characters in string occupy offsets
            // 0 ... in this structure.

            ends    string

// str.buf
//
// Allocate storage for an empty string
// with the specified maximum size:

            .macro  str.buf strName, maxSize
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   \maxSize
            .word   0
\strName:   .space  ((\maxSize+16) & 0xFFFFFFF0), 0
            .endm

// str.literal
//
// Allocate storage for a string buffer and initialize
// it with a string literal:

            .macro  str.literal strName, strChars
            .align  4   // Align on 16-byte boundary.
            .dword  0   // NULL ptr for allocation ptr
            .word   len_\strName    // string.maxlen
            .word   len_\strName    // string.len

            // Emit the string data and compute the
            // string's length:

\strName:   .ascii  "\strChars"
len_\strName=       .-\strName
            .byte   0   // Zero-terminating byte
```

```
            // Ensure object is multiple of 16 bytes:

            .align  4
            .endm

// str.len
//
//          Return the length of the string pointed at by X0.
//          Returns length in X0

            .macro  str.len
            ldr     w0, [x0, #string.len]
            .endm

// External declarations:

            .extern str.cpy
            .extern str.cmp
            .extern str.substr
            .extern str.bufInit
            .extern str.alloc
            .extern str.free

// This would be a good place to include external
// declarations for any string functions you write.

#endif
```

The source file for the `str.cpy` function is in *str.cpy.S*:

```
// str.cpy.S
//
// A str.cpy string copy function

            #include    "aoaa.inc"
          ❶ #include    "strings.inc"

            .code

////////////////////////////////////////////////////////
//
// str.cpy
//
// Copies the data from one string variable to another
//
// On entry:
//
// X0- Pointer to source string (string struct variable)
// X1- Pointer to destination string
//
// On exit:
//
// Carry flag clear if no errors, carry is set if
// the source string will not fit in the destination.
```

```
❷ proc    str.cpy, public

          locals  str_cpy
          qword   str_cpy.saveV0
          qword   str_cpy.saveX2X3
          dword   str_cpy.saveX4
          byte    str_cpy.stkSpace,64
          endl    str_cpy

          enter   str_cpy.size

          // Preserve X2 ... X4 and V0:

          str     q0,     [fp, #str_cpy.saveV0]
          stp     x2, x3, [fp, #str_cpy.saveX2X3]
          str     x4,     [fp, #str_cpy.saveX4]

          // Ensure the source will fit in the destination
          // string object:

          ldr     w4, [x0, #string.len]
          ldr     w3, [x1, #string.maxlen]
          cmp     w4, w3
          bhi     str.cpy.done    // Note: carry is set.

          // Set the length of the destination string
          // to the length of the source string.

          str     w4, [x1, #string.len]

          // X4 contains the number of characters to copy;
          // while this is greater than 16, copy 16 bytes
          // at a time from source to dest:

          mov     x2, x0  // Preserve X0 and X1.
          mov     x3, x1

cpy16:    ldr     q0, [x2], #16
          str     q0, [x3], #16
          subs    w4, w4, #16
          bhi     cpy16

// At this point, you have fewer than 16 bytes to copy. If
// W4 is not 0, just copy 16 remaining bytes (you know,
// because of the string data structure, that if you have at
// least 1 byte left to copy, you can safely copy
// 16 bytes):

          beq     setZByte    // Skip if 0 bytes.

          ldr     q0, [x2]
          str     q0, [x3]

// Need to add a zero-terminating byte to the end of
// the string. Note that maxlen does not include the
```

```
// 0 byte, so it's always safe to append the 0
// byte to the end of the string.

setZByte:   ldr     w4,  [x0, #string.len]
            strb    wzr, [x1, w4, uxtw]

            adds    wzr, wzr, wzr   // Clears the carry

str.cpy.done:
            ldr     q0,     [fp, #str_cpy.saveV0]
            ldp     x2, x3, [fp, #str_cpy.saveX2X3]
            ldr     x4,     [fp, #str_cpy.saveX4]
            leave
            endp    str.cpy
```

The *str.cpy.S* source file is created by including the *strings.inc* header file ❶ and cutting and pasting the `str.cpy` function ❷ from Listing 14-2 on page 819. Note the public argument after the `proc` macro. This causes the `proc` macro to emit a `.global` directive for the `str.cpy` symbol so that the function is available to other source files.

The *str.cmp.S*, *str.substr.S*, *str.alloc.S*, *str.free.S*, and *str.bufInit.S* source files are created from their corresponding functions (in Chapter 14) in a similar fashion. I won't include those source files here because they are redundant and consume too much space, but you can find copies in the online source files at *https://artofarm.randallhyde.com*.

If you try to use the usual `build` command to assemble any of these modules, you get an error from the system complaining about missing symbols. This is because these modules are not stand-alone assembly language programs. In the following section, I describe the correct way to build these library modules; in the meantime, here are some cheesy commands that will assemble these files without error (though there will be a warning):

```
./build -c str.cpy
./build -c str.cmp
./build -c str.substr
./build -c str.bufInit
./build -c str.alloc
./build -c str.free
```

This will assemble the files without running the linker (`-c` means *compile only*), generating the files *str.cpy.o*, *str.cmp.o*, *str.substr.o*, *str.bufInit.o*, *str.alloc.o*, and *str.free.o*, respectively. Of course, the next question is how to link these files with an application program. Listing 15-1 is an amalgamation of various `asmMain` functions from Chapter 14 that make calls to the `str.cpy`, `str.cmp`, and `str.substr` functions.

```
// Listing15-1.S
//
// A program that calls various string functions

            #include    "aoaa.inc"
```

```
            #include    "strings.inc"

//////////////////////////////////////////////////////////

            .data

            str.buf     destination, 256
            str.literal src,    "String to copy"
            str.literal left,   "some string"
            str.literal right1, "some string"
            str.literal right2, "some string."
            str.literal right3, "some string"

            str.buf     smallDest, 32
            str.literal dest,   "Initial destination string"

//                            1111111111222222222233333
//                  012345678901234567890123456789012345
str.literal source, "Hello there, world! How's it going?"

fmtStr:     .asciz      "source='%s', destination='%s'\n"
ltFmtStr:   .asciz      "Left ('%s') is less than right ('%s')\n"
gtFmtStr:   .asciz      "Left ('%s') is greater than right ('%s')\n"
eqFmtStr:   .asciz      "Left ('%s') is equal to right ('%s')\n"

successStr: .asciz      "substr( '%s', %2d, %3d )= '%s'\n"
failureStr: .asciz      "substr( '%s', %2d, %3d ) failed\n"

//////////////////////////////////////////////////////////

            .code
ttlStr:     wastr  "Listing15-1"

// Standard getTitle function
// Returns pointer to program name in X0

            proc    getTitle, public
            lea     x0, ttlStr
            ret
            endp    getTitle

//////////////////////////////////////////////////////////
//
// prtResult
//
// Utility function to print the result of a string
// comparison:

            proc    prtResult

            mov     x2, x1
            mov     x1, x0
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
```

```
            beq     strsEQ
            bhi     strGT

            // Must be LT at this point.

            lea     x0, ltFmtStr
            b       printf

strsEQ:     lea     x0, eqFmtStr
            b       printf

strGT:      lea     x0, gtFmtStr
            b       printf

            endp    prtResult

//////////////////////////////////////////////////////////
//
// testSubstr
//
// Utility function to test call to str.substr
//
// On entry:
// X0, X1, X2, X3 -- str.substr parameters

            proc    testSubstr

            locals  testSS
            byte    testSS.stkspace, 64
            endl    testSS

            enter   testSS.size

            lea     x5, successStr
            bl      str.substr
            bcc     success
            lea     x5, failureStr

success:
            mov     x4, x3
            mov     x3, x2
            mov     x2, x1
            mov     x1, x0
            mov     x0, x5
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            mstr    x3, [sp, #16]
            mstr    x4, [sp, #24]
            bl      printf
            leave
            endp    testSubstr

//////////////////////////////////////////////////////////
```

```
//
// Main program to test the code:

            proc    asmMain, public

            locals  lcl
            byte    stkSpace, 64
            endl    lcl

            enter   lcl.size      // Reserve space for locals.

            lea     x0, src
            lea     x1, destination
            bl      str.cpy

            mov     x2, x1
            mov     x1, x0
            lea     x0, fmtStr
            mstr    x1, [sp]
            mstr    x2, [sp, #8]
            bl      printf

            lea     x0, left
            lea     x1, right1
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right2
            bl      str.cmp
            bl      prtResult

            lea     x0, left
            lea     x1, right3
            bl      str.cmp
            bl      prtResult

            lea     x0, source
            mov     x1, #0
            mov     x2, #11
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #20
            mov     x2, #15
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #20
            mov     x2, #20
```

```
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #40
            mov     x2, #20
            lea     x3, dest
            bl      testSubstr

            lea     x0, source
            mov     x1, #0
            mov     x2, #100
            lea     x3, smallDest
            bl      testSubstr

AllDone:    leave
            endp    asmMain
```

If you try to build this program with the following command

```
./build Listing15-1
```

the system will complain that it cannot locate the symbols `str.cpy`, `str.cmp`, and `str.substr` in the object files provided. Unfortunately, the *build* shell script doesn't support linking in multiple object modules (other than the *c.cpp* and the specified file's object files). Therefore, you must specify an explicit `g++` command to process all the files:

```
g++ -DisMacOS c.cpp Listing15-1.S str.cpy.o str.cmp.o str.substr.o -o Listing15-1
```

The `-DisMacOS` command line argument should be changed to `-DisLinux` when compiling the code under Linux or Pi OS (rather than macOS). As you may recall from section 1.10.1, "Assembling Programs Under Multiple OSes," on page 36, the *build* shell script determines the OS and emits a `g++` command line define (the `-D`*xxxx* option) to make the OS known to the assembly source file (and, especially, the *aoaa.inc* header file). As this `g++` command will attempt to assemble the *Listing15-1.S* source file (which includes *aoaa.inc*), the command line must include a definition of either `isMacOS` or `isLinux`, or the assembly will fail.

This `g++` command will compile *c.cpp*, assemble *Listing15-1.S*, and link their object files together with the *str.cpy.o*, *str.cmp.o*, and *str.substr.o* object files. This assumes, of course, that you've already assembled the *str.*.S* source files and that their object files are sitting in the current directory. The sample program in Listing 15-1 does not call the `str.alloc`, `str.free`, or `str.bufInit` functions, so there was no need to link in their respective object code files, though doing so would not have generated an error.

Here's the full set of commands needed to build all these files and generate and run the Listing 15-1 executable:

```
% g++ -c -DisMacOS str.cpy.S
% g++ -c -DisMacOS str.cmp.S
% g++ -c -DisMacOS str.substr.S
% g++ -DisMacOS c.cpp Listing15-1.S str.cpy.o str.cmp.o str.substr.o -o Listing15-1
% ./Listing15-1
Calling Listing15-1:
source='String to copy', destination='String to copy'
Left ('some string') is equal to right ('some string')
Left ('some string') is less than right ('some string.')
Left ('some string') is greater than right ('some string')
substr( 'Hello there, world! How's it going?',  0,  11 )= 'Hello there'
substr( 'Hello there, world! How's it going?', 20,  15 )= 'How's it going?'
substr( 'Hello there, world! How's it going?', 20,  20 )= 'How's it going?'
substr( 'Hello there, world! How's it going?', 40,  20 )= ''
substr( 'Hello there, world! How's it going?',  0, 100 ) failed
listing15-1 terminated
```

Granted, that's a lot of typing in order to compile and link a simple source file. You could remedy this by putting all the commands into a text file and executing them as a shell script (similar to the *build* script), but there's a better way: makefiles.

15.5 Introducing Makefiles

The *build* file used throughout this book has been far more convenient than the manual commands needed to build the example of the previous section. Unfortunately, the build mechanism that *build* supports is good for only a few fixed source files. While you could easily construct a shell script to compile all the files in a large assembly project, it would largely defeat the purpose of using separate assembly, as running the script file would reassemble every source file in the project. Although you can use complex command line functions to avoid some of this, it's easier to use makefiles.

A *makefile* is a script in a language (designed in early releases of Unix) that specifies how to execute a series of commands based on certain conditions. In its simplest form, a makefile can behave exactly like a shell script; you can list a sequence of commands in a text file and have the `make` program execute them. Of course, there would be no benefit over using a shell script if you did this; you should take advantage of `make`'s features if you're going to use makefiles.

The `make` program is an executable, just like Gas (as) or GCC. As `make` is not a part of the Linux or macOS system, you must obtain a `make` program before you can use it. Fortunately, `make` comes preinstalled on most Linux and macOS distributions (if you can run GCC, you can certainly run `make`). Execute it from a command line as follows:

```
make optionalArguments
```

If you execute `make` on a command line without any arguments, `make` will search for a file named *Makefile* and attempt to process the commands

in that file. For many projects, this is very convenient. If you put all your source files in a single directory (potentially with subdirectories) along with a single makefile (named *Makefile*), you can then change into that directory and execute `make`, building the project with minimal fuss.

If you like, you can use a different filename than *Makefile*. Rather than simply placing the filename after `make` on the command line, however, you must preface the filename with the `make -f` option as follows:

```
make -f mymake.mak
```

You don't need to give your filename the *.mak* extension, but this is a popular convention when using makefiles with custom names.

The `make` program provides many command line options, and you can list common ones by using `--help`. You can look up `make` documentation online (or type `man make` from the command line) for a description of the other command line options, but most of them are advanced and unnecessary for most tasks.

Of course, to make practical use of `make`, you need to create makefiles. The following subsections describe the `make` scripting language and some common conventions for makefiles.

15.5.1 Basic Makefile Syntax

A makefile is a standard ASCII text file containing a sequence of lines (or a set of multiple occurrences of this sequence) as follows:

```
target: dependencies
    commands
```

All components of this code—*target*, *dependencies*, and *commands*—are optional. The *target* item is an identifier or filename of some sort that, if present, must begin in column 1 of its source line. The *dependencies* item is a list of filenames on which the target depends in order to be built properly. The *commands* item is a list of one or more command line commands, which must have at least one tab character in front of them.

Consider the following makefile, which builds a set of string library functions (note that a tab appears before each `g++` command):

```
all:
    g++ -c -DisMacOS str.cpy.S
    g++ -c -DisMacOS str.cmp.S
    g++ -c -DisMacOS str.substr.S
    g++ -DisMacOS c.cpp Listing15-1.S str.cpy.o str.cmp.o \
         str.substr.o -o Listing15-1
```

If these commands appear in a file named *Makefile* and you execute `make`, they will execute exactly as the command line interpreter would have executed them, had they appeared in a shell script.

Consider the following modification of the previous makefile:

```
executable:
  g++ -c -DisMacOS Listing15-1.S
  g++ -DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1

library:
  g++ -c -DisMacOS str.cpy.S
  g++ -c -DisMacOS str.cmp.S
  g++ -c -DisMacOS str.substr.S
```

This separates the build commands into two groups: one specified by the `executable` label and another specified by the `library` label.

If you run `make` without any command line options, it will execute only the commands appearing after the first target in the file. Therefore, in this example, if you run `make` by itself, it will assemble *Listing15-1.S*, compile *c.cpp*, and attempt to link (the resulting) *c.obj* with *str.cpy.o*, *str.cmp.o*, *str.substr.o*, and *Listing15-1.o*. Assuming you had previously compiled the string functions, this should successfully produce the `Listing15-1` executable (without having to recompile the string functions).

To convince `make` to process the commands after the library target, you must specify the target name as a `make` command line argument:

```
make library
```

This `make` command compiles *str.cpy.S*, *str.cmp.S*, and *str.substr.S*. If you execute this command once (and never change the string functions thereafter), you need only execute the `make` command by itself to generate the executable file. You can also use `make executable` if you want to explicitly state that you are building the executable.

The ability to specify which targets you want to build on the command line is useful. However, as your projects get larger, with many source files and library modules, keeping track of which source files you need to recompile all the time can be burdensome and error-prone. If you're not careful, you'll forget to compile an obscure library module after you've made changes to it and wonder why the application is still failing. The `make` dependencies option helps you avoid these problems by allowing you to automate the build process.

A list of one or more whitespace-separated dependencies can follow a target in a makefile:

```
target: dependency1 dependency2 dependency3 ...
```

Dependencies are either target names (of targets appearing in that makefile) or filenames. If a dependency is a target name (that is not also a filename), `make` will go execute the commands associated with that target. Consider the following makefile (if compiling under Linux or Pi OS, be

sure to change the -DisMacOS command line option to -DisLinux in this example and all that follow):

```
executable:
  g++-c -DisMacOS Listing15-1.S
  g++-DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1

library:
  g++ -c -DisMacOS str.cpy.S
  g++-c -DisMacOS str.cmp.S
  g++-c -DisMacOS str.substr.S

all: library executable
```

The all target in this code does not have any commands associated with it. Instead, the all target depends on the library and executable targets, so it will go execute the commands associated with those targets, beginning with library. This is because the library object files must be built before the associated object modules can be linked into the executable program). The all identifier is a common target in makefiles. Indeed, it is often the first or second target to appear in a makefile.

If a *target*: *dependencies* line becomes too long to be readable (make doesn't care much about line length), you can break the line into multiple lines by putting a backslash character (\) as the last character on a line. The make program will combine source lines that end with a backslash with the next line in the makefile. The backslash must be the very last character on the line; whitespace characters (tabs and spaces) are not allowed to follow the backslash.

Target names and dependencies can also be filenames. Specifying a filename as a target name is generally done to tell the make system how to build that particular file. For example, you could rewrite the current example as follows:

```
executable:
  g++ -c -DisMacOS Listing15-1.1
  g++ -DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1

library: str.cpy.o str.cmp.o str.substr.o

str.cpy.o:
  g++ -c -DisMacOS str.cpy.S

str.cmp.o:
  g++ -c -DisMacOS str.cmp.S

str.substr.o:
  g++ -c -DisMacOS str.substr.S

all: library executable
```

When dependencies are associated with a target that is a filename, you can read the *target*: *dependencies* statement as "*target* depends on *dependencies*."

When processing a command, make compares the modification date/timestamp of the files specified as target and dependency filenames.

If the date/time of the target is older than *any* of the dependencies (or the target file doesn't exist), make will execute the commands after the target. If the target file's modification date/time is later (newer) than *all* of the dependent files, make will not execute the commands. If one of the dependencies after a target is itself a target elsewhere, make will first execute that command (to see if it modifies the target object, changing its modification date/time, and possibly causing make to execute the current target's commands). If a target or dependency is just a label (not a filename), make will treat its modification date/time as older than any file.

Consider the following modification to the running makefile example:

```
Listing15-1:Listing15-1.o str.cpy.o str.cmp.o str.substr.o
  gcc -DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1

Listing15-1.o:
  g++ -c -DisMacOS Listing15-1.S

str.cpy.o:
  g++ -c -DisMacOS str.cpy.S

str.cmp.o:
  g++ -c -DisMacOS str.cmp.S

str.substr.o:
  g++ -c -DisMacOS str.substr.S
```

This code has removed all and library targets, as they turn out to be unnecessary, and changed executable to Listing15-1, the final target executable file.

Because str.cpy.o, str.cmp.o, str.substr.o, and Listing15-1.o are all targets (as well as filenames), make will first go process those targets. After that, make will compare the modification date/time of Listing15-1 against that of the four object files. If Listing15-1 is older than any of those object files, make will execute the command following the Listing15-1 target line (to compile *c.cpp* and link it with the object files). If Listing15-1 is newer than its dependent object files, make will not execute the command.

The same process happens, recursively, for each of the dependent object files following the Listing15-1 target. While processing the Listing15-1 target, make will also process the str.cpy.o, str.cmp.o, str.substr.o, and Listing15-1.o targets (in that order). In each case, make compares the modification date/time of the *.o* file with the corresponding *.S* file. If the *.o* file is newer than the *.S* file, make returns to processing the Listing15-1 target without doing anything; if the *.o* file is older than the *.S* file (or doesn't exist), make executes the corresponding g++ command to generate a new *.o* file.

If Listing15-1 is newer than all the *.o* files (and they are all newer than the *.S* files), then executing make simply reports that Listing15-1 is up-to-date, but it will not execute any of the commands in the makefile. If any of the files are out-of-date (because they've been modified), this makefile will compile and link only the files necessary to bring Listing15-1 up-to-date.

The makefiles have a pretty serious defect thus far: they are missing an important dependency. Since all the *.S* files include the *aoaa.inc* file, a change to *aoaa.inc* could possibly require a recompilation of these *.S* files. Listing 15-2 adds this dependency to the *Listing15-2.mak* makefile, and it also demonstrates how to include comments in a makefile by using the # character at the beginning of a line.

```
# Listing15-2.mak
#
# makefile for Listing15-1

Listing15-1:Listing15-1.o str.cpy.o str.cmp.o str.substr.o
  gcc -DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1

Listing15-1.o:aoaa.inc Listing15-1.S
  gcc -c -DisMacOS Listing15-1.S

str.cpy.o:aoaa.inc str.cpy.S
  gcc -c -DisMacOS str.cpy.S

str.cmp.o:aoaa.inc str.cmp.S
  gcc -c -DisMacOS str.cmp.S

str.substr.o:aoaa.inc str.substr.S
  gcc -c -DisMacOS str.substr.S
```

Here's an example of executing `make` (under macOS):

```
% make -f Listing15-2.mak
gcc -c -DisMacOS Listing15-1.S
gcc -c -DisMacOS str.cpy.S
gcc -c -DisMacOS str.cmp.S
gcc -c -DisMacOS str.substr.S
gcc -DisMacOS c.cpp Listing15-1.o str.cpy.o str.cmp.o str.substr.o -o Listing15-1
```

To execute this command under Linux or Pi OS, don't forget to change all the `-DisMacOS` command line options to `-DisLinux` in the makefile and make sure all commands have a tab in column 1. If you want to be able to automatically compile the code for any OS, simply steal the code from the *build* script that sets up a shell variable with the appropriate command line option, as shown in Listing 15-3.

```
# Listing15-3.mak
#
# makefile for Listing15-1 with dependencies that will
# automatically set up the define for the OS

❶ unamestr=`uname`

Listing15-1:Listing15-1.o str.cpy.o str.cmp.o str.substr.o
    gcc -D$(unamestr) c.cpp Listing15-1.o str.cpy.o str.cmp.o \
        str.substr.o -o Listing15-1
```

```
Listing15-1.o:aoaa.inc Listing15-1.S
  ❷ gcc -c -D$(unamestr) Listing15-1.S

str.cpy.o:aoaa.inc str.cpy.S
    gcc -c -D$(unamestr) str.cpy.S

str.cmp.o:aoaa.inc str.cmp.S
    gcc -c -D$(unamestr) str.cmp.S

str.substr.o:aoaa.inc str.substr.S
    gcc -c -D$(unamestr) str.substr.S
```

The first statement ❶ is an example of a makefile *macro* (or *variable*). The OS command `uname` will display the OS (kernel) name. Under Linux systems, this will be replaced by the string `Linux`, and on macOS systems by the string `Darwin` (the internal name of the macOS kernel).

Makefile macros use deferred execution. This means that the macro `unamestr` actually contains the text `` `uname` `` and that the `uname` command will execute in place when the `make` program expands the `unamestr` macro. The `make` program will expand the `-D$(unamestr)` command line option, producing `-D`uname`` ❷. The backticks (`` ` ``) tell `make` to execute the command and replace it with the text printed by the command: the OS kernel name.

The only issue is that the `uname` command prints Linux or Darwin, so the `-D` command defines one of these two symbols. The *build* script translates these strings to `isMacOS` and `isLinux`. I originally did this because the symbol `Linux` would likely appear in a Linux-based assembly language program. Unfortunately, the symbol translation trick didn't work out in a makefile, so I modified *aoaa.inc* to accept `Linux` and `Darwin` as well as `inLinux` and `inMacOS`. I modified *aoaa.inc* to do the translation and undefine Linux or Darwin, should those symbols get used:

```
// Makefiles define the symbols Darwin (for macOS)
// and Linux (for Linux) rather than isMacOS and
// isLinux. Deal with that here:

#ifdef Darwin
    #define isMacOS (1)
    #undef isLinux
    #undef Darwin
#endif
#ifdef Linux
    #define isLinux (1)
    #undef isMacOS
    #undef Linux
#endif
```

Here's the execution of the `make` command to build the code for Listing 15-3 (assuming no object files were already created):

```
% make -f Listing15-3.mak
g++ -c -D`uname` Listing15-1.S
```

```
g++ -c -D`uname` str.cpy.S
g++ -c -D`uname` str.cmp.S
g++ -c -D`uname` str.substr.S
g++ -D`uname` c.cpp Listing15-1.o str.cpy.o str.cmp.o \
    str.substr.o -o Listing15-1
```

Note that `-D`uname`` is translated to either `-DLinux` or `-DDarwin`, depending on the OS.

15.5.2 Make Clean and Touch

One common target you will find in most professionally made makefiles is `clean`, which deletes an appropriate set of files to force the entire system to be remade the next time you execute the makefile. This command typically deletes all the *.o* and executable files associated with the project.

Listing 15-4 provides an example `clean` target for the makefile appearing in Listing 15-3.

```
# Listing15-4.mak
#
# makefile for listing15-1 with dependencies that will
# automatically set up the define for the OS
#
# Demonstrates the clean target

unamestr=`uname`

Listing15-1:Listing15-1.o str.cpy.o str.cmp.o str.substr.o
    gcc -D$(unamestr) c.cpp listing15-1.o str.cpy.o str.cmp.o \
        str.substr.o -o Listing15-1

Listing15-1.o:aoaa.inc Listing15-1.S
    gcc -c -D$(unamestr) Listing15-1.S


str.cpy.o:aoaa.inc str.cpy.S
    gcc -c -D$(unamestr) str.cpy.S

str.cmp.o:aoaa.inc str.cmp.S
    gcc -c -D$(unamestr) str.cmp.S

str.substr.o:aoaa.inc str.substr.S
    gcc -c -D$(unamestr) str.substr.S


clean:
    rm str.cpy.o
    rm str.cmp.o
    rm str.substr.o
    rm Listing15-1.o
    rm c.o
    rm Listing15-1
```

Issuing the command

```
% make -f Listing15-4.mak clean
```

will delete all the executable and object code files associated with the project.

To force the recompilation of a single file (without manually editing and modifying it), you can use the Unix utility `touch`. This program accepts a filename as its argument and updates the modification date/time of the file (without otherwise modifying the file). For example, after building *Listing15-1.S* using the makefile in Listing 15-4, were you to execute the command

```
touch Listing15-1.S
```

and then re-execute the makefile in Listing 15-4, `make` would reassemble the code in *Listing15-1.S*, recompile *c.cpp*, and produce a new executable.

15.6 Generating Library Files with the Archiver Program

Many common projects reuse code the developers created long ago, or code from a source outside the developer's organization. These libraries of code are relatively *static*: they rarely change during the development of a project that uses them. In particular, you would not usually incorporate the building of the libraries into a given project's makefile. A specific project might list the library files as dependencies in the makefile, but the assumption is that the library files are built elsewhere and supplied as a whole to the project.

Beyond that, there is one major difference between a library and a set of object code files: packaging. Dealing with a myriad of separate object files becomes troublesome when you're working with large sets of library object files. A library may contain tens, hundreds, or even thousands of object files. Listing all these object files (or even just the ones a project uses) is a lot of work and can lead to consistency errors.

The common way to deal with this problem is to combine object files into a separate package (file) known as a *library file*. Under Linux and macOS, library files typically have a *.a* suffix (where *a* stands for *archive*). For many projects, you will be given a library file that packages together a specific library module. You supply this file to the linker when building your program, and the linker automatically picks out the object modules it needs from the library. This is an important point: including a library while building an executable does not automatically insert all the code from that library into the executable. The linker is smart enough to extract only the object files it needs and to ignore the object files it doesn't use (remember, a library is just a package containing a bunch of object files).

How do you create a library file? The short answer is, "By using the archiver program (`ar`)." Here is its basic syntax

```
ar rcs libname.a list-of-.o-files
```

where *libname.a* is the name of the library file you want to produce and *list-of-.o-files* is a (space-separated) list of object filenames you want to collect together into the library. For example, here's the command to combine the *print.o* and *getTitle.o* files into a library module (*aoaalib.a*):

```
ar rcs aoaalib.a getTitle.o print.o
```

The `rcs` component is actually a series of three command options. The `r` option tells the command to replace existing (if present) object files in the archive; `c` says to create the archive (you generally don't specify this option if you are adding object files to an existing archive file); and `s` says to add an index to the archive file, or update the index if it already exists. (For more `ar` command line options, see section 15.9, "For More Information," on page 887.)

Once you have a library module, you can specify it on a linker (or `ld` or `gcc`) command line just as you would an object file. For example, if you build a *strings.a* library module to hold the *str.cpy.o*, *str.cmp.o*, *str.substr.o*, *str.bufInit.o*, *str.free.o*, and *str.alloc.o* object files, and you want to link *strings.a* with the program in Listing 15-1, you could use the following command:

```
g++ -DisMacOS c.cpp Listing15-1.S strings.a -o Listing15-1
```

Listing 15-5 is an example of a makefile that will build the *strings.a* library file.

```
# Listing15-5.mak
#
# makefile to build the string.a library file

unamestr=`uname`

strings.a:str.cpy.o str.cmp.o str.substr.o str.bufInit.o \
            str.alloc.o str.free.o
    ar rcs strings.a str.cpy.o str.cmp.o str.substr.o \
        str.bufInit.o str.alloc.o str.free.o

str.cpy.o:aoaa.inc str.cpy.S
    g++ -c -D$(unamestr) str.cpy.S

str.cmp.o:aoaa.inc str.cmp.S
    g++ -c -D$(unamestr) str.cmp.S

str.substr.o:aoaa.inc str.substr.S
    g++ -c -D$(unamestr) str.substr.S

str.bufInit.o:aoaa.inc str.bufInit.S
    g++ -c -D$(unamestr) str.bufInit.S

str.free.o:aoaa.inc str.free.S
    g++ -c -D$(unamestr) str.free.S

str.alloc.o:aoaa.inc str.alloc.S
    g++ -c -D$(unamestr) str.alloc.S
```

```
clean:
    rm -f strings.a
    rm -f str.cpy.o
    rm -f str.cmp.o
    rm -f str.substr.o
    rm -f str.bufInit.o
    rm -f str.alloc.o
    rm -f str.free.o
```

Listing 15-6 modifies the Listing 15-5 makefile that builds the code by using the *strings.a* library module.

```
# Listing15-6.mak
#
# makefile that uses the string.a library file

unamestr=`uname`

Listing15-1:Listing15-1.o strings.a
    g++ -D$(unamestr) c.cpp Listing15-1.o strings.a -o Listing15-1

Listing15-1.o:aoaa.inc Listing15-1.S
    g++ -c -D$(unamestr) Listing15-1.S

lib:
    rm -f strings.a
    rm -f str.*.o
    make -f Listing15-5.mak

clean:
    rm -f Listing15-1
    rm -f c.o
    rm -f Listing15-1.o
```

Note that the `clean` command does not delete the library files. If you want a clean library build, just specify the `lib` command line option when running `make`:

```
make -f Listing15-6.mak lib
```

As a general rule, you build the library code independently of the application code. Most of the time, the library is prebuilt, and you don't have to rebuild it. However, *strings.a* must be a dependency of the application, because if the library changes, you'll probably need to rebuild the application as well.

One more Unix utility is useful for processing library files: `nm` (names). The `nm` utility will list all the global names found in a library module. For example, the command

```
nm strings.a
```

lists all the (global) symbols found in the *strings.a* library file (it's rather long, so I won't provide the printout here).

15.7 Managing the Impact of Object Files on Program Size

The basic unit of linkage in a program is the object file. When combining object files to form an executable, the linker will take all the data from a single object file and merge it into the final executable. This is true even if the main program doesn't call all the functions (directly or indirectly) in the object module or use all the data in that object file. If you put 100 routines in a single assembly language source file and compile them into an object module, the linker will therefore include the code for all 100 routines in your final executable, even if you use only one of them.

To avoid this situation, you can break those 100 routines into 100 separate object modules and combine the resulting 100 object files into a single library. When the linker processes that library file, it will pick out the single object file containing the function the program uses and incorporate only that file into the final executable.

Generally, this is far more efficient than linking in a single object file with 100 functions buried in it. However, in some cases, there are good reasons to combine multiple functions into a single object file. First, consider what happens when the linker merges an object file into an executable. To ensure proper alignment, whenever the linker takes a section/segment (for example, the `.code` section) from an object file, it adds sufficient padding so that the data in that section is aligned on that section's specified alignment boundary. Most sections have a default 16-byte section alignment. This means that the linker will align each section from the object file it links on a 16-byte boundary.

Normally, this isn't much of a problem, especially if your procedures are large. However, if those 100 procedures are all really short (a few bytes each), you wind up wasting a lot of space. Granted, on modern machines, a few hundred bytes of wasted space isn't a big deal. Still, it might be more practical to combine several of these procedures into a single object module (even if you don't call all of them) to fill in some of the wasted space. Look for elements that are naturally paired or otherwise used together or have a dependency, such as `alloc` and `free`. Don't go overboard, though. Once you've gone beyond the alignment, whether you're wasting space because of padding or wasting space because you're including code that never gets called, you're still wasting space.

15.8 Moving On

If you write large applications in assembly language, you'll want to break the source code into various modules and automate building the application from those modules. This chapter began by discussing Gas's mechanisms for sharing external and public symbols between modules. It then

introduced the make application for building applications from multiple source files, then covered how to build library modules by using the linker and archiver applications.

One large source of library code is the OS kernel (macOS, Linux, or Pi OS). However, don't link OS library functions into your applications; that code is already present in memory when your application runs. To call an OS function, you use an OS API invocation sequence. The next chapter discusses how to call OS functions in the Linux (Pi OS) and macOS (Darwin) kernels.

15.9 For More Information

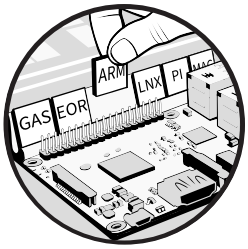
- For information about makefiles, check out the following websites:
  - Computer Hope: *https://www.computerhope.com/unix/umake.htm*
  - GNU make: *https://www.gnu.org/software/make/*
  - Wikipedia: *https://en.wikipedia.org/wiki/Make_(software)*
- Also check out the following books on make:
  - Robert Mecklenburg, *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*, 3rd edition (O'Reilly Media, 2004). You can also access this book online at *https://www.oreilly.com/openbook/make3/book/index.csp*.
  - John Graham-Cumming, *The GNU Make Book,* 1st edition (No Starch Press, 2015)
  - Andrew Oram and Steve Talbott, *Managing Projects with Make* (O'Reilly & Associates, 2004)
- See *https://man7.org/linux/man-pages/man1/ar.1.html* for a complete list of the ar command line options. You can also enter **ar --help** or **man ar** for online help.

TEST YOURSELF

1. What statement(s) would you use to prevent recursive include files?
2. What is an assembly unit?
3. What directive would you use to tell Gas that a symbol is global and visible outside the current source file?
4. What directive(s) would you use to tell Gas to use a global symbol from another object module?
5. What is the basic makefile syntax?
6. What is a makefile-dependent file?
7. What does a makefile clean command typically do?
8. What is a library file?

16

STAND-ALONE ASSEMBLY LANGUAGE PROGRAMS



Until now, this book has relied upon a C/C++ main program to call the example code written in assembly language. While this is probably the biggest use of assembly language in the real world, it is also possible to write stand-alone code (no C/C++ main program) in assembly language. In this chapter, you'll learn how to write such stand-alone programs.

For the purposes of this book, *stand-alone assembly language program* means that the assembly language code contains an actual *main* program (not `asmMain`, which is just a function that a C++ program calls). Such a program does not make any C/C++ stdlib calls; the only external calls to code outside the application itself are OS API function calls.

NOTE *Some readers might take the term* stand-alone *to mean that an assembly language program makes no external function calls, not even to an OS, and handles all I/O at the hardware level within the application itself. That's an appropriate definition for embedded systems, but not the definition I use in this book.*

Technically, your assembly code will always be called by a C/C++ program. That's because the OS itself is written in C/C++, with a tiny bit of assembly code. When the OS transfers control to your assembly code, this is not much different from a C/C++ main program calling your assembly code. Nevertheless, "pure" assembly applications have some clear advantages: you're not dragging along the C/C++ library code and application runtime system, so your programs can be smaller and you won't have external naming conflicts with C/C++ public names.

This chapter covers OS system calls for macOS and Linux (including Pi OS). It begins by explaining how to maintain portability in your code, given that system calls are not portable between OSes. It then introduces the concept of system calls for these two OSes. After discussing the `svc` (supervisor call) instruction used to make calls to OS API functions, it provides two examples: a stand-alone "Hello, world!" application and a file I/O application. Finally, it points out that macOS frowns on direct system calls and expects you to interface to the OS via C library function calls.

16.1 Portability Issues with System Calls

While most of the example programs appearing in this book so far are portable between macOS and Linux, system API calls vary by OS. Code in the previous chapters ignored this issue by calling C/C++ stdlib functions that handled the low-level OS details, but the example code in this chapter makes OS-specific calls. Therefore, portability won't happen automatically. You have four options for handling this issue:

- Ignore portability and write a given example program only for macOS or only for Linux. In general, I take this approach when writing code specific to an OS.
- Write two (nonportable) versions of the same program: one for Linux and one for macOS.
- Write a single program that uses conditional assembly to include OS-specific code, as necessary.
- Create two wrapper files, one that has macOS versions of OS calls and another that has Linux version, and include the appropriate wrapper with your main (portable) code.

The appropriate mechanism to use depends on your application. If you are not interested in writing portable assembly code that will work across OSes (the most common case when writing assembly applications), you'll use the first approach and write code just for the OS you are targeting.

If you do want your assembly application to run on macOS and Linux, your approach will depend on the size of the application. If the application is relatively small, writing two OS-specific variants is not that difficult (though maintenance may be an issue, as you'll have to maintain two separate versions of the application). If the application is large, or you expect to

upgrade and maintain it frequently, the third or fourth approach is probably better. A single application that uses conditional assembly to deal with OS-specific issues is usually much easier to maintain and expand than two separate applications, and using wrapper code makes it easier to maintain the code for each specific OS.

There is a fifth approach: write all your OS-dependent code in C/C++ and call assembly functions that deal with the non-OS-specific functionality. That's how all the example programs in this book have been written.

It should go without saying that the code in this chapter does not use the *build* script to compile/assemble the example applications. The *build* script assumes the use of the *c.cpp* main program (and the whole point of this chapter is to stop using that code). Therefore, each example program in this chapter includes a makefile that builds the code.

16.2 Stand-Alone Code and System Calls

The first example program in this book, Listing 1-1 on page 5, is a stand-alone program. Here it is as Listing 16-1, with a couple of changes, for the sake of discussion.

```
// Listing16-1.S
//
// Comments consist of all text from a //
// sequence to the end of the line.
// The .text directive tells MASM that the
// statements following this directive go in
// the section of memory reserved for machine
// instructions (code).

        .text

// Here is the main function.
// (This example assumes that the
// assembly language program is a
// stand-alone program with its own
// main function.)
//
// Under macOS, the main program
// must have the name _main
// beginning with an underscore.
// Linux systems generally don't
// require the underscore.
//
// The .global _main statement
// makes the _main procedure's name
// visible outside this source file
// (needed by the linker to produce
// an executable).
```

```
        ❶ .global _main  // This is the macOS entry point.
        ❷ .global main   // This is the Linux entry point name.

// The .align 2 statement tells the
// assembler to align the following code
// on a 4-byte boundary (required by the
// ARM CPU). The 2 operand specifies
// 2 raised to this power (2), which
// is 4.

        .align 2

// Here's the actual main program. It
// consists of a single ret (return)
// instruction that simply returns
// control to the operating system.

❶ _main:
❷ main:
      ❸ ret
```

I've made two changes to this code compared with that in Listing 1-1. At both instances of ❶ and ❷, I've introduced a new symbol, `main` (and `_main`). This is because Linux requires the main program to be named `main`, whereas macOS requires the name `_main`. Were you to attempt to compile Listing 1-1 under Linux, you would get something like an `undefined reference to` `` `main` `` message. Rather than mess around with conditional assembly (or write two separate versions of Listing 16-1), I simply include both symbols in the source file. Linux largely ignores the `_main` symbol, and macOS ignores the `main` symbol; the program happily compiles under either OS.

Listing 16-1 consists of a single instruction: `ret` ❸. On entry, the LR register contains a return address that transfers control back to the OS. Therefore, this program (should you actually execute it) returns immediately to the OS.

Although returning to the OS via a `ret` instruction works (particularly if building this code with GCC), this isn't the standard way to return to Linux or macOS. Instead, an application should make a call to the `exit()` API function. To call a system API function, a program must load a function number into a register, load appropriate parameters into the parameter registers (X0 through X7), and then execute the supervisor (OS) call instruction `svc #OSint`, where *`OSint`* is 0 for Linux and 0x80 for macOS.

NOTE *In reality, macOS seems to ignore the immediate constant following the* `svc` *instruction. Many online examples use the value 0 as the* `svc` *operand (and personal experiments show that it works). However, the macOS source code seems to use 0x80 as the constant, so I recommend using this value under macOS.*

Under Linux, you load the system call number into the X8 register, while under macOS you load this into X16. I've added the following statements in *aoaa.inc* to handle this:

```
#if isMacOS

    // Under macOS, the system call number
    // goes into X16:

    #define svcReg x16
    #define OSint  0x80

#else

    // Under Linux, the system call number
    // is passed in X8:

    #define svcReg x8
    #define OSint  0

#endif
```

Under both Linux and macOS, the `exit` function expects a single integer parameter in the X0 register holding the program's return code (generally this is 0 if no errors occurred while running the program). The only question remaining is, "What is the system call number for `exit()`?" Under Linux, the code is 93, while under macOS it's 1 (I'll discuss how I determined these magic numbers in section 16.3, "The `svc` Interface and OS Portability," on the next page). Listing 16-2 provides a very simple assembly application that immediately returns to the OS and that you can compile for macOS or Linux.

```
// Listing16-2.S
//
// Simple shell program that calls exit()

      ❶ #include    "aoaa.inc"

        // Specify OS-dependent return code:

      ❷ #ifdef      isMacOS
        #define     exitCode 1
        #else
        #define     exitCode 93
        #endif

        .text
        .global     _main
        .global     main
        .align      2
```

```
_main:
main:
      ❸ mov         x0, #0  // Return success.
      ❹ mov         svcReg, exitCode
      ❺ svc         #0Sint
```

Listing 16-2 includes *aoaa.inc* ❶ in order to generate an error if the OS symbol (Linux or Darwin) is not defined on the command line (which *aoaa.inc* translates to `isLinux` or `isMacOS`), as well as to obtain the `0Sint` and `svcReg` constants.

The program uses conditional assembly to generate the different code needed for macOS or Linux, setting the constant `exitCode` equal to the OS's exit function number ❷. This function loads X0 with 0 ❸ to indicate success when it returns. It then loads the `exitCode` function number into the OS's function number parameter register ❹ (`svcReg` is X8 under Linux and X16 under macOS, as defined in *aoaa.inc*, as per the previous example). Finally, the code issues the supervisor call instruction to call the OS ❺. Because of the nature of this call, the `svc` instruction never returns control back to the program, so there is no need for a `ret` instruction.

Here's the makefile to build the program in Listing 16-2:

```
# Listing16-2.mak
#
# makefile to build the Listing16-2 file

unamestr=`uname`

Listing16-2:
    g++ -D$(unamestr) Listing16-2.S -o Listing16-2

clean:
    rm -f Listing16-2.o
    rm -f Listing16-2
```

To build and run this program, enter the following commands into the shell program:

```
% make -f Listing16-2.mak
g++ -D`uname` Listing16-2.S -o Listing16-2
% ./Listing16-2
```

The program returns without producing any output, as expected.

16.3 The svc Interface and OS Portability

Both macOS and Linux use the supervisor call instruction (`svc`) to make API calls to the OS. However, the exact calling sequence varies considerably between the two OSes. This section clarifies the differences between them with respect to the functions (the API) that they support—in particular, regarding call numbers, parameters, and error handling.

Although both OSes are Unix based (and share many POSIX-compliant functions), each has its own set of OS-specific functions that may have no equivalence on the other system. Even the common (for example, POSIX) functions may expect different arguments and produce different return results, meaning you must be especially careful when attempting to write assembly code that is portable between these two OSes. This is a good example of how using wrappers to localize OS system calls can help improve the portability and maintainability of your code.

16.3.1 Call Numbers

As noted earlier, the function call number values differ between OSes, as does the location where you pass the call numbers (X8 for Linux, X16 for macOS). It's easy enough to overcome the register location issue by using a `#define` (or `.req` directive). However, the function call number values are completely OS dependent.

The *sys/syscall.h* file is a header file that contains definitions for all the system API call numbers. (You can include it in an assembly language source file even though it's a C header file.) This file is generally installed on your system when you install your C compiler (GCC or Clang) and is typically found in the default include path used by the compiler. See the GCC or Xcode documentation for more details.

Although `#include <sys/syscall.h>` will work on both Linux and macOS, the actual definitions may appear in a different file elsewhere in the compiler's directory tree, with an appropriate `#include` inside *sys/syscall.h* leading to the actual file.

Here are a few lines from *sys/syscall.h* on a macOS machine:

```
#ifdef __APPLE_API_PRIVATE
#define SYS_syscall        0
#define SYS_exit           1
#define SYS_fork           2
#define SYS_read           3
#define SYS_write          4
#define SYS_open           5
#define SYS_close          6
#define SYS_wait4          7
   .
   .
   .
```

The `#ifdef` statement in this code is a warning that Apple considers the `svc` API interface to be undocumented and private, as discussed in the "Using `svc` Under macOS" box on the next page.

On my macOS system, I used the Unix `find` command to locate *sys/syscall.h* buried deep in the Xcode directory path */Library/Developer/CommandLineTools/SDK/* . . . , but your mileage may vary.

USING SVC UNDER MACOS

While using the svc interface to the OS is perfectly acceptable under Linux, using svc to call system functions is considered an undocumented or private API call under macOS. Apple does not guarantee that such calls will continue to work in future versions of macOS. Furthermore, if you attempt to sell an application in the App Store that contains such calls, Apple may reject your application because it uses undocumented APIs.

According to Apple, the correct way to make such API calls is through the use of system libraries. For example, you should call the `_exit()` function rather than loading 1 into X16 and executing svc. I'll have more to say about this subject in section 16.6, "Calling System Library Functions Under macOS," on page 926).

Under Debian Linux, `#include <sys/syscall.h>` includes */usr/include/asm-generic/unistd.h* (again, use the Unix `find` command if this file isn't present at this location). Here are a few lines from that file, sorted to match the order of the statements in the macOS *syscall.h* file:

```
#define __NR_exit 93
__SYSCALL(__NR_exit, sys_exit)
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
#define __NR_openat 56
__SYSCALL(__NR_openat, sys_openat)
#define __NR_close 57
__SYSCALL(__NR_close, sys_close)
   .
   .
   .
```

As you can see, the function names, the constant names, and the function call numbers in the two files don't agree. For example, Linux generally prefers `openat()` over the `open()` function. Fortunately, macOS also provides `openat()`, so it's possible to use the same function on both OSes. However, the symbolic names macOS and Linux use for the same functions are quite different, which means including *sys/syscall.h* isn't a portable solution. You'll still have to provide your own local names that map to the corresponding Linux and macOS names (suggestion: use two syscall wrapper *.S* files, one for Mac and one for Linux, to resolve these problems).

Unfortunately, the *sys/syscall.h* header files don't provide the parameter lists for the various functions. You can find the parameter information for Linux at *https://arm.syscall.sh*. For example, consider the entry for the `exit()` function:

```
name  reference   x8    x0              x1 x2 x3 x4 x5
exit  man/ cs/    5D    int error_code  -- -- -- -- --
```

This line tells you that X8 must contain 0x5D (93) and X0 must contain the exit code (`error_code`). Linux system calls have a maximum of six parameters (X0 through X5), but `exit()` uses only one of them (in addition to the function call number in X8).

On macOS, you must use the macOS call number (1 for exit) and load that call number into X16. The parameters are generally the same for equivalent macOS functions, subject (of course) to the differences in the Linux versus macOS ABI (see section 16.8, "For More Information," on page 930 for the set of system calls).

16.3.2 API Parameters

As a general rule, all the system calls have well-defined names and parameter lists, which you can find online by searching for the function's name or by using the `man` command at the command line. For example, the `openat()` call has the following parameters (from the Linux man page):

```
int openat(int dfd, const char *pathname, int flags, mode_t mode);
```

The `openat` code (57 in Linux, 463 under macOS) goes in X8 (Linux) or X16 (macOS), a directory descriptor goes in X0, a pointer to the filename (pathname) goes in X1, and the flags get passed in X2 (an optional `mode` parameter can be passed in X3). You choose the parameter registers according to the ARM ABI.

To create code that's portable between macOS and Linux, you can use the following conditional assembly at the beginning of your source file to select the constants based on the OS:

```
    #include "aoaa.inc"
    #include <sys/syscall.h>

#if isMacOS

    #define  sys_openat SYS_openat
    #define  sys_read SYS_read
    #define  sys_close SYS_close

#elif isLinux

    #define  sys_openat __NR_openat
    #define  sys_read __NR_read
    #define  sys_close __NR_close

#endif
```

From this point forward, you can use the symbols `sys_*` on either OS. Of course, if you don't require portability between the two OSes, you can

simply include *sys/syscall.h* and use the `SYS_*` or `_NR_*` symbols as appropriate for your OS choice.

16.3.3 API Error Handling

One other big difference between macOS and Linux API calls is in the way they return an indication of an error. For macOS API calls, the error status is returned in the carry flag (C = 1 for error, C = 0 for no error). If the carry flag is set, macOS returns an error code in the X0 register. Linux, on the other hand, returns –1 in X0 if there is an error; you must then fetch the actual error code from the `errno` variable (as shown in Listing 7-2 on page 358, for example).

Dealing with error return values in portable code could be problematic. One solution is to use a set of wrapper functions to handle errors in an OS-specific fashion for each OS. I've chosen to create a small macro to translate the error return status to a common value on both macOS and Linux:

```
            .macro  checkError

            #if     isMacOS

            // If macOS, convert the error code to be
            // compatible with Linux (carry set is
            // error flag and X0 is error code):

          ❶ bcc     0f
          ❷ neg     x0, x0

            #elif   isLinux

            // If Linux, fetch the errno error code
            // (if return value is -1), negate it,
            // and return that as the error code:

          ❸ cmp     x0, #-1
            bne     0f
          ❹ getErrno
            neg     x0, x0

            #endif
0:
            .endm
```

Under macOS, if the carry is clear ❶, then this macro does nothing (no error). If there is an error (carry set), the macro negates the value in X0 ❷ (which is currently a positive error code).

Under Linux, an error is indicated by returning –1 in X0, in which case the code has to retrieve the actual error code from the `errno` variable. If the API function returns –1 ❸, the code fetches `errno`'s value ❹ (which is a positive number) and negates it.

This macro assumes that the API function will return a nonnegative result in X0 if there is no error, so it returns a negation of the actual error code if there was an error. This will provide a sign-consistent set of values you can test for under either OS.

Although the `checkError` macro produces a portable set of error codes, do not assume that the two OSes will produce exactly the same error codes for any given situation. They are more likely to produce slightly different error codes under the same circumstances. At the very least, you should be able to handle any error return code that either the macOS or Linux man pages list for a given API function (yet another argument for using a wrapper function to handle error codes in portable code).

You can lift the appropriate defines from the *errno.h* file (or other files it may include); this will allow you to refer to Unix-compatible constant names like `EPERM` or `EACCES` in your assembly source code. Don't forget that the `checkError` macro negates the error code, so you have to compare against negated *errno.h* constants (for example, `-EPERM` or `-EACCES`).

16.4 A Stand-Alone "Hello, World!" Program

By convention, the first "real" program to try when writing a stand-alone assembly program is "Hello, world!" Under Linux and macOS, you can use the system `write()` function to write a string to the standard output device, as shown in Listing 16-3.

```
// Listing16-3.S
//
// A stand-alone "Hello, world!" program

        #include    "aoaa.inc"
      ❶ #include    <sys/syscall.h>

        // Specify OS-dependent return code:

        #if         isMacOS

      ❷ #define     exitCode    SYS_exit
        #define     sys_write   SYS_write

        #else

      ❸ #define     exitCode    __NR_exit
        #define     sys_write   __NR_write

        #endif

        .data
hwStr:  .asciz      "Hello, world!\n"
hwSize  =           .-hwStr
```

```
        .text
        .global     _main
        .global     main
        .align      2

_main:
main:

      ❹ mov         x0, #1          // stdout file handle
        lea         x1, hwStr       // String to print
        mov         x2, #hwSize     // Num chars to print
        mov         svcReg, #sys_write
        svc         #0Sint          // Call OS to print str.

      ❺ mov         svcReg, #exitCode
        mov         x0, #0
        svc         #0Sint          // Quit program.
```

In Listing 16-3, the `#include` statement loads in the operating-specific constant names for the API function call values ❶. The code defines the system call constants for macOS ❷. As the `write` symbol was already defined in the *aoaa.inc* header file (the name of the C stdlib `write()` function), I've used `sys_write` to overcome namespace pollution. Likewise, the code defines the Linux system call constants ❸.

Calling the system API `write()` function prints `Hello, world!` ❹. This call expects a pointer to the string in X1, the string length in X2, and a file descriptor value in X0. For the stdout device, the file descriptor is 1. Finally, I include the usual program termination code ❺.

Note that the C stdlib `write()` function is nothing more than facade code that directly calls the Linux `write()` API function. If we were willing to link with C code, we could have accomplished the same thing by calling `write()`, but doing so would defeat the purpose of this chapter.

Here's the makefile that will build the program in Listing 16-3:

```
# Listing16-3.mak
#
# makefile to build the Listing16-3 file

unamestr=`uname`

Listing16-3:
    g++ -D$(unamestr) Listing16-3.S -o Listing16-3

clean:
    rm -f Listing16-3.o
    rm -f Listing16-3
```

Here are the commands to build and run the program, along with the sample output:

```
% make -f Listing16-3.mak clean
rm -f Listing16-3.o
rm -f Listing16-3
% make -f Listing16-3.mak
g++ -D`uname` Listing16-3.S -o Listing16-3
% ./Listing16-3
Hello, world!
```

Note that the program did not print `Calling Listing16-3` nor `Listing16-3 terminated`. That output is produced by the *c.cpp* `main()` function, which this code is not using.

16.5 A Sample File I/O Program

File I/O has been conspicuously absent from the book thus far. While reading and writing file data is easily achieved using the C stdlib functions such as `fopen`, `fclose`, and `fprintf`, the Linux and macOS APIs provide many useful functions (on which the C stdlib is built) for this purpose. This section describes a few of these functions:

`open` Open (or create) a file for reading, writing, or appending.

`read` Read data from an open file.

`write` Write data to an open file.

`close` Close an open file.

For the purposes of this example, I'll implement these calls as a *files* library consisting of three source modules:

volatile.S A pair of utility functions that save and restore all the volatile registers

stdio.S A set of I/O routines that write data to the stdout device and read data from the stdin device (console I/O)

files.S A set of routines for opening, reading, writing, and closing files

I've put these files in a *files* subdirectory, along with a *files.mak* makefile that will assemble these files and put them in a *file.a* archive file. Here's the makefile:

```
# files.mak
#
# makefile to build the files library

unamestr=`uname`

files.a:files.o stdio.o volatile.o
    ar rcs files.a files.o stdio.o volatile.o
  ❶ cp files.a ..

files.o:files.S files.inc ../aoaa.inc
    g++ -c -D$(unamestr) files.S
```

```
stdio.o:stdio.S files.inc ../aoaa.inc
    g++ -c -D$(unamestr) stdio.S

volatile.o:volatile.S files.inc ../aoaa.inc
    g++ -c -D$(unamestr) volatile.S

clean:
    rm -f files.o
    rm -f volatile.o
    rm -f stdio.o
    rm -f files.a
```

After this makefile successfully builds the source files (and combines them into the *file.a* archive file) ❶, it copies *file.a* into the parent directory, where the application that uses *files.a* appears.

Before discussing the file library's source files, I'll present the *files.inc* header file, since it contains definitions that both the library and application source code will use:

```
// files.inc
//
// Header file that holds the files library
// globals and constants

            #include "../aoaa.inc"  // Get isMacOS and isLinux.
❶ #if isMacOS
#define __APPLE_API_PRIVATE
#endif
            #include        <sys/syscall.h>

            #if     isMacOS

❷ sys_Read    =       SYS_read
sys_Write   =       SYS_write
sys_Open    =       SYS_openat
sys_Close   =       SYS_close
AT_FDCWD    =       -2

#define O_CREAT     00000200

            #else

❸ sys_Read    =       __NR_read
sys_Write   =       __NR_write
sys_Open    =       __NR_openat
sys_Close   =       __NR_close
AT_FDCWD    =       -100

#define O_CREAT     00000100

            #endif

// Handles for the stdio files:
```

```
❹ stdin       =       0
  stdout      =       1
  stderr      =       2

  // Other useful constants:

  cr          =       0xd     // Carriage return (ENTER)
  lf          =       0xa     // Line feed/newline char
  bs          =       0x8     // Backspace

  // Note the following are octal (base 8) constants!
  // (Leading 0 indicates octal in Gas.)
  //
  // These constants were copied from fcntl.h.

❺ #define S_IRWXU  (00700)
  #define S_RDWR   (00666)
  #define S_IRUSR  (00400)
  #define S_IWUSR  (00200)
  #define S_IXUSR  (00100)
  #define S_IRWXG  (00070)
  #define S_IRGRP  (00040)
  #define S_IWGRP  (00020)
  #define S_IXGRP  (00010)
  #define S_IRWXO  (00007)
  #define S_IROTH  (00004)
  #define S_IWOTH  (00002)
  #define S_IXOTH  (00001)
  #define S_ISUID  (0004000)
  #define S_ISGID  (0002000)
  #define S_ISVTX  (0001000)

  #define O_RDONLY    00000000
  #define O_WRONLY    00000001
  #define O_RDWR      00000002
  #define O_EXCL      00000200
  #define O_NOCTTY    00000400
  #define O_TRUNC     00001000
  #define O_APPEND    00002000
  #define O_NONBLOCK  00004000
  #define O_DSYNC     00010000
  #define FASYNC      00020000
  #define O_DIRECT    00040000
  #define O_LARGEFILE 00100000
  #define O_DIRECTORY 00200000
  #define O_NOFOLLOW  00400000
  #define O_NOATIME   01000000
  #define O_CLOEXEC   02000000

  // Macro to test an error return
  // value from an OS API call:

          ❻ .macro  file.checkError

              #if     isMacOS
```

```
            // If macOS, convert the error code to be
            // compatible with Linux (carry set is
            // error flag, and X0 is error code):

            bcc     0f
            neg     x0, x0

            #elif   isLinux

            // If Linux, fetch the errno error code
            // (if return value is -1), negate it,
            // and return that as the error code:

            cmp     x0, #-1
            bne     0f
            getErrno
            neg     x0, x0

            #endif
0:
            .endm

          ❼ .extern saveVolatile
            .extern restoreVolatile

            .extern file.write
            .extern file.read
            .extern file.open
            .extern file.openNew
            .extern file.close

            .extern stdout.puts
            .extern stdout.newLn

            .extern stdin.read
            .extern stdin.getc
            .extern stdin.readln
```

As mentioned earlier, the macOS `SYS_*` symbols appear inside an `#ifdef` block that hides the definitions if the symbol `__APPLE_API_PRIVATE` is not defined. Therefore, when including the `sys/syscall.h` header file under macOS, *files.inc* needs to define the symbol `__APPLE_API_PRIVATE` so that all the `SYS_*` labels will be processed by the CPP ❶.

The *files.inc* header then defines various symbols whose values differ by OS (in particular, the API function call numbers) ❷ ❸. This conditional assembly block also defines the `O_CREAT` symbol, which is different for the two OSes.

Next, the header defines various constants that will be useful in both the library source code and in applications that link against the library ❹. The `stdin`, `stdout`, and `stderr` constants are the Unix file descriptor values for the standard input device, the standard output device, and the standard

error (output) device, respectively. The library uses `cr`, `lf`, and `bs` as ASCII character code constants.

I've then inserted several `#define` statements lifted from *fcntl.h* ❺ (yet another C/C++ header file containing useful API constant definitions; you'll usually find it in the same directory as *syscall.h*). These constants are used with the `openat()` function when creating a new file (you supply these constants for the `mode` parameter). As with *errno.h*, you cannot simply include *fcntl.h* because Gas will not be able to process the C/C++ statements that appear in it.

As previously discussed, the library uses the `file.checkError` macro ❻ after `svc` instructions to check error return results. Finally, the code includes external definitions for all the functions that appear in the *files.a* library ❼.

16.5.1 volatiles.S Functions

The *volatiles.S* source file contains two functions that save and restore all the volatile registers, `saveVolatile` and `restoreVolatile`:

```
// volatiles.S
//
// saveVolatile and restoreVolatile functions used
// to preserve volatile registers

            #include    "../aoaa.inc"
            #include    "files.inc"

            .code
            .align  2

// saveVolatile
//
// A procedure that will save all the volatile
// registers at the location pointed at by FP

            proc    saveVolatile, public
            stp     x0,  x1,  [fp], #16
            stp     x2,  x3,  [fp], #16
            stp     x4,  x5,  [fp], #16
            stp     x6,  x7,  [fp], #16
            stp     x8,  x9,  [fp], #16
            stp     x10, x11, [fp], #16
            stp     x12, x13, [fp], #16
            stp     x14, x15, [fp], #16
            stp     q0,  q1,  [fp], #32
            stp     q2,  q3,  [fp], #32
            stp     q4,  q5,  [fp], #32
            stp     q6,  q7,  [fp], #32
            stp     q8,  q9,  [fp], #32
            stp     q10, q11, [fp], #32
            stp     q12, q13, [fp], #32
            stp     q14, q15, [fp], #32
```

```
            ret
            endp    saveVolatile

// restoreVolatile
//
// A procedure that will restore all the volatile
// registers from the location pointed at by FP

            proc    restoreVolatile, public
            ldp     x0,  x1,  [fp], #16
            ldp     x2,  x3,  [fp], #16
            ldp     x4,  x5,  [fp], #16
            ldp     x6,  x7,  [fp], #16
            ldp     x8,  x9,  [fp], #16
            ldp     x10, x11, [fp], #16
            ldp     x12, x13, [fp], #16
            ldp     x14, x15, [fp], #16
            ldp     q0,  q1,  [fp], #32
            ldp     q2,  q3,  [fp], #32
            ldp     q4,  q5,  [fp], #32
            ldp     q6,  q7,  [fp], #32
            ldp     q8,  q9,  [fp], #32
            ldp     q10, q11, [fp], #32
            ldp     q12, q13, [fp], #32
            ldp     q14, q15, [fp], #32
            ret
            endp    restoreVolatile
```

These functions simply store the registers into successive locations at the address held in the FP register. It is the caller's responsibility to preserve FP and load it with the address of the `volatile_save` structure before calling `saveVolatile` or `restoreVolatile`. As you can see in *volatiles.S*, this code does not preserve the value in the FP register.

The purpose behind `saveVolatile` and `restoreVolatile` is to overcome the fact that the OS API calls can modify the volatile register set. It's good assembly language programming style to always preserve register values unless you are explicitly returning a result in a register. The *volatiles.S* functions allow you to conform to this style even when calling low-level API functions that trash the volatile registers.

The one downside of these functions is that you never know which volatile registers a given API function might modify, so you have to preserve them all, even if the API function changes only a few. This, sadly, introduces inefficiency into the code; reading and writing memory is not especially fast. However, not having to worry about volatile registers in your assembly language code is worth the slight efficiency loss. (Moreover, file I/O is usually a relatively slow process to begin with, so if you're frequently calling file I/O functions, saving and restoring the registers is probably a very small percentage of the running time.)

The *aoaa.inc* header file contains the following structure to define the layout of the registers saved by `saveVolatile` and loaded by `restoreVolatile`:

```
struct  volatile_save
qword   volatile_save.x0x1
qword   volatile_save.x2x3
qword   volatile_save.x4x5
qword   volatile_save.x6x7
qword   volatile_save.x8x9
qword   volatile_save.x10x11
qword   volatile_save.x12x13
qword   volatile_save.x14x15
qword   volatile_save.v0
qword   volatile_save.v1
qword   volatile_save.v2
qword   volatile_save.v3
qword   volatile_save.v4
qword   volatile_save.v5
qword   volatile_save.v6
qword   volatile_save.v7
qword   volatile_save.v8
qword   volatile_save.v9
qword   volatile_save.v10
qword   volatile_save.v11
qword   volatile_save.v12
qword   volatile_save.v13
qword   volatile_save.v14
qword   volatile_save.v15
ends    volatile_save
```

Because this structure is rather large, `saveVolatile` and `restoreVolatile` do not refer to the individual fields. The offsets to certain members of this structure are too large to encode in the addressing mode offset field of a 32-bit load instruction. Nevertheless, these structures do document where `saveVolatile` and `restoreVolatile` place the data.

16.5.2 files.S File I/O Functions

The *files.S* source file contains the file I/O functions in the library. Because this file is rather long, I'll break it into pieces and discuss each section in turn. (I won't include the parameter values you pass to these functions; these are well documented online, or you can use the Unix `man` command for the `read()`, `write()`, `open()`, `openat()`, and `close()` functions.)

Most of the *files.S* functions are facade code—that is, they exist to change the environment or parameters of another function (in this case, the OS API functions). These functions preserve volatile registers so that the caller doesn't have to worry about their preservation; in a few cases (open calls), they automatically set up certain default parameters for the caller; or, in the event of an error, they modify the return codes to produce a consistent result across OSes. The `file.write` function demonstrates

providing a uniform interface (across OSes), preserving the volatile registers, and returning a consistent error code:

```
// files.S
//
// File I/O functions:

            #include    "../aoaa.inc"
            #include    "files.inc"

            .code
            .align  2

// file.write
//
// Write data to a file handle.
//
// X0- File handle
// X1- Pointer to buffer to write
// X2- Length of buffer to write
//
// Returns:
//
// X0- Number of bytes actually written
//     or -1 if there was an error

            proc    file.write, public

            locals  fw_locals
            qword   fw_locals.saveX0
          ❶ byte    fw_locals.volSave, volatile_save.size
            byte    fw_locals.stkspace, 64
          ❷ dword   fw_locals.fpSave
            endl    fw_locals

            enter   fw_locals.size

            // Preserve all the volatile registers because
            // the OS API write function might modify them.
            //
            // Note: Because fw_locals.volSave is at the
            // bottom of the activation record, SP just
            // happens to be pointing at it right now.
            // Use it to temporarily save FP so you can
            // pass the address of fw_locals.volSave to
            // saveVolatile in the FP register.

          ❸ str     fp, [sp]    // fw_locals.fpSave
            add     fp, fp, #fw_locals.volSave
            bl      saveVolatile
            ldr     fp, [sp]    // Restore FP.

            // Okay, now do the write operation (note that
            // the sys_Write arguments are already sitting
```

```
    // in X0, X1, and X2 upon entry into this
    // function):

❹ mov     svcReg, #sys_Write
    svc     #OSint

    // Check for error return code:

❺ file.checkError

    // Restore the volatile registers, except
    // X0 (because we return the function
    // result in X0):

❻ str     x0, [fp, #fw_locals.saveX0] // Return value.
    str     fp, [sp]    // fw_locals.fpSave
    add     fp, fp, #fw_locals.volSave
    bl      restoreVolatile
    ldr     fp, [sp]    // Restore FP.
    ldr     x0, [fp, #fw_locals.saveX0]
    leave
    endp    file.write
```

In the activation record, `file.write` reserves space for the volatile register save area ❶ and the special variable `fw_locals.fpSave` ❷. The code will use this variable to preserve the FP register across calls to `saveVolatile` and `restoreVolatile`. Note that `fw_locals.fpSave` appears last in the activation record, so it will be located on the top of the stack when `file.write` builds the activation record. This is a temporary variable that will not be used when system calls use the space on the top of the stack (assuming they do).

Next, `file.write` saves all the volatile registers to the volatile save area (`fw_locals.volSave`) ❸. Because `saveVolatile` expects FP pointing at the save area, this code saves FP to the top of the stack (which just happens to be the location of the `fw_locals.fpSave` variable), loads FP with the address of the `fw_locals.volSave` structure, calls `saveVolatile`, and then restores FP upon return.

Note that the code could not reference the `fw_locals.fpSave` variable by using the `[FP, #fw_locals.fpSave]` addressing mode. First, the size of the activation record is too large, and the offset of `fw_locals.fpSave` cannot be encoded into a 32-bit instruction. Second, FP is not pointing at the activation record upon return from `saveVolatile`, so the `[FP, #fw_locals.fpSave]` addressing mode would reference the wrong location (even if the offset weren't too large).

The code then actually calls the API function ❹. This code is almost trivial, because all the parameters the `write()` function requires are already in the appropriate registers, as they were passed into `file.write` in those registers.

The code checks for an error and massages the value in X0 if an error occurred during the write operation ❺. The `file.write` function restores the registers previously saved to `fw_locals.volSave` ❻. The code and

explanation are almost identical to that at ❸, with one exception: the value in X0. Because this code returns a function result in X0 and `restoreVolatile` will restore X0 to its original value, this code has to save and restore X0 across the call to `restoreVolatile`. Because the variable `fw_locals.saveX0` appears in the activation record before `fw_locals.volSave`, there is no problem with the offset when using the `[FP, #fw_locals.saveX0]` addressing mode; only the variables appearing after `fw_locals.volSave` will have offsets too large to use in the addressing mode.

The `file.read` function is almost identical to the `file.write` function:

```
// files.S (cont.)
//
// file.read
//
// Read data from a file handle.
//
// X0- File handle
// X1- Pointer to buffer receive data
// X2- Length of data to read
//
// Returns:
//
// X0- Number of bytes actually read
//     or negative value if there was an error

            proc    file.read, public

            locals  fr_locals
            qword   fr_locals.saveX0
            byte    fr_locals.volSave, volatile_save.size
            byte    fr_locals.stkspace, 64
            dword   fr_locals.fpSave
            endl    fr_locals

            enter   fr_locals.size

            // Preserve all the volatile registers because
            // the OS API read function might modify them.
            //
            // Note: Because fr_locals.volSave is at the
            // bottom of the activation record, SP just
            // happens to be pointing at it right now.
            // Use it to temporarily save FP so we can
            // pass the address of fr_locals.volSave to
            // saveVolatile in the FP register.

            str     fp, [sp]    // fr_locals.fpSave
            add     fp, fp, #fr_locals.volSave
            bl      saveVolatile
            ldr     fp, [sp]    // Restore FP.

            // Okay, now do the read operation (note that
            // the sys_Read arguments are already sitting
```

```
            // in X0, X1, and X2 upon entry into this
            // function):

            mov     svcReg, #sys_Read
            svc     #OSint

            // Check for error return code:

            file.checkError

            // Restore the volatile registers, except
            // X0 (because we return the function
            // result in X0):

            str     x0, [fp, #fr_locals.saveX0] // Return value.
            str     fp, [sp]    // fr_locals.fpSave
            add     fp, fp, #fr_locals.volSave
            bl      restoreVolatile
            ldr     fp, [sp]    // Restore FP.
            ldr     x0, [fp, #fr_locals.saveX0]
            leave
            endp    file.read
```

The only real difference between `file.read` and `file.write` is the function number loaded into the `svcReg` register.

Next, *files.S* provides the code for the `file.open` function:

```
// files.S (cont.)
//
// file.open
//
// Open existing file for reading or writing.
//
// X0- Pointer to pathname string (zero-terminated)
// X1- File access flags
//     (O_RDONLY, O_WRONLY, or O_RDWR)
//
// Returns:
//
// X0- Handle of open file (or negative value if there
//     was an error opening the file)

            proc    file.open, public

            locals  fo_locals
            qword   fo_locals.saveX0
            byte    fo_locals.volSave, volatile_save.size
            byte    fo_locals.stkspace, 64
            dword   fo_locals.fpSave
            endl    fo_locals

            enter   fo_locals.size
```

```
    // Preserve all the volatile registers because
    // the OS API open function might modify them:

    str     fp, [sp]    // fo_locals.fpSave
    add     fp, fp, #fo_locals.volSave
    bl      saveVolatile
    ldr     fp, [sp]    // Restore FP.

    // Call the OS API open function:

❶   mov     svcReg, #sys_Open
    mov     x2, x1
    mov     x1, x0
    mov     x0, #AT_FDCWD
    mov     x3, #S_RDWR     // Mode, usually ignored
    svc     #OSint

    // Check for error return code:

    file.checkError

    // Restore the volatile registers, except
    // X0 (because we return the function
    // result in X0):

    str     x0, [fp, #fo_locals.saveX0] // Return value.
    str     fp, [sp]    // fo_locals.fpSave
    add     fp, fp, #fo_locals.volSave
    bl      restoreVolatile
    ldr     fp, [sp]    // Restore FP.
    ldr     x0, [fp, #fo_locals.saveX0]
    leave
    endp    file.open
```

The `file.open` function is identical to `file.write` and `file.read`, except for the call to the OS API function ❶. Instead of calling the API `open()` function, `file.open` calls the API `openat()` function, a more modern version of `open()`. Here are the C/C++ prototypes for these two functions:

```
int open(const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags);
```

The `openat()` function has one extra parameter, `int dirfd`. This complicates matters because `file.open` expects the same parameters as the `open()` function; therefore, upon entry to `file.open`, the parameters are sitting in the wrong registers for a call to `openat()`.

This is easily fixed by moving X1 to X2 and X0 to X1, then loading X0 with the value `AT_FDCWD` to make the `openat()` function behave like `open()` ❶. The `open()` and `openat()` functions have an optional third or fourth parameter (respectively) that lets you set the permissions when creating a new file. The `file.open` function is intended for opening existing files, so you don't normally specify that extra parameter when calling it. However, just in case

the caller specifies O_CREAT in X1, this code sets X3 to a reasonable value (read and write permissions for everyone).

The file.openNew function is a variant of file.open used to create new files:

```
// files.S (cont.)
//
// file.openNew
//
// Creates a new file and opens it for writing
//
// X0- Pointer to filename string (zero-terminated)
//
// Returns:
//
// X0- Handle of open file (or negative if there
//     was an error creating the file)

            proc    file.openNew, public

            locals  fon_locals
            qword   fon_locals.saveX0
            byte    fon_locals.volSave, volatile_save.size
            byte    fon_locals.stkspace, 64
            dword   fon_locals.fpSave
            endl    fon_locals

            enter   fon_locals.size

            // Preserve all the volatile registers because
            // the OS API open function might modify them:

            str     fp, [sp]    // fon_locals.fpSave
            add     fp, fp, #fon_locals.volSave
            bl      saveVolatile
            ldr     fp, [sp]    // Restore FP.

            // Call the OS API open function:

            mov     svcReg, #sys_Open
            mov     x2, #O_CREAT+O_WRONLY+O_EXCL
            mov     x1, x0
            mov     x0, #AT_FDCWD
            mov     x3, #S_RDWR // User/Group has RW perms.
            svc     #0Sint

            // Check for error return code:

            file.checkError

            // Restore the volatile registers, except
            // X0 (because we return the function
```

```
            // result in X0):

            str     x0, [fp, #fon_locals.saveX0] // Return value.
            str     fp, [sp]    // w_locals.fpSave
            add     fp, fp, #fon_locals.volSave
            bl      restoreVolatile
            ldr     fp, [sp]    // Restore FP.
            ldr     x0, [fp, #fon_locals.saveX0]
            leave
            endp    file.openNew
```

The only difference between `file.openNew` and `file.open` is that `file.openNew` expects just a single parameter (the pathname in X0) and automatically supplies the flag values (`O_CREAT+O_WRONLY+O_EXCL`) for the call to `openat()`.

The `file.close` function is the final file I/O function in the *files.S* source file:

```
// files.S (cont.)
//
// file.close
//
// Closes a file specified by a file handle
//
// X0- Handle of file to close

            proc    file.close, public

            locals  fc_locals
            qword   fc_locals.saveX0
            byte    fc_locals.volSave, volatile_save.size
            byte    fc_locals.stkspace, 64
            dword   fc_locals.fpSave
            endl    fc_locals

            enter   fc_locals.size

            // Preserve all the volatile registers because
            // the OS API open function might modify them:

            str     fp, [sp]    // fc_locals.fpSave
            add     fp, fp, #fc_locals.volSave
            bl      saveVolatile
            ldr     fp, [sp]    // Restore FP.

            // Call the OS API close function (handle is
            // already in X0):

            mov     svcReg, #sys_Close
            svc     #OSint

            // Check for error return code:

            file.checkError
```

```
            // Restore the volatile registers, except
            // X0 (because we return the function
            // result in X0):

            str     x0, [fp, #fc_locals.saveX0] // Return value.
            str     fp, [sp]    // w_locals.fpSave
            add     fp, fp, #fc_locals.volSave
            bl      restoreVolatile
            ldr     fp, [sp]    // Restore FP.
            ldr     x0, [fp, #fc_locals.saveX0]
            leave
            endp    file.close
```

The `file.close` function expects a file descriptor in X0 (returned by a successful call to `file.open` or `file.openNew`) and passes that descriptor along to the API `close()` function. Otherwise, it's similar in form to the `file.read` and `file.write` functions.

16.5.3 stdio.S Functions

The last source file in the files library is the *stdio.S* file. This module contains functions you can use to read and write strings on the console (standard I/O) device. Again, because of its size, I've broken this source file into more easily digestible pieces.

First, the `stdout.puts` function writes a (zero-terminated) string to the standard output device (usually the display console):

```
// stdio.S
//
// Standard input and standard output functions:

            #include    "../aoaa.inc"
            #include    "files.inc"
            #include    <sys/syscall.h>

            .code
            .align  2

// stdout.puts
//
// Outputs a zero-terminated string to standard output device
//
// X0- Address of string to print to standard output

            proc    stdout.puts, public

            locals  lcl_puts
          ❶ qword   lcl_puts.saveX0X1
            dword   lcl_puts.saveX2
            byte    lcl_puts.stkSpace, 64
            endl    lcl_puts

            enter   lcl_puts.size
```

```
                stp     x0, x1, [fp, #lcl_puts.saveX0X1]
                str     x2,     [fp, #lcl_puts.saveX2]

                mov     x1, x0

    // Compute the length of the string:

❷ lenLp:        ldrb    w2, [x1], #1
                cbnz    w2, lenLp
                sub     x2, x1, x0  // Compute length

                // Call file_write to print the string:

              ❸ mov     x1, x0
                mov     x0, #stdout
                bl      file.write

                // Return to caller:

                ldr     x2,     [fp, #lcl_puts.saveX2]
                ldp     x0, x1, [fp, #lcl_puts.saveX0X1]
                leave
                endp    stdout.puts
```

Note that this code does not preserve all the volatile registers ❶, because the `stdout.puts` function does not directly call an OS API function that might modify the registers. Therefore, this function preserves only the actual registers it uses.

This function will call `file.write` to write the string to the standard output device. The `file.write` function requires three parameters: a file descriptor (the `stdout` constant works fine for the descriptor value), the address of the data (string) to write, and a length value. Although `stdout.put` has the address of the string in X0, there is no length parameter. Therefore this code computes the length of the zero-terminated string whose address appears in X0 ❷.

NOTE *Scanning the string a byte at a time is a lame way to compute string length, but I've done so here because it is simpler than the alternative. If this really bothers you, you can link in the C stdlib `strlen()` function. However, keep in mind that making the system call and drawing all those pixels on the screen to print the string is many orders of magnitude slower than this string-length calculation, so you won't save much time by using faster string-length computation code.*

Once the length is computed, the `stdout.put` function calls `file.write` to actually print the string to the standard output device ❸. After restoring the few registers this code changed, the function returns.

Technically, `file.write` could return an error code (which `stdout.puts` ignores and doesn't return to its caller). However, the likelihood of such an error is low, so this code ignores errors. One likely problem could be if standard output were redirected to a disk file and there was an issue writing

to the disk, so this bug is worth addressing if this routine finds its way into production code; I chose not to address this here to keep the code cleaner (and because of the extremely low probability of occurrence).

The `stdout.newLn` function is identical to `stdout.puts`, except that it writes a fixed string (a newline character) to the standard output device:

```
// stdio.S (cont.)
//
// stdout.newLn
//
// Outputs a newline sequence to the standard output device:

stdout.nl:  .ascii  "\n"
nl.len      =       .-stdout.nl
            .byte   0
            .align  2

            proc    stdout.newLn, public
            locals  lcl_nl
            qword   lcl_nl.saveX0X1
            dword   lcl_nl.saveX2
            byte    lcl_nl.stkSpace, 64
            endl    lcl_nl

            enter   lcl_nl.size
            stp     x0, x1, [fp, #lcl_nl.saveX0X1]
            str     x2,     [fp, #lcl_nl.saveX2]

            lea     x1, stdout.nl
            mov     x2, #nl.len
            mov     x0, stdout
            bl      file.write

            ldr     x2,     [fp, #lcl_nl.saveX2]
            ldp     x0, x1, [fp, #lcl_nl.saveX0X1]
            leave
            endp    stdout.newLn
```

The `stdin.read` function is the input complement to `stdout.write`:

```
// stdio.S (cont.)
//
// stdin.read
//
// Reads data from the standard input
//
// X0- Buffer to receive data
// X1- Buffer count (note that data input will
//     stop on a newline character if that
//     comes along before X1 characters have
//     been read)
//
```

```
// Returns:
//
// X0- Negative value if error, bytes read if successful

            proc    stdin.read, public
            locals  sr_locals
            qword   sr_locals.saveX1X2
            byte    sr_locals.stkspace, 64
            dword   sr_locals.fpSave
            endl    sr_locals

            enter   sr_locals.size
            stp     x1, x2, [fp, #sr_locals.saveX1X2]

            // Call the OS API read function:

          ❶ mov     svcReg, #sys_Read
            mov     x2, x1
            mov     x1, x0
            mov     x0, #stdin
            svc     #OSint

            file.checkError

            ldp     x1, x2, [fp, #sr_locals.saveX1X2]
            leave
            endp    stdin.read
```

When you pass `stdin.read` a buffer's address and size, it will read up to that number of characters from the standard input device (usually the keyboard) and place those characters into the buffer. This function will stop reading characters when it either reads the specified number of characters or encounters a newline character (line feed) in the input.

The twist to this otherwise straightforward code is that it has to move the address and byte count around ❶ prior to calling the OS `read()` function. This is because the buffer address needs to go in X1 and X2 when calling `read()`, and the function has to load the standard input file descriptor into X0.

The `stdin.getc` function is a one-character version of `stdin.read`:

```
// stdio.S (cont.)
//
// stdin_getc
//
// Read a single character from the standard input.
// Returns character in X0 register

            proc    stdin.getc, public
            locals  sgc_locals
            qword   sgc_locals.saveX1X2
          ❶ byte    sgc_buf, 16
            byte    sgc_locals.stkspace, 64
            endl    sgc_locals
```

```
            enter   sgc_locals.size
            stp     x1, x2, [fp, #sgc_locals.saveX1X2]

            // Initialize return value to all 0s:

          ❷ str     xzr, [fp, #sgc_buf]


            // Call the OS API read function to read
            // a single character:

            mov     svcReg, #sys_Read
            mov     x0, #stdin
          ❸ add     x1, fp, #sgc_buf
            mov     x2, #1
            svc     #0Sint

          ❹ file.checkError
            cmp     x0, #0
            bpl     noError

            // If there was an error, return the
            // error code in X0 rather than a char:

            str     x0, [fp, #sgc_buf]

noError:
            ldp     x1, x2, [fp, #sgc_locals.saveX1X2]
          ❺ ldr     x0, [fp, #sgc_buf]
            leave

            endp    stdin.getc
```

The `stdin.getc` function returns the character it reads in X0 rather than placing it in a buffer. This function has to reserve storage for a buffer ❶ because the call to the API `read()` function requires a buffer. Technically, the buffer needs to be only eight characters long, but this function reserves 16 bytes just to help keep the stack 16-byte aligned. This code initializes the first 8 bytes of the buffer to 0 ❷. The function will actually return all 8 bytes (even though the read operation stores only a single byte into the buffer). This function computes the address of the buffer to pass to the API `read()` function ❸.

If, somehow, the call to the API `read()` function returns an error, the code will store the negative error return code into the first 8 bytes of the buffer ❹. Before returning, `stdin.getc` loads the 8 bytes at the beginning of the buffer into X0 and returns that value (this is either the single character plus seven 0s, a UTF-8 value, or the 8-byte negative error code) ❺.

NOTE *The `stdin.get` function does not read a single character from the keyboard and then immediately return to the caller. Instead, the OS will read a whole line of text from the*

keyboard and return the first character of that line. Successive calls to `stdin.get` *will read the remaining characters from that internal OS buffer. This is standard Unix behavior, not a specific feature of this function.*

The last function in the *stdio.S* file is `stdin.readln`:

```
// stdio.S (cont.)
//
// stdin.readln
//
// Reads a line of text from the user.
// Automatically processes backspace characters
// (deleting previous characters, as appropriate).
// Line returned from function is zero-terminated
// and does not include the ENTER key code (carriage
// return) or line feed.
//
// X0- Buffer to place line of text read from user
// X1- Maximum buffer length
//
// Returns:
//
// X0- Number of characters read from the user
//     (does not include ENTER key)

            proc    stdin.readln, public
            locals  srl_locals
            qword   srl_locals.saveX1X2
            dword   srl_locals.saveX3
            byte    srl_buf, 16
            byte    srl_locals.stkspace, 64
            endl    srl_locals

            enter   srl_locals.size
            stp     x1, x2, [fp, #srl_locals.saveX1X2]
            str     x3,     [fp, #srl_locals.saveX3]

            mov     x3, x0          // Buf ptr in X3
            mov     x2, #0          // Character count
            cbz     x1, exitRdLn    // Bail if zero chars.

            sub     x1, x1, #1      // Leave room for 0 byte.
readLp:
          ❶ bl      stdin.getc      // Read 1 char from stdin.

            cmp     w0, wzr         // Check for error.
            bmi     exitRdLn

          ❷ cmp     w0, #cr         // Check for newline code.
            beq     lineDone

            cmp     w0, #lf         // Check for newline code.
            beq     lineDone
```

```
            ❸ cmp     w0, #bs         // Handle backspace character.
              bne     addChar

    // If a backspace character came along, remove the previous
    // character from the input buffer (assuming there is a
    // previous character):

                cmp     x2, #0          // Ignore BS character if no
                beq     readLp          // chars in the buffer.
                sub     x2, x2, #1
                b.al    readLp

    // If a normal character (that we return to the caller),
    // add the character to the buffer if there is room
    // for it (ignore the character if the buffer is full):

❹ addChar:      cmp     x2, x1          // See if you're at the
                bhs     readLp          // end of the buffer.
                strb    w0, [x3, x2]    // Save char to buffer.
                add     x2, x2, #1
                b.al    readLp

    // When the user presses the ENTER key (or line feed)
    // during input, come down here and zero-terminate the string:

    lineDone:
              ❺ strb    wzr, [x3, x2]

    exitRdLn:   mov     x0, x2          // Return char cnt in X0.
                ldp     x1, x2, [fp, #srl_locals.saveX1X2]
                ldr     x3,     [fp, #srl_locals.saveX3]
                leave
                endp    stdin.readln
```

This function, intended mainly for interactive use, reads a line of text from the keyboard with a modicum of editing (handling backspace characters), placing those characters into a buffer. In many respects, it works just like `stdin.read`, except that pressing BACKSPACE deletes a character from the input buffer rather than returning the backspace ASCII code as a character of the buffer.

This function repeatedly calls `stdin.getc` ❶, reading one character at a time. If `stdin.getc` returns an error (a negative return value), this function immediately returns, passing the error code on to its caller.

The code checks whether the input line is complete by comparing the character against the ASCII codes for a carriage return or newline (line feed) ❷. If the character matches one of these two, the code exits the loop.

The `stdin.readln` function then checks for a backspace character ❸. If it is a backspace, this function will delete the previous character from the input buffer (if there was one). If the character is not a backspace, the code branches down to ❹, where it appends the character to the end of the buffer.

When the function finds a carriage return or line feed in the input stream, it transfers control to ❺, where it zero-terminates the string and returns the number of characters actually read in the X0 register.

Beyond processing backspace characters, there are two additional differences between reading a line of text with `stdin.readln` and simply calling `stdin.read`. First, `stdin.readln` will zero-terminate the string read into the buffer. Second, `stdin.readln` does not place the newline character (or carriage return) in the buffer.

16.5.4 File I/O Demo Application

The simple application in Listing 16-4 demonstrates the use of the *file.a* library.

```
// Listing16-4.S
//
// File I/O demonstration:

            #include    "aoaa.inc"
            #include    "files/files.inc"
            #include    <sys/syscall.h>

            #if isMacOS

// Map main to "_main" as macOS requires
// underscores in front of global names
// (inherited from C code, anyway).

#define main _main
sys_Exit    =   SYS_exit

            #else

sys_Exit    =   __NR_exit

            #endif

            .data

// Buffer to hold line of text read from user:

inputLn:    .space  256, (0)
inputLn.len =       .-inputLn

// Buffer to hold data read from a file:

fileBuffer: .space  4096, (0)
fileBuffer.len =    .-fileBuffer

// Prompt the user for a filename:

prompt:     .ascii  "Enter (text) filename:"
prompt.len  =       .-prompt
            .byte   0
```

```
// Error message string:

badOpenMsg: wastr   "Could not open file\n"

OpenMsg:    wastr   "Opening file: "

            .code

// Here is the asmMain function:

            proc    main, public
            locals  am
            dword   am.inHandle
            byte    am_stkSpace, 64
            endl    am

            enter   am.size

// Get a filename from the user:

          ❶ lea     x0, prompt
            bl      stdout.puts

            lea     x0, inputLn
            mov     x1, #inputLn.len
            bl      stdin.readln
            cmp     x0, #0
            bmi     badOpen

            lea     x0, OpenMsg
            bl      stdout.puts
            lea     x0, inputLn
            bl      stdout.puts
            bl      stdout.newLn

// Open the file, read its contents, and display
// the contents to the standard output device:

          ❷ lea     x0, inputLn
            mov     x1, #O_RDONLY
            bl      file.open
            cmp     x0, xzr
            ble     badOpen

            str     x0, [fp, #am.inHandle]

// Read the file 4,096 bytes at a time:

readLoop:
          ❸ ldr     x0, [fp, #am.inHandle]
            lea     x1, fileBuffer
            mov     x2, fileBuffer.len
            bl      file.read
```

```
            // Quit if there was an error or
            // file.read read 0 bytes:

            cmp     x0, xzr
            ble     allDone

            // Write the data just read to the
            // stdout device:

          ❹ mov     x2, x0        // Bytes to write
            lea     x1, fileBuffer
            mov     x0, #stdout
            bl      file.write
            b.al    readLoop

badOpen:    lea     x0, badOpenMsg
            bl      stdout.puts

allDone:
          ❺ ldr     x0, [fp, #am.inHandle]
            bl      file.close

            // Return error code 0 to the OS:

            mov     svcReg, #sys_Exit
            mov     x0, #0
            svc     #0Sint
            endp    main
```

This program begins by prompting the user to enter a filename ❶. It reads this filename from the user and then echoes the filename to the display. The program opens the file and saves away the file handle that `file.open` returns ❷. If an error occurred opening the file, the program drops down to the `badOpen` label, prints an error message, and quits.

Next, the program continuously reads a block of (up to) 4,096 bytes until the end of the file is reached (or another error occurs) ❸. When reading from a file, the `file.read` function will read the full 4,096 bytes, ignoring any newline characters (it stops on newlines only when reading from the standard input). If this function reads 0 bytes from the input, it has reached the end of the file, and the loop exits.

The code then writes the bytes read to the standard output device ❹, using the return value from `file.read` as the byte count on the call to `file.write`. This is because the last block of bytes read from the file might not be 4,096 bytes in length; if it read fewer than 4,096 bytes, the next read will return 0 bytes and the operation will be complete. Once the program completes, it closes the file and quits ❺.

Here's the makefile that will build the program in Listing 16-4:

```
# Listing16-4.mak
#
# makefile to build the Listing16-4.S file
```

```
unamestr=`uname`

Listing16-4:Listing16-4.S aoaa.inc files/files.inc files.a
    cd files; make -f files.mak; cd ..
    g++ -D$(unamestr) -o Listing16-4 Listing16-4.S files.a

clean:
    rm -f Listing16-4.o
    rm -f Listing16-4
    rm -f file.a
    cd files; make -f files.mak clean; cd ..
```

Here's a sample build operation and execution of the program:

```
% make -f Listing16-4.mak clean
rm -f Listing16-4.o
rm -f Listing16-4
rm -f file.a
cd files; make -f files.mak clean; cd ..
rm -f files.o
rm -f volatile.o
rm -f stdio.o
rm -f files.a
% make -f Listing16-4.mak
cd files; make -f files.mak; cd ..
g++ -c -D`uname` files.S
g++ -c -D`uname` stdio.S
g++ -c -D`uname` volatile.S
ar rcs files.a files.o stdio.o volatile.o
cp files.a ..
g++ -D`uname` -o Listing16-4 Listing16-4.S files.a
% ./Listing16-4
Enter (text) filename:Listing16-4.mak
Opening file: Listing16-4.mak
# listing16-4.mak
#
# makefile to build the Listing16-4.S file.

unamestr=`uname`

Listing16-4:Listing16-4.S aoaa.inc files/files.inc files.a
    cd files; make -f files.mak; cd ..
    g++ -D$(unamestr) -o Listing16-4 Listing16-4.S files.a

clean:
    rm -f Listing16-4.o
    rm -f Listing16-4
    rm -f file.a
    cd files; make -f files.mak clean; cd ..
```

I've used the *Listing16-4.mak* text file as the input for this run of the program.

16.6 Calling System Library Functions Under macOS

As I mentioned earlier, Apple frowns on applications that make direct calls to the macOS kernel via the `svc` instruction. The company claims the proper way to make those calls is via the C library code Apple has provided. This chapter has shown you the low-level calls because, well, that was the purpose of this chapter; if you're the one writing the C library code (or similar library code that interfaces to the OS), you need to know this information. However, I would be remiss if I didn't show you how Apple recommends interfacing applications with macOS.

I've created a variant of the *files.a* library, stored in the *files-macOS* directory in the online source code set, that links in the kernel `read()`, `write()`, `open()`, and `close()` functions. To avoid redundancy, I don't print all that code in this chapter, but I'll list the `file.write` function here to give you an idea of the simplicity of the change:

```
// file.write
//
// Write data to a file handle.
//
// X0- File handle
// X1- Pointer to buffer to write
// X2- Length of buffer to write
//
// Returns:
//
// X0- Number of bytes actually written
//     or -1 if there was an error

            proc    file.write, public
            locals  fw_locals
            qword   fw_locals.saveX0
            byte    fw_locals.volSave, volatile_save.size
            byte    fw_locals.stkspace, 64
            dword   fw_locals.fpSave
            endl    fw_locals

            enter   fw_locals.size

            // Preserve all the volatile registers because
            // the OS API write function might modify them.
            //
            // Note: because fw_locals.volSave is at the
            // bottom of the activation record, SP just
            // happens to be pointing at it right now.
            // Use it to temporarily save FP so you can
            // pass the address of w_locals.volSave to
            // saveVolatile in the FP register.

            str     fp, [sp]    // fw_locals.fpSave
            add     fp, fp, #fw_locals.volSave
            bl      saveVolatile
            ldr     fp, [sp]    // Restore FP.
```

```
            // Okay, now do the write operation (note that
            // the write arguments are already sitting
            // in X0, X1, and X2 upon entry into this
            // function):

          ❶ bl      _write

            // Check for error return code:

            file.checkError

            // Restore the volatile registers, except
            // X0 (because we return the function
            // result in X0):

            str     x0, [fp, #fw_locals.saveX0] // Return value.
            str     fp, [sp]    // w_locals.fpSave
            add     fp, fp, #fw_locals.volSave
            bl      restoreVolatile
            ldr     fp, [sp]    // Restore FP.
            ldr     x0, [fp, #fw_locals.saveX0]
            leave
            endp    file.write
```

The only difference between this version of `file.write` and the version in the original *files.a* library is that I've replaced the `svc` instruction sequence with a call to the `_write()` function ❶.

The new *files.a* library also includes a couple of changes to the *files.inc* header file. The most important change is to the `file.checkError` macro:

```
            .macro  file.checkError

            cmp     x0, #-1
            bne     0f
            getErrno
            neg     x0, x0
0:
            .endm
```

The macOS `_write()` function returns –1 when an error occurs, since C code can't test the carry flag. Therefore, I modified `file.checkError` to handle errors the same way Linux does.

I had to build the *files-macOS* library first (to create a new version of *file.a*, replacing the version that made direct OS calls), then made *Listing16-4.S* by using the *file.a* library from *files-macOS*. The program ran the same as the original file I/O example from the previous section.

In theory, you could use this same approach with Linux, creating slightly more portable code between the two OSes. However, the `svc` API interface under Linux is well defined and documented, so there is no reason not to call the API functions directly.

16.7 Creating Assembly Applications Without GCC

Throughout this chapter, I've continued to use GCC to assemble and link the assembly language files. That's because most of the example code in this chapter includes *aoaa.inc*, and that file depends on the CPP. You might view this approach with suspicion, thinking GCC might be sneaking some C code into your program. And you would be right: even when you build a "pure" assembly language program with GCC, it links in some code to set up the environment prior to the execution of your program (so that if you do call any C library code, the environment has been set up for it).

Generally, such extra code is of little consequence—it executes once, is fairly fast, and doesn't take up that much space. However, if you are an absolute purist and you want to execute only the code you've written, you can do so with a little extra work. You just won't be able to use *aoaa.inc*, and you'll have to write non-portable code specifically for macOS or Linux.

Listing 16-5 is a "pure" assembly language program written for Linux.

```
❶ // Listing16-5.s
  //
  // A truly stand-alone "Hello, world!" program
  // written for Linux

          .text
       ❷ .global     _start
          .align     2
  hwStr:  .asciz     "Hello, world!\n"
  hwSize  =          .-hwStr
          .align     2

❸ _start:

          mov        x0, #1          // stdout file handle
          adr        x1, hwStr       // String to print
          mov        x2, #hwSize     // Num chars to print
          mov        X8, #64         // __NR_write
          svc        #0              // Call OS to print str.

          mov        X8, #93         // __NR_exit
          mov        x0, #0
          svc        #0              // Quit program.
```

Please note that this filename must have a lowercase *.s* suffix ❶; you will not be compiling this using GCC, so you won't be using CPP with this code. Under Linux, the default program entry point is named `_start`. Therefore, this code declares `_start` as a global symbol ❷ and uses `_start` as the entry point for the program ❸. I got away with using `main` (or `_main`) in earlier examples in this chapter because the C code that GCC links in supplies the `_start` label and transfers control to `main` (or `_main`); however, as we're giving up the GCC-generated code, we have to explicitly supply the `_start` label.

To assemble, link, and run this program, use the following Linux commands:

```
as -o Listing16-5.o Listing16-5.s
ld -s -o Listing16-5 Listing16-5.o
./Listing16-5
Hello, world!
```

To make this program run under macOS, you must first modify the source code to use the appropriate macOS API constants, as shown in Listing 16-6.

```
// Listing16-6.s
//
// A truly stand-alone "Hello, world!" program
// written for macOS

        .text
        .global     _start
        .global     _main    // Later versions of macOS require this name.
        .align      2
hwStr:  .asciz      "Hello, world!\n"
hwSize  =           .-hwStr
        .align      2

_start:
_main:

        mov         x0, #1          // stdout file handle
        adr         x1, hwStr       // String to print
        mov         x2, #hwSize     // Num chars to print
        mov         X16, #4         // SYS_write
        svc         #0x80           // Call OS to print str.

        mov         X16, #1         // SYS_exit
        mov         x0, #0
        svc         #0x80           // Quit program.

        svc         #0              // Quit program.
```

Assembling the code is similar to Linux (note that the program name suffix is also a lowercase *.s*):

```
as -arch arm64 -o Listing16-6.o Listing16-6.s
```

Linking the program, however, is a bit more complex:

```
ld -macos_version_min 12.3.0 -o HelloWorld Listing16-6.o \
   -lSystem -syslibroot `xcrun -sdk macosx --show-sdk-path` -arch arm64
```

It's best to use a makefile when building pure assembly files under macOS; manually typing these commands every time you want to build the application can be quite tedious!

As you can see, the linker (`ld`) command still links in a bunch of C code (`libSystem`). There's no other way (that I know of) to avoid this, which is why I'm perfectly happy letting GCC do all this work for me.

NOTE *Apple isn't kidding when it warns you against writing code like this. Between the time I first wrote this "Hello, World!" and the time this chapter was reviewed, Apple made changes to its system that broke the program. In particular, the linker now expects the program to be named* `_main` *rather than* `_start`*, and the command line for* `ld` *has some subtle changes. Moral of the story: stick with GCC (Clang) to do all this work for you.*

16.8 For More Information

- You can find the system call numbers for Linux at *https://github.com/torvalds/linux/blob/v4.17/include/uapi/asm-generic/unistd.h*.
- Find the system call numbers for macOS at *https://github.com/opensource-apple/xnu/blob/master/bsd/kern/syscalls.master* or *https://opensource.apple.com/source/xnu/xnu-1504.3.12/bsd/kern/syscalls.master*.

TEST YOURSELF

1. What is the purpose of the svc instruction?
2. What is the svc operand for Linux?
3. What is the svc operand for macOS?

PART IV

REFERENCE MATERIALS

A

THE ASCII CHARACTER SET

Binary	Hex	Decimal	Character
0000_0000	00	0	NUL
0000_0001	01	1	CTRL-A
0000_0010	02	2	CTRL-B
0000_0011	03	3	CTRL-C
0000_0100	04	4	CTRL-D
0000_0101	05	5	CTRL-E
0000_0110	06	6	CTRL-F
0000_0111	07	7	bell
0000_1000	08	8	BACKSPACE
0000_1001	09	9	TAB
0000_1010	0A	10	line feed
0000_1011	0B	11	CTRL-K
0000_1100	0C	12	form feed
0000_1101	0D	13	RETURN
0000_1110	0E	14	CTRL-N

(continued)

Binary	Hex	Decimal	Character
0000_1111	0F	15	CTRL-O
0001_0000	10	16	CTRL-P
0001_0001	11	17	CTRL-Q
0001_0010	12	18	CTRL-R
0001_0011	13	19	CTRL-S
0001_0100	14	20	CTRL-T
0001_0101	15	21	CTRL-U
0001_0110	16	22	CTRL-V
0001_0111	17	23	CTRL-W
0001_1000	18	24	CTRL-X
0001_1001	19	25	CTRL-Y
0001_1010	1A	26	CTRL-Z
0001_1011	1B	27	ESC (CTRL-[)
0001_1100	1C	28	CTRL-\
0001_1101	1D	29	CRTL-]
0001_1110	1E	30	CTRL-^
0001_1111	1F	31	CTRL-_
0010_0000	20	32	space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41	)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2

Binary	Hex	Decimal	Character
0011_0011	33	51	3
0011_0100	34	52	4
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V

(continued)

Binary	Hex	Decimal	Character
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93	]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z

Binary	Hex	Decimal	Character
0111_1011	7B	123	{
0111_1100	7C	124	|
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	DEL

B

GLOSSARY

A

AARCH64
The 64-bit variant of the ARM architecture. Also known as ARM64.

address
The numeric index associated with a memory location.

address bus
A set of electronic signals that hold a binary address of a memory element.

address space layout randomization (ASLR)
The use of random load addresses for program modules (to reduce the possibility of hacks and exploits in the code).

alternate half-precision control bit (AHP)
A bit in the floating-point control register that selects an alternative half-precision format (different from the IEEE half-precision format).

application binary interface (ABI)
A set of rules that allows interaction between programming languages and systems. Includes rules for passing parameters, data types, and other features.

ARM
Advanced RISC machines (originally an acronym for Acorn RISC machine).

ASCII
American Standard Code for Information Interchange (a standardized character set).

assembler
A compiler for an assembly language.

B

.bss
A block started by a symbol; a data area in the program containing uninitialized data (that usually doesn't consume space in the executable file).

C

cache
The high-speed memory sitting between the CPU and main memory to improve system performance.

control bus
A set of electronic signals from the CPU that control activities such as reading, writing, and generating wait states.

D

data bus
A set of electronic signals from the CPU that transfer data between CPU and external devices (such as memory or I/O).

default NaN enable (DN)
When default NaN is enabled (FPSCR[25]), any operation that generates a NaN result returns the default NaN value.

denormalized numbers
Floating-point numbers with an exponent of 0 and no implied HO mantissa bit. Denormalized numbers offer less precision than normalized numbers, but the alternative is setting the value to 0 upon underflow.

E

effective address (EA)
The final memory address computed for an addressing mode (generally involving addition, subtraction, and shifting).

equate
An assembler directive that associates a value with a symbolic name.

F

facade code
The code around a call to a function to modify the behavior of that function (such as adjusting input and output values, checking for range limits, and other such operations).

flags
Boolean variables that indicate the state of a system. Generally, this term is associated with the condition code flags in the PSTATE register.

floating-point status and control register
The FPSCR contains the status flags that are set by floating-point operations, and it contains the status flags that control the operation of various floating-point instructions.

floating-point unit (FPU)
The (optional on some CPUs) hardware that is responsible for computing floating-point arithmetic operations.

frame pointer
A special register used to access parameters, local variables, and other items in an activation record.

G

Gas
GNU assembler.

general-purpose registers
Special memory cells accessible to user applications, used for most integer and address calculations on ARM CPUs.

H

high-level language (HLL)
Programming language that allows developers to create software that is independent of the underlying machine architecture.

high-order (HO)
Most significant.

I

idiom
A peculiarity of an instruction or operation. For example, shifting a binary number one bit position to the left is the same as multiplying by 2.

input/output (I/O)
Data provided to a CPU from outside sources (inputs) or presented to the outside world by the CPU (outputs).

integrated development environment (IDE)
Generally consists of two or more programs combined into a single tool. At the very least, an IDE will include a text editor and a compiler. Most IDEs also include a debugger, build (make) system, and other file management tools.

L

last in, first out (LIFO)
An access mechanism whereby the last object inserted into a list will be the first object removed from the list. CPU stacks use this mechanism to preserve and restore data and return addresses.

`ld`
A linker program (loader).

lexically scoped symbols
Symbols that are visible only within a particular block where they are defined. For example, in an HLL, symbols defined within a function or procedure are local to that function/procedure and are not visible outside it.

link register (LR)
Holds the return address after a `bl` instruction so the target (of `bl`) function or procedure can return to the caller.

low-order (LO)
Least significant.

M

machine code
The binary instruction encoding for each assembly language instruction.

manifest constant
A symbolic name in a program that is textually replaced by its associated value.

memory management unit (MMU)
The hardware in the CPU that controls access to memory (protection) and remaps memory addresses (typically to allow safe multitasking).

N

natural boundary
An object's address that is a multiple of that object's size.

nonvolatile registers
Registers that must be preserved across a function call.

not a number (NaN)
A special floating-point value that represents an illegal result (other than infinity) from a floating-point operation.

O

operating system (OS)
The software that controls computer operations, such as scheduling tasks, executing applications, and providing access to I/O devices.

operation code (opcode)
A numeric value that encodes a machine instruction.

P

position-independent executables (PIE)
Code that can execute at any address in memory without modification (relocation).

processor state register (PSTATE)
A special register that holds the condition code flags, interrupt masks, and other miscellaneous processor control bits.

program counter
A special register that holds the memory address of the currently executing instruction.

program-counter relative (PC-relative) addressing
Memory addresses based on an offset from the current instruction.

R

reduced instruction set computer (RISC)
Read as "reduced-instruction set computer" (not "reduced instruction-set computer"). In a RISC machine, the CPU designer creates instructions that do as little as possible in order to simplify the hardware needed to implement those instructions. In theory, this allows the designer to create faster CPUs at a lower cost (though the Intel x86 series demonstrates that non-RISC CPUs can have these attributes).

registers
Specialized memory components built directly into the CPU and accessible by most machine instructions.

S

single instruction, multiple data (SIMD)
A SIMD instruction operates on multiple pieces of data concurrently. For example, a SIMD add instruction may perform as many as 16 different addition operations in parallel. Though SIMD instructions are more difficult to use than single-instruction, single-data instructions (for example, typical ARM assembly instructions), they have the potential for executing application much faster.

stack pointer register (SP)
A special register that references a hardware stack in memory.

subnormal
See denormalized numbers.

system bus
A collection of electronic signals comprising the address, data, and control buses.

T

token
A string of characters that has special meaning to a compiler. Examples include operators, reserved words, identifiers, literal constants, and other punctuation.

trampoline
See veneer.

V

veneer
A special code sequence to extend the range of a control-transfer instruction beyond the normal displacement allowed by an instruction; also known as a *trampoline*.

volatile registers
Registers that don't have to be preserved across a function call.

W

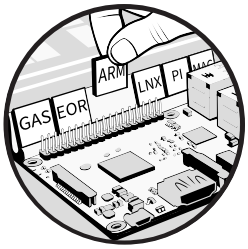
WZR
32-bit zero register.

X

XZR
64-bit zero register.

C

INSTALLING AND USING GAS



To compile ARM assembly language source files (including those in this book) on your computer, you will need the Gas assembler and other tools installed. The assembler is part of the GNU C compiler suite, so if you install that package on your system, you'll get the assembler as well. Since the installation instructions differ by OS, this appendix will help you locate and install the files you need on your machine.

Before attempting to install any software, check to see whether that software is already installed by issuing the following commands:

```
as --version
gcc --help
```

If these commands print a help message for Gas and GCC, the assembler (and GCC) are already installed on your system, and you're good to go. However, if either command complains that the file is not found (or otherwise does not print the help screen), skip to the section for your OS to install GCC and Gas.

C.1 macOS

To write assembly code on an Apple Silicon machine (such as an M*x* class machine or even an iPad or iPhone), you must install Apple's Xcode development platform on your Mac. Go to the Apple App Store and then locate, download, and install Xcode. This is a very large file (many gigabytes) and will take a while to download. After you run the installer, you should be able to compile the assembly code in this book.

Technically, when installing Xcode, you're installing Apple's LLVM Clang compiler and Clang assembler, not GCC and Gas. However, these tools are mostly compatible with GCC and Gas (and can certainly handle all the source code appearing in this book). The two assemblers have occasional syntax differences, and the *aoaa.inc* file was created to help smooth over those differences (review the *aoaa.inc* source code if you are interested in seeing some of them). Various comments throughout this book also point out the differences between Gas and the Clang assembler (for example, see the discussion of the `lea` macro in section 3.8, "Getting the Address of a Memory Object," on page 153).

C.2 Linux

Many Linux installations come with GCC and Gas already installed. If you are running Debian/Ubuntu Linux or Raspberry Pi OS, enter **`gcc --version`** at the command line; if the shell doesn't respond with an error but instead prints information on GCC, you're all set.

If you do need to install GCC and Gas, the first step, as with any Linux installation (this book assumes a Debian/Ubuntu installation), is to update your system:

```
sudo apt update
```

Note that this requires an account with sudo privileges.

Next, install the *build-essential* package with the following command:

```
sudo apt install build-essential
```

This installs several programs, including `gcc`, Gas, `make`, and `ld`. Verify that the system has properly installed these tools by using the following command:

```
gcc --version
```

This should print the version information for GCC if it has been properly installed.

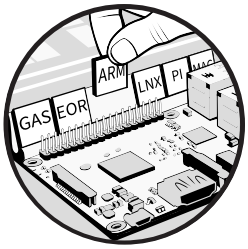
Optionally, you can install the man pages for GCC and the other tools by using the following command:

```
sudo apt-get install manpages-dev
```

If you are using a different variant of Linux, please consult the documentation for your distribution or search for installation instructions online to install GCC.

D

THE BASH SHELL INTERPRETER



The *Bourne-again shell*, otherwise known as the *Bourne shell* or *bash*, is a Unix shell interpreter. Bash is an upgraded version of the venerable Unix sh (shell) program, the default shell program in Unix System 7.

Bash is the typical shell used in Linux systems (though other shells are available, such as zsh and csh). Most Linux and macOS shell programs are roughly compatible with one another for simple command line activities, but they differ in their support for sophisticated shell programming purposes.

All the programming examples throughout this text run GCC and the Gas assembler via shell commands given by a bash command line. Therefore, you should have at least a small amount of bash knowledge in order to understand the basic commands in this book. This appendix gives instructions for using bash, including descriptions of its more common commands, but this discussion largely applies to other shell interpreters as well.

To avoid having to refer to specific OS or distribution names, I'll use the name *Unix* in this appendix to refer to the underlying system (at the time of this writing, Unix is a registered trademark of The Open Group).

D.1 Running Bash

The bash shell interpreter is an application similar to other Unix applications. To use it, you must first execute the bash application. On text-based Unix systems, some sort of shell application will run after you log in to the system. You can set up your system to automatically run bash (or any other shell program).

On GUI-based Linux systems or on macOS systems, you usually have to run a terminal program to start a shell interpreter. In either case, you'll typically be presented with a *command prompt* when the shell application runs. This should be `$` or `#`, depending on whether you're logged in as a normal user or root, respectively; some shells display a `%` prompt. After printing the command prompt, the shell will wait for you to type a command.

At this point, you're running a shell interpreter, though it might not be bash; it could be the standard sh shell or another shell (for example, the macOS *terminal* application runs zsh). Though most shells will behave approximately the same, to ensure you're running bash, type the following line (followed by ENTER) after the command prompt:

```
$ bash
```

This will ensure that you're running the bash application so that all the comments in this appendix will apply. You can terminate this instance of bash and return to the original shell by executing the `exit` command from the command line.

D.2 Command Lines

The previous example (`bash`) is an instance of a command given to a shell interpreter. *Commands* consist of a line of text entered after a command prompt (typically via the keyboard) and take the following form:

```
commandName optionalCommandLineArguments optionalRedirection
```

This whole line of code is known as a *command line*, which consists of three components: a command (usually a single word, such as `bash` in the previous example), followed by optional command line arguments, and, finally, optional redirection or piping operands.

The command is the name of a built-in bash command or the name of an executable application (or shell script). For example, the command could be the name of an assembled assembly language source file you've just created.

D.2.1 Command Line Arguments

Command line arguments are strings of characters separated by spaces or tabs that bash will pass along to the application. The exact syntax of these command line arguments is application dependent. Some simple applications (including the assembly language examples in this book) may completely ignore any command line arguments; others may require very specific arguments and report an error if they are not syntactically correct.

Some applications define two types of command line arguments: options and arguments. Historically, command line *options* consist of a dash (-) prefix followed by a single character, or a double dash (--) prefix followed by a sequence of characters. For example, the bash command supports the option

```
bash --help
```

which displays help information and then terminates without running the bash interpreter.

In contrast, an actual command line *argument* is typically a filename or other word (or string) that the application will use as an input value. Consider the following command line:

```
bash script
```

In this example, bash will start a second instance of itself and read a set of commands (one per line of text) from the *script* file and execute those commands as though they had been typed from the keyboard. The *build* script file that appeared on page 38 is a good example of a shell script.

Because bash uses spaces or tabs to separate command line arguments, problems arise if you want to specify a single argument that contains such *delimiters*, characters that separate items on the command line. Fortunately, bash provides a syntax that allows you to include such delimiters on the command line: if you surround a command line argument with quotation marks, bash will pass everything within the quotes to the application as a single command line argument.

For example, if a command requires a filename and the filename you wish to use contains spaces, you can pass that filename to the command as follows:

```
command "filename with spaces"
```

Bash will not include the quotes as part of the argument it passes to the command. If you need to pass a quote character as part of a command line argument, precede the quote with a backslash character (\). For example, consider the following command:

```
command "argument containing \"quotes\""
```

This passes *argument containing "quotes"* as a single argument to *command*.

As you will see later in this appendix, you can also surround a command line argument with single quotes (apostrophe characters). The difference has to do with variable expansion.

D.2.2 Redirection and Piping Arguments

Bash programs (and Unix-like OSes in general) provide a standard input device, a standard output device, and a standard error device.

The *standard input device* is usually the console keyboard. If a program reads data from the standard input device, the program will halt until the user types a line of text from the keyboard.

The *standard output device* is the console display. If an application writes data to the standard output device, the system will display it on the display screen. The standard error device also defaults to the console display, so data written to the standard error device is also written to the display.

The bash shell provides the ability to *redirect* input and output by using special arguments on the command line. I/O redirection typically allows you to specify a filename. When redirecting the standard input device, the application will read lines of text from a file (rather than from the keyboard). When redirecting the standard output, the application will write data to a text file rather than to the console display.

To redirect the standard input from a file, use a command line argument of the form

```
command <InputFile
```

where *InputFile* is the name of a file containing text to be read by the application. Whenever the application command would normally read a line of text from the keyboard, it will instead read that line of text from the specified file.

To redirect the standard output to a file, use the following command line syntax:

```
command >OutputFile
```

Any output normally written to the standard output device (the display) will be written to the specified file (*OutputFile*). This syntax will delete the contents of any existing file named *OutputFile* and replace its contents with the output of the command application.

A variation of output redirection will append a program's output to the end of an existing file rather than replacing its contents. To use output redirection this way, use this syntax:

```
command >>OutputFile
```

Note that redirecting the standard output device does not change the standard error output device. If you've redirected the standard output to a

file and an application writes data to the standard error device, that output still appears on the console display. You can redirect the standard error device by using the following syntax:

```
command 2>ErrorOutput
```

The `2>` tells bash to redirect the output sent to file handle 2. Under Unix-style systems, handle 0 is reserved for the standard input, handle 1 is reserved for the standard output, and handle 2 is reserved for the standard error output device. Sticking the handle number in front of the `>` specifies which output to redirect.

If you like, you can also redirect the standard output by using this syntax:

```
command 1>OutputFile
```

The final form of I/O redirection is the *pipe*, which connects the standard output from one application to the standard input of a second application. This allows the second application to read, as input, the output from the first application. Here's the syntax for a pipe redirection:

```
command1 OptionalCommand1Arguments | command2 OptionalCommand2Arguments
```

This tells bash to redirect the output from `command1` as the input to `command2`.

D.3 Directories, Pathnames, and Filenames

When you run bash, it will default to a current directory in the OS's file structure. Unix calls this the *current working directory*. Whenever you run bash (for example, when you first log in), the current working directory is typically your home directory (as determined by the OS). For example, on my Debian system, this is */home/rhyde* (under macOS, it's */Users/rhyde*).

When you specify a filename on the command line that does not contain any slash characters, bash or the application will assume that file exists in the executable path supplied to the system. A *pathname* consists of a sequence of one or more directory names, separated by slashes, ending with a filename. A relative pathname begins with a directory name; the system looks for that directory within the current working directory. For example, *dir1/dir2/filename* specifies a file (*filename*) appearing in *dir1* in the current working directory and within *dir2*, which is itself within *dir1*. An *absolute pathname* begins with a slash followed by the outermost root directory. For example, */home/rhyde/x.txt* specifies the file *x.txt* appearing in the */home/rhyde* directory (my home directory under Debian).

The tilde special character (~) is shorthand for the current user's home directory. Therefore, *~/x.txt* is another way for me to specify */home/rhyde/x.txt* on my Debian system. This scheme also works in macOS, so it's a useful way to specify user directory paths in a system-independent fashion.

The special character period (.), by itself, is shorthand for the current working directory. The double-period sequence (..) is shorthand for the directory that contains the current working directory (the parent directory). For example, *../x.txt* refers to the file named *x.txt* in the parent directory. On Linux-based systems, to execute an application from the current working directory, you must specify *./filename* rather than just *filename* (unless you've placed *./* in your execution path).

Some Unix commands allow you to specify multiple filenames on the command line. Such commands often allow the use of wildcard characters to specify multiple names. Unix supports a rich set of regular expressions when specifying wildcards, one of which you'll commonly use: the asterisk (*). Bash will match any number of characters (zero or more) in place of the asterisk. Therefore, the filename **.txt* will match any file ending with the four-character sequence *.txt* (this includes *.txt* by itself).

D.4 Built-in and External Bash Commands

Bash supports two types of commands: built-in and external. *Built-in commands* exist as part of the bash application itself; a function inside the bash source code handles the given built-in command. *External commands* correspond to executable programs separate from bash that bash will load and execute (then take control back from those programs when they terminate). Built-in commands are always available when you run bash, but external commands may or may not be available, depending on the presence of the executable code for those commands. You can assume that the commands appearing in the following subsections are all external, unless otherwise noted.

The assembly language example programs in this book are examples of external commands. When you enter something like

```
./Listing1-5
```

at the command line, bash will locate the *Listing1-5* executable in the current working directory (`./`) and attempt to execute that code.

For security reasons, bash will not automatically execute a program in the current working directory unless you explicitly prepend the characters `./` to the executable's name. Bash assumes that program names without explicit path information can be found in the *execution path*. The execution path (see section D.6.1, "Defining Shell Script Variables and Values," on page 961) is a list of directories where bash will search for an executable program you specify without explicit path information. Usually, bash will look in places such as */bin*, */usr/bin*, and */sbin* for executable programs.

To enable bash to execute programs from your current working directory without having to prefix the executable filename with the `./` characters, you can add `./` to your execution path. However, there are some very good security reasons for not doing this. For more on this, see section D.8, "For More Information," on page 968.

D.5 Basic Unix Commands

It would be impossible to describe all Unix commands in this appendix. That would take a large book by itself. This section describes several commands useful to those developing assembly language programs, along with some of their options and parameters. For information on additional bash commands, check out section D.8, "For More Information," on page 968.

D.5.1 man

If you know the name of the command but are unsure about the syntax for its command line parameters and options, you can use the `man` command to learn about it. This command brings up the manual page for a (supported) command, with the syntax

```
man CommandName
```

where *`CommandName`* is the name of the command whose manual page you would like to read. For example, the following brings up the manual page for the `man` command itself:

```
man man
```

You can use `man`, with the command names listed in the following subsections, to learn additional information about each.

D.5.2 cd or chdir

You can set the current working directory by using the `cd` (change directory) command (`chdir` is an alias of this command). The standard syntax is

```
cd DirectoryPath
```

where *`DirectoryPath`* is a relative or absolute path to a directory in the filesystem. Unix will report an error if the directory does not exist or if this is the name of a file rather than a directory.

If you specify the `cd` command without any arguments, it will switch to the current user's home directory. This is equivalent to entering the following on the command line:

```
cd ~
```

The `cd` and `chdir` commands are built into bash.

D.5.3 pwd

The `pwd` (print working directory) command prints the path to the current working directory. Bash is generally set up to print the current working directory as part of the command line prompt; if this is the case for your

system, you probably won't need to use `pwd`. This is also an internal bash command.

D.5.4 ls

The `ls` (list directory) command prints a directory listing to the standard output. When used with no options, it displays the contents of the current directory.

When printing the directory listing to the display, `ls` defaults to a multicolumn format. If you direct the output to a file, by redirecting the standard output or by using a pipe, the command prints the listing in a single-column format.

If you supply a directory path as an argument, the `ls` command will display the contents of the specified directory (assuming it exists). If you specify a pathname to a file as an argument, the `ls` command will display only that filename (again, assuming that file exists in the specified path).

By default, the `ls` command will not display filenames that begin with a period. Unix treats such files as *hidden*. If you want to display such filenames, use the `-a` command line option:

```
ls -a
```

By default, the `ls` command lists only the filenames and directory names in the specified directory. If you specify the `-l` (long) option, the `ls` command will display additional information about each file:

```
$ ls -l
total 3256
-rw-r--r--@ 1 rhyde  staff   168089 Dec 29  20xx encoder.pdf
-rw-r--r--@ 1 rhyde  staff  1492096 Dec 27  20xx mcp23017.png
```

The first column in the listing specifies the file permissions. The next three provide a link count and ownership information, followed by file size and modification date and time, followed by the filename.

D.5.5 file

Unlike macOS and Windows, Unix does not associate a specific data type with a file. You can use the Unix `file` command to determine a file type for a particular file:

```
file pathname
```

The `file` command will respond with its best guess as to the type of the file specified by *`pathname`*.

D.5.6 cat, less, more, and tail

To view the contents of a text file, you can display that file in its entirety using the `cat` (catenate) command

```
cat pathname
```

where *pathname* is the path to the name of the file you wish to display.

The problem with `cat` is that it tries to write the entire file to the display, all at once. Many files are larger than can be displayed on the screen at one time, so `cat` ends up displaying only the last few lines of the file; moreover, very large files may take a while to display their contents. If you would like to be able to page through a file one screenful at a time, you can use the `more` and `less` commands:

```
more pathname
less pathname
```

The `more` command is now obsolete but is still available to handle older script files that contain it. It displays a pageful of text and allows you to scroll through the file a line at a time (by pressing ENTER) or a page at a time (by pressing the spacebar). The big drawback to `more` is that you can view only forward in a file; after information scrolls off the screen, it is lost.

The `less` command (whose name comes from the phrase *less is more*) is an upgraded version of `more` that allows you to scroll forward and backward in a page. Most people use the `less` command rather than `more` because of the additional features (such as being able to use the arrow keys to consistently scroll up and down a line at a time).

If you want to view only the last few lines of a large file, use the `tail` command:

```
tail pathname
```

By default, `tail` prints the last 10 lines of the file. You can use the `-n` *xxxx* command line option, where *xxxx* is a decimal numeric value, to specify a different line count. For example

```
tail -n 20 x.txt
```

displays the last 20 lines of the file *x.txt*.

D.5.7 mv

The `mv` (move) command has the following syntax:

```
mv SourcePath DestinationPath
```

SourcePath is the pathname of the file you want to move or rename, and *DestinationPath* is the final destination path where you want the file moved (or the new name you want to use for the file).

To rename a file in the current directory, `mv` takes the form

```
mv OldName NewName
```

where *OldName* is the existing filename you want to change and *NewName* is the new filename you want to rename the file. These are both simple filenames (no directory path components). Note that *NewName* must be different from *OldName*.

To move a file from one directory to another, either the *SourcePath* or *DestinationPath* (or both) must contain a directory component. The *SourcePath* must contain a filename component at the end of the pathname (the name of the file to move). For the *DestinationPath*, a filename at the end is optional. If the *DestinationPath* is the name of a directory (rather than a file), `mv` will move the source file into the destination directory and use the same filename as the original source file. If there is a filename at the end of *DestinationPath*, then `mv` will change the filename while it is moving it.

You can use wildcard characters with `mv`, subject to the following restrictions: wildcard characters may appear only in the source pathname, and the destination path must be a directory, not an actual filename.

D.5.8 cp

The `cp` command has the following syntax:

```
cp SourcePath DestinationPath
```

This command will make a copy of the file specified by *SourcePath*, using the name *DestinationPath* for the copy. If both pathnames are simple filenames (that is, you're making a copy of a file in the current directory), the two filenames must be different.

The `cp` command accepts wildcard characters in the source operand. If wildcard characters are present, the destination must be a directory path. The `cp` command will copy all files matching the wildcard designation to the specified directory.

If both source and destination operands specify a directory, use the `-R` (recursive) command line option. This will copy all the files from the source directory to a directory by the same name in the destination directory (creating the new directory in the destination if it is not already present); it will also recursively copy any subdirectories in the source directory into similarly named subdirectories in the destination.

D.5.9 rm

The `rm` command removes (deletes) a file from a directory, using the following syntax:

```
rm pathname
```

The *pathname* argument must be a path to an individual file, not a directory. To delete a directory and all the files in it, use the following command:

```
rm -R DirectoryPath
```

This will recursively delete all files and subdirectories in *DirectoryPath*, then delete the directory specified by *DirectoryPath*.

To delete all the files in a directory without removing the directory itself, use the following command:

```
rm -R DirectoryPath/*
```

Be very careful when using wildcard characters in an `rm` command. Depending on the current working directory, the following command could delete everything on your storage devices:

```
rm -R *
```

There is also an `rmdir` command you can use to remove empty directories. However, the `rm -R` *`directory`* command is easier to use for this purpose.

D.5.10 mkdir

The `mkdir` command creates a new (empty) directory, using the syntax

```
mkdir DirectoryPath
```

where *`DirectoryPath`* specifies the pathname to a directory that does not already exist. If *`DirectoryPath`* is an actual pathname, all subdirectory names up to the final name in the path must exist; the final directory name (after the last /) must not exist. If you specify a simple directory name (no path), bash will create the directory in the current working directory.

The `mkdir` command supports a `-p` command line option that will create all nonexistent directories in the path.

D.5.11 date

The `date` command displays the current date and time. You can also use this command to set the Unix real-time clock. Run `man date` for the details.

D.5.12 echo

The `echo` command prints the text on the remainder of the command line (subject to some expansions by bash) to the standard output device. For example

```
echo hello, world!
```

will write `hello, world!` to the standard output. You'll use this command most often in scripts or to display the value of various shell variables.

D.5.13 chmod

Although Unix files do not have a specific type, the directory does maintain whether a file is readable, writable, or executable by the owner of the file, a

group associated with the file, or anyone (standard Unix permissions). The `chmod` command allows you to set (or clear) permission mode bits for a particular file.

The basic syntax for `chmod` is

```
chmod options pathname
```

where *`pathname`* is the path to the file whose mode you want to change, and the *`options`* parameter(s) specifies the new permissions.

The *`options`* argument is either an octal (base-8) number (typically three digits) or a special string to set the permissions. Unix has three permission categories: owner/user, group, and other. The *owner* category applies to the user who created the file in the first place. The *group* category covers any groups that the user belongs to (and other users may belong to). The *other* category covers everyone else.

In addition to the three categories, Unix has three main types of permissions: permission to *read* a file, permission to *write* data to a file (or delete it), and permission to *execute* a file (this generally applies to object code or shell scripts).

A typical `chmod` option consists of a string of one to three characters from the set `{ugo}` followed by a plus or minus character (`+` or `-`, not ±), followed by a single character from the set `{rwx}`. For example, `u+r` enables user read access, `u+x` enables execution permission, and `ugo-x` removes execution privileges for all categories. Note that the `ls -l` command will list the user, group, and other permissions for a given file.

You can also specify the permissions for the three categories as a three-digit octal number, where each digit represents the three `rwx` bits for users (HO digit), groups (middle digit), and other (LO digit). For example, `755` specifies read/write/execute permissions for the user ($111_2 = 7_8$), read and execute permissions for the group, and other ($101_2 = 5_8$). Note that `755` is a typical set of permissions you would assign to a publicly usable script file.

D.6 Shell Scripts

A *shell script* is a text file that bash will interpret as a sequence of commands to execute, exactly as though each line in the text file were entered from the keyboard while running bash. For those with Microsoft Windows experience, this is similar to a batch file. This section discusses using shell variables and values, using special built-in shell variables, and creating your own bash shell scripts.

The bash interpreter is a full-fledged programming language that supports conditional and looping constructs as well as the sequential execution of commands on the command line. It supports `if...elif...else` statements, a `case` statement (similar to C's `switch` statement), and various loops (`while`, `for`, and so on). It also supports functions, local variables, and other features typically found in HLLs. Going into detail on these topics is beyond the scope of this book. See Ryan's Tutorials in section D.8, "For More Information," on page 968 for details beyond what this section covers.

D.6.1 Defining Shell Script Variables and Values

Bash allows you to define shell variables. A *shell variable* is a name (think: programming language identifier) to which you can assign some text. For example, the following bash command assigns the text `ls` to `list`:

```
list=ls
```

You can tell bash to expand a shell variable name to its associated text by prefixing the name with a `$` character. For example

```
$list
```

expands to

```
ls
```

which will display the current directory listing.

Normally, you would not use shell variables to create aliases of existing commands, as the `alias` command is better suited to the job. Instead, you would use shell variables to keep track of paths, options, and other information commonly used on the command line.

Bash supplies several predefined shell variables, including the following:

`$HOME` Contains the path to the current user's home directory

`$HOSTNAME` Contains the machine's name

`$PATH` Contains a list of directory paths, separated by colons (`:`), that bash will search through when searching for an external command's executable file

`$PS1` Contains a string that bash will print as the command line prompt

`$PWD` Contains the current working directory

For a complete list of predefined shell variables, and for more details on the `$PS1` variable in particular, see section D.8, "For More Information," on page 968.

Shell variables to which you assign values will retain those values during the execution of the current bash shell. Often, when executing shell scripts, a second bash shell begins execution, and any variable values created or modified in that execution will be lost when that shell terminates. To avoid this problem, use the built-in `export` command:

```
export variable name=value
```

This command will make the variable assignment visible to parent shells. Typically, you must use `export` when assigning values in a shell script that you want to retain after the shell script completes.

You can define shell variables in a script file, just as you can when interactively typing commands at the command line. As noted earlier, however,

any variable values defined in a shell script are lost when the shell terminates. This is because bash will make a copy of the execution environment (including all the shell variable values) when it starts a shell script. Any changes or additions you make, such as creating new variables or modifying existing ones, affect only the copy of the environment. When the shell script terminates, it deletes the copy of the environment and reverts to the original environment. The `export` command tells bash to export the variable assignment to the parent environment (as well as applying to the current local environment).

Values assigned to a shell variable are generally treated as text. Because the bash interpreter breaks up command lines by using spaces or other delimiters, the string of text you assign to a script variable must consist of a single *word* (that is, a sequence of characters surrounded by delimiters). If you want to include delimiter (and other) characters within the value, you must surround the text value with quotes or apostrophes, as in the following example:

```
value1="Value containing delimiters (spaces)"
value2='Another value with delimiters'
```

Bash will expand the text inside double quotes (") and will maintain the text as is inside single quotes ('). Consider the following example:

```
aVariable="Some text"
value3="aVariable=$aVariable"
value4='aVariable=$aVariable'
echo $value3
echo $value4
```

Executing this sequence will produce the following output:

```
aVariable=Some Text
aVariable=$aVariable
```

Bash expands `$aVariable` inside the string enclosed in double quotes but will not expand it inside the string enclosed in single quotes.

You might see strings surrounded by grave accent characters (`), commonly called *backticks* in Unix. Originally, such strings surrounded a command that the shell would execute, then substitute the textual output of the program in place of the backticked string. This syntax has been deprecated in modern shells. To capture the output of a command and assign it to a variable, use `$(`*command*`)`, as in the following example:

```
dirListing=$(ls)
```

This creates a string consisting of the listing of the current working directory and assigns that string to the `dirListing` variable.

D.6.2 Defining Special Shell Variables

In addition to the shell variables a script inherits from the parent environment, bash also defines certain shell variables that might prove useful in shell scripts you write. These special variables begin with `$` and typically deal with command line parameters passed to the script (see Table D-1).

Table D-1: Special Shell Variables

Variable	Description
`$0`	Expands to the pathname of the shell script file.
`$1` through `$`*n*	Expands to the first, second, . . . , *n*th command line argument.
`$#`	Expands to a decimal number specifying the parameter count.
`$*`	Expands to a string containing all the command line parameters. Generally used to pass the parameters on to another command. To assign this command line parameter-list string to a shell variable, use $* to capture the text as a single string.
`$@`	Similar to $*, except this variant puts quotes around each argument. Useful if the original arguments may have been quoted and could contain spaces or other delimiter characters. It's best to invoke this as $@ as well.

See the references in section D.8, "For More Information," on page 968 for more details on these features.

D.6.3 Writing Your Own Shell Scripts

Consider the following text, from a file named *lsPix*:

```
cd $HOME/Pictures
ls
```

If you were to execute this shell script via the following command, bash would switch to the *Pictures* subdirectory in the user's home directory, display the directory listing, and the return control to bash:

```
bash lsPix
```

Entering `bash` in front of a shell script to execute it can become annoying if you commonly execute certain shell scripts. Fortunately, Unix (via shells such as sh, bash, or zsh) provides a mechanism to specify the shell script directly as a command: make the script file executable. You can use the `chmod` command to accomplish this:

```
chmod 755 lsPix
```

This sets the permissions to RWX (readable, writable, and executable) for the owner and R-X (readable and executable) for members of the group and other users.

Note that the *build* script (used throughout this book) has been made executable via a `chmod 777 build` command (this allows everyone to modify the file). That is why you need to enter only `./build` at the beginning of the command line rather than `bash build`.

When making a bash shell script executable, also add the following statement to the beginning of the shell script file:

```
#! /bin/bash
```

The *shebang* (`#!`) sequence tells bash that this is a shell script and provides the path to the shell interpreter to execute for this command. (The interpreter would be bash in this case, but you could specify a different shell interpreter, such as `/bin/sh`, if you really wanted.) If you don't know the path to the bash interpreter, execute the Unix command `which bash` to print the path you need. The inclusion of the shebang on the first line also allows the `file lspix` command to identify the file as a shell script rather than a simple ASCII text file.

Once you've added this line to *lsPix* and made the file executable, you need to enter only the following at the command line to execute the script:

```
./lsPix
```

The `#` character is normally used to create comments in a shell script. With the exception of the shebang in the first line, the bash interpreter will ignore all the text from a `#` symbol to the end of the line.

It's important to understand that shell scripts execute in their own copy of bash. Therefore, any changes they make to the bash environment—such as setting the current working directory with the `cd` command or changing shell variable values—are lost when the script terminates. For example, when this *lsPix* script terminates, the current working directory will return to its original directory; it will not be *$HOME/Pictures* (unless it happened to be *$HOME/Pictures* prior to executing *lsPix*).

D.7 The build Script

Shell scripts are useful for automating manual activities. For example, this book uses the *build* script to assemble/compile most of the example programs. The following listing presents the *build* script and describes how it works; I'll explain the full script section by section.

As with any good shell script, the build script begins with the shebang that defines the shell interpreter to use (bash in this case):

```
#!/bin/bash
#
# build
#
# Automatically builds an Art of ARM Assembly
# example program from the command line.
```

```
#
# Usage:
#
#   build {options} fileName
#
# (no suffix on the filename.)
#
# options:
#
#   -c: Assemble .S file to object code only.
#   -pie: On Linux, generate a PIE executable.

fileName=""
compileOnly=" "
pie="-no-pie"
cFile="c.cpp"
lib=" "
```

The script also defines several variables (`filename`, `compileOnly`, `pie`, `cFile`, and `lib`) that it will use to specify GCC command line options when assembling and compiling the source files.

The next section of the script processes command line parameters found on the `build` command line:

```
❶ while [[ $# -gt 0 ]]
do

    key="$1"
    case $key in

        -c)
        compileOnly='-c'
      ❷ shift
        ;;

        -pie)
        pie='-pie'
        shift
        ;;

        -math)
        math='-lm'
        shift
        ;;

        *)
        fileName="$1"
        shift
        ;;
    esac
done
```

The `while` loop processes each command line argument individually ❶. The Boolean expression `$# -gt 0` returns true as long as there are one or more command line arguments (`$#` is the number of arguments).

The body of the loop sets the key local variable equal to the value of the first command line parameter (`$1`). It then executes a `case` statement that compares this argument against the options `-c`, `-pie`, and `-math`. If the argument matches one of these, the script sets appropriate local variables to values that note the presence of these options. If the case expression doesn't match any of these, the default case (*) sets the filename variable to the value of the command line argument.

At the end of each case, you'll notice a shift statement ❷. This statement shifts all the command line parameters to the left one position (deleting the original `$1` argument), setting `$1 = $2`, `$2 = $3`, . . . and decrementing the parameter count (`$#`) by one. This sets up the `while` loop for the next iteration to process the remaining command line parameters.

The next section sets up the `objectFile` variable that the script expands as part of the `gcc` command line:

```
# If -c option was provided, assemble only the .S
# file and produce a .o output file.
#
# If -c not specified, compile both c.cpp and the .S
# file and produce an executable:

if [ "$compileOnly" = '-c' ]; then
    objectFile="-o $fileName".o
    cFile=" "
else
    objectFile="-o $fileName"
fi
```

This code sets `objectFile` to a string that will specify an object filename on the `gcc` command line. If `-c` is not present, this code will set `cFile` to expand to an empty string so that the `gcc` command does not also compile *c.cpp* (the default case).

The following section of the *build* script deletes any existing object files or executable files that this command would create:

```
# If the executable already exists, delete it:

if test -e "$fileName"; then
    rm "$fileName"
fi

# If the object file already exists, delete it:

if test -e "$fileName".o; then
    rm "$fileName".o
fi
```

The `test` built-in function returns true if the specified file exists. Therefore, these `if` statements will delete the object and executable files if they already exist.

Next, the *aoaa.inc* header file requires the definition of either the `isLinux` or `isMacOS` symbols in order to determine the OS. The definition of these symbols allows *aoaa.inc* to select OS-specific code so that the example code will compile (portably) across the two OSes. Rather than force the user to manually define this symbol, the *build* script automatically defines one of these symbols when invoking GCC. To accomplish this, *build* uses the `uname` command, which returns the name of the OS kernel:

```
# Determine what OS you're running under (Linux or Darwin [macOS]) and
# issue the appropriate GCC command to compile/assemble the files:

unamestr=$(uname)
```

Under Linux, `uname` returns the string `Linux`; under macOS, it returns the string `Darwin`.

Finally, the *build* script invokes the GCC compiler with command line arguments appropriate to the OS:

```
if [ "$unamestr" = 'Linux' ]; then
    gcc -D isLinux=1 $pie $compileOnly $objectFile  $cFile $fileName.S $math
elif [ "$unamestr" = 'Darwin' ]; then
    gcc -D isMacOS=1  $compileOnly $objectFile $cFile  $fileName.S -lSystem $math
fi
```

Note the `-D` *`name`*`=1` command line option that defines the `isLinux` or `isMacOS` symbols as appropriate. Also note that the `pie` (position-independent code) option appears only when compiling under Linux, as macOS code is always position-independent.

It's easy to modify the *build* script to add more features, should you desire. For example, one limitation to this script is that it allows you to specify only a single assembly language source file (if you specify two or more names, it will use only the last name you specify). You can change this with three modifications to the file, in the `case` statement.

The first change is to append the filename and add a `.S` suffix to the `fileName` variable rather than replacing its value. You must also set the executable output filename to the first assembly file specified on the command line:

```
*)
    if[ fileName = "" ]
    then
        objectFile = "$1"
    fi
    fileName="$filename $1.S"
    shift
    ;;
```

The next change is to set `objectFile` to the empty string if specifying compile-only mode:

```
if [ "$compileOnly" = '-c' ]; then
cFile=" "
else
    objectFile="-o $fileName"
fi
```

The original code set this to the specified filename; however, that is the default for compile-only mode, and specifying a single object name when assembling multiple source files is problematic.

The final change is to modify the two `gcc` command lines to remove the `.S` suffix from the assembly filenames (since this was added in the `case` statement):

```
if [ "$unamestr" = 'Linux' ]; then
    gcc -D isLinux=1 $pie $compileOnly $objectFile  $cFile $fileName $math
elif [ "$unamestr" = 'Darwin' ]; then
    gcc -D isMacOS=1  $compileOnly $objectFile $cFile  $fileName -lSystem $math
fi
```

However, if you're going to do a complex assembly using multiple source files, you're probably better off using makefiles rather than shell scripts to do the job.

D.8 For More Information

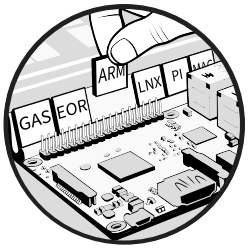
- For details on Unix regular expressions and wildcards, see the Bash Reference Manual at *https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html*.
- For more information on bash shell scripts, check out *The Linux Command Line*, 2nd edition, by William Shotts (No Starch Press, 2019).
- For an explanation of the risks of including `./` in your execution path, see the Unix & Linux Stack Exchange question at *https://unix.stackexchange.com/questions/65700/is-it-safe-to-add-to-my-path-how-come*.
- A complete list of bash shell script variables appears at the Advanced Bash Scripting Guide site, *https://tldp.org/LDP/abs/html/internalvariables.html*.
- For details on changing the command line prompt, see the phoenixNap site at *https://phoenixnap.com/kb/change-bash-prompt-linux*.

Here are some websites that describe how to write bash scripts:

- freeCodeCamp: *https://www.freecodecamp.org/news/shell-scripting-crash-course-how-to-write-bash-scripts-in-linux*
- Ryan's Tutorials: *https://ryanstutorials.net/bash-scripting-tutorial/bash-script.php*
- Linux Hint: *https://linuxhint.com*
- Bash scripting cheat sheet: *https://devhints.io/bash*

E

USEFUL C LANGUAGE FUNCTIONS



This appendix contains a list of several C functions from the C stdlib (and Unix system library) that may be useful to assembly language programmers.

The macOS variants of these functions use an external name that begins with an underscore. For example, under macOS, `strlen()` becomes the `_strlen()` function. The *aoaa.inc* header file contains `#define` statements for many of these function names that add the underscore prefix in front of the unadorned names: `#define strlen _strlen`.

E.1 String Functions

Various chapters in this book have introduced many of the C stdlib string functions (declared in the *strings.h* header file). This section describes most of the available functions, including those this book has not used:

`char *strcat( char *dest, const char *src );`

Concatenates the zero-terminated string at which X1 (`src`) points to the end of the string at which X0 (`dest`) points. Returns a pointer to the `dest` string in X0.

`char *strchr( const char *str, int c );`

Searches for the first occurrence of the character specified by `c` (X1) in the string at which `str` (X0) points. Returns a pointer into the string (in X0) where the character was found, or a NULL (`0`) pointer if `c` does not exist in `str`.

`char *strcpy( char *dest, const char *src );`

Copies the string pointed at by `src` (X1) to `dest` (X0), including the zero-terminating byte. Returns a pointer to `dest` in X0.

`char *strdup( char *str );`

Duplicates a string on the heap. On entry, X0 contains a pointer to the string to duplicate. On return, X0 contains a pointer to a copy of the string allocated on the heap. When the application is done using the string, the application should call the C stdlib `free()` function to return the storage to the heap. Though `strdup()` is not defined in the C stdlib, most systems include it in their libraries.

`char *strncat( char *dest, const char *src, size_t n );`

Concatenates at most `n` characters from the zero-terminated string at which X1 (`src`) points to the end of the string at which X0 (`dest`) points, plus a zero-terminating byte. Returns a pointer to the `dest` string in X0. If the length of `src` is less than `n`, this string copies only the first `n` characters from `src` to `dest` (plus a zero-terminating byte).

`char *strpbrk( const char *str1, const char *str2 );`

Finds the first character in the string `str1` (passed in X0) that matches any character specified in `str2` (passed in X1). Returns a pointer to the matching character in `str1` in the X0 register (or `NULL` if no match).

`char *strrchr( const char *str, int c );`

Searches for the last occurrence of the character `c` (a char passed in X1) in the string pointed to by the argument `str` (passed in X0). Returns a pointer into `str` where the character was found in X0. If the character was not found in `str`, this function returns NULL (`0`) in X0.

`char *strstr( const char *inStr, const char *search4 );`

Searches for the first occurrence of the string `search4` (passed in X1) within `inStr` (passed in X0). It returns a pointer to the matching

position in the X0 register, returning NULL (0) if the search4 string is not present within inStr.

char *strtok(char *str, char *delim);

Breaks string str (passed in X0) into a series of *tokens* (words) separated by characters found in the delim (passed in X1) string. On a first call, the function expects a C string as an argument for str, whose first character is used as the starting location to scan for tokens. In subsequent calls, the function expects a NULL (0) pointer and uses the position right after the end of the last token as the new starting location for scanning (skipping any leading delimiter characters). Each call returns a pointer (in X0) to the next token within the string. This function returns NULL when it exhausts all the tokens in the string.

This function modifies the contents of the string at which str (X0) points. If your program cannot tolerate this, make a copy of str before calling strtok(). The strtok() function maintains internal state in a static variable and therefore is not safe to use in multithreaded applications.

int memcmp(void *mem1, void *mem2, size_t n);

Compares the first n bytes of mem1 and mem2 (mem1 is passed in X0, mem2 is passed in X1, and n is passed in X2). Similar in operation to strcmp(), except that this function doesn't end the comparison upon encountering a 0 byte; strcmp(), on the other hand, returns a negative value, 0, or a positive value to indicate the comparison status.

int strcasecmp(const char *str1, const char *str2);

Compares the string at which str1 (X0) points against the string at which str2 (X1) points using a case-insensitive comparison. Returns (in X0) a negative number if str1 < str2, 0 if str1 == str2, or a positive number if str1 > str2. Though strcasecmp() is not defined in the C stdlib, many systems include it in their libraries; some use the function name strcmpi() or stricmp() instead.

int strcmp(const char *str1, const char *str2);

Compares the string at which str1 (X0) points against the string at which str2 (X1) points and returns (in X0) a negative number if str1 < str2, 0 if str1 == str2, or a positive number if str1 > str2.

int strncmp(char *str1, char *str2, size_t n);

Compares two strings up to the first n characters, or until encountering the first zero-terminating byte (in either string). Pointer to str1 is passed in X0, pointer to str2 is passed in X1, and n is passed in X2. Returns 0 if the strings were equal (through n characters, or less if both strings are equal and their length is less than n). Returns a negative value if str1 is less than str2. Returns a positive value if str1 is greater than str2. You can use this function to see if str1 is a prefix of str2 by setting n equal to the length of str1.

size_t strcspn(const char *str1, const char *str2);

Calculates the length of the initial segment of `str1` (passed in X0), which consists entirely of characters not in `str2` (passed in X1). Returns this count in X0.

size_t strlen(char *str);

Computes the length of a zero-terminated string. X0 contains a pointer to the string upon entry, and this function returns the string length in X0 (not including the zero-terminating byte).

size_t strspn(const char *str1, const char *str2);

Calculates the length of the initial segment of `str1` (passed in X0), which consists entirely of characters in `str2` (passed in X1). Returns the count in X0.

strlwr(str);

Converts all the characters in a string to lowercase. On entry, X0 contains a pointer to the string to convert; on return, X0 points at this same string with the uppercase characters converted to lowercase. Though `strlwr()` is not defined in the C stdlib, many systems include it in their libraries.

strncpy(char *dest, const char *src, size_t n);

Copies, at most, `n` (passed in X2) characters from `src` (passed in X1) to `dest` (passed in X0). If `n` is less than or equal to the length of `src`, this function will not copy the zero-terminating byte, and the caller is responsible for adding this extra byte. This function has two primary uses. First, it prevents overwriting data beyond the end of `dest` (when `n` contains the size of the `dest` buffer, plus 1, at which X0 points). Second, it serves as a substring function, allowing you to extract `n` characters from a particular position within a string.

strupr(str);

Converts all lowercase characters in a string to uppercase. On entry, X0 contains a pointer to the string to convert; on return, X0 points at this same string with the lowercase characters converted to uppercase. Though `strupr()` is not defined in the C stdlib, many systems include it in their libraries.

void *memchr(void *mem, int c, size_t n);

Searches for the first occurrence of the character `c` (an unsigned char passed in X1) in the first `n` (passed in X2) bytes of the memory block at which the argument `mem` (passed in X0) points. Very similar to `strchr()`, except this function will not stop scanning when it finds a 0 byte in `str`. Returns, in X0, a pointer into `mem` where it found the character, or NULL (`0`) if character `c` does not exist in `mem`.

`void *memcpy( void *dest, const void *src, size_t n );`

Copies `n` bytes from `src` to `dest` (passed in X2, X1, and X0, respectively). Returns a pointer to `dest` in X0. If the memory block defined by `dest` overlaps the memory block defined by `src`, the results are undefined.

`void *memmove( void *dest, const void *src, size_t n );`

Copies `n` bytes from `src` to `dest` (passed in X2, X1, and X0, respectively). Returns a pointer to `dest` in X0.

The `memmove()` function correctly handles situations in which the source and destination blocks overlap. However, this function may run a little bit slower than `memcpy()`, so you should use it only when you cannot guarantee that the blocks do not overlap.

`void *memset( void *mem, int c, size_t n );`

Copies the LO byte of `c` (passed in X1) to the first `n` (passed in X2) bytes of the memory block at which the argument `mem` (passed in X0) points. Returns a pointer to the memory block in X0.

E.2 Other C Stdlib and Unix Functions

The string functions covered in this appendix are but a small sampling of the many functions available in the C stdlib. Other useful functions include the POSIX file I/O functions (declared in the *fcntl.h* and *unistd.h* header files), the math libraries (found in *math.h*), and many others. For more information on these header files, see the following:

fcntl.h

https://pubs.opengroup.org/onlinepubs/000095399/basedefs/fcntl.h.html

math.h

https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/math.h.html

unistd.h

https://pubs.opengroup.org/onlinepubs/007908775/xsh/unistd.h.html

You can easily call each of these functions by specifying its name (don't forget to prepend an underscore when calling functions in macOS). You always pass parameters and retrieve function results by using the ARM ABI for Linux and the macOS ABI under macOS (remember that macOS differs when passing variable argument lists to a function, such as `printf()`).

F

ANSWERS TO QUESTIONS

F.1 Chapter 1

1. *as*
2. address, data, and control
3. The PSTATE register
4. (a) 4, (b) 8, (c) 16, (d) 8
5. 64 bits
6. `bl`
7. `ret`
8. Application binary interface
9. (a) LO byte of W0, (b) LO hword of W0, (c) W0, (d) X0, (e) X0
10. X0, X1, X2, and X3 registers (respectively)

F.2 Chapter 2

1. $9 \times 10^3 + 3 \times 10^2 + 8 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} + 7 \times 10^{-2} + 6 \times 10^{-3}$
2. (a) 10, (b) 12, (c) 7, (d) 9, (e) 3, (f) 15

3. (a) A, (b) E, (c) B, (d) D, (e) 2, (f) C, (g) CF, (h) 98D1
4. (a) 0001_0010_1010_1111
   (b) 1001_1011_1110_0111
   (c) 0100_1010
   (d) 0001_0011_0111_1111
   (e) 1111_0000_0000_1101
   (f) 1011_1110_1010_1101
   (g) 0100_1001_0011_1000
5. (a) 10, (b) 11, (c) 15, (d) 13, (e) 14, (f) 12
6. (a) 32, (b) 128, (c) 16, (d) 64, (e) 4, (f) 8, (g) 4
7. (a) 4, (b) 8, (c) 16, (d) 2
8. (a) 16, (b) 256, (c) 65,536, (d) 2
9. 4
10. 0 through 7
11. Bit 0
12. Bit 63
13. (a) 0, (b) 0, (c) 0, (d) 1
14. (a) 0, (b) 1, (c) 1, (d) 1
15. (a) 0, (b) 1, (c) 1, (d), 0
16. XORing with 1 (bitwise, all 1 bits in a register)
17. AND
18. OR
19. NOT (XOR with all 1 bits too)
20. XOR
21. `not` (`eor` with all 1 bits too)
22. 1111_1011
23. 0000_0010
24. (a) 1111_1111b, (c) 1000_0000b, (e) 1000_0001b
25. The `neg` instruction
26. (a) 1111_1111_1111_1111
    (c) 000_0000_0000_0001
    (d) 1111_1111_1111_0000
27. `b` (`b.al`)
28. *`label:`*
29. Negative/sign (N), zero (Z), carry (C), and overflow (V)
30. Z = 1
31. C = 0 and Z = 0
32. `bhi`, `bhs`, `bls`, `blo`, `beq`, and `bne` conditional jump instructions

33. `bgt`, `bge`, `blt`, `ble`, `beq`, and `bne` conditional jump instructions
34. The `lsl` instruction does not affect the zero flag.
35. A multiplication by 2
36. A division by 2
37. Multiplication and division
38. A normalized floating-point value has a 1 bit in the HO mantissa position.
39. 7 bits
40. 0x30 through 0x39
41. The apostrophe (or single quotation mark) character

F.3 Chapter 3

1. The PC 64-bit register
2. Operation code, the numeric encoding for a machine instruction
3. Static/scalar variables and memory-based constants
4. About ±1 MB, with the `ldr` and `str` instructions
5. The address of the memory location to access
6. (b) X0 and (d) SP
7. The `lea` macro (or `adr` and `adrp` instructions)
8. The final address obtained after all addressing mode calculations are completed
9. Use the `.align 3` directive to align a variable in the `.data` section to an 8-byte boundary.
10. Memory management unit
11. An arithmetic expression that computes the (static) runtime address of a memory object
12. A big-endian value stores high-order portions of the value in lower memory addresses, while a little-endian value stores the low-order portions of the value in lower memory addresses.
13. The `rev32` instruction
14. The `rev16` instruction
15. The `rev` instruction
16. Subtract 16 from SP, then store the value in X0 at the memory address pointed at by SP.
17. Load X0 from the address pointed at by SP, then add 16 to the SP register.
18. Reverse
19. Last-in, first-out

F.4 Chapter 4

1. A symbolic name for a constant that the assembler (or preprocessor) will replace with the numeric equivalent of that constant during assembly
2. Use the `.equ`, `.set`, and `=` directives. You can also use the C preprocessor (CPP) `#define` directive if your source file's name ends with *.S*.
3. A constant expression is an arithmetic expression that Gas can compute during assembly. You determine the number of data elements in the operand field of a byte directive by counting the expressions separated by commas.
4. The current offset into a section (such as `.data` or `.text`)
5. The period operator (`.`)
6. Subtract the label of the second declaration from the label of the first (for example, *`second - first`*).
7. A 64-bit memory variable containing the address of another memory object; you would use a `.dword` directive to allocate storage for a pointer (or other mechanism to reserve 64 bits).
8. Load that pointer into a 64-bit register and use the register-indirect addressing mode to access memory.
9. Use the `.dword` directive.
10. Using an uninitialized pointer; using a pointer that contains an illegal value; continuing to use allocated data after it has been freed (a *dangling pointer*); failing to free memory after you are done using it (a *memory leak*); accessing indirect data by using the wrong data type
11. A pointer to allocated memory that has already been freed
12. A memory leak occurs when a program allocates memory (using `malloc()`) but fails to free that storage when it is done using it.
13. An object composed of (made up from) a collection of other data types
14. A sequence of characters delimited by a zero value (typically a byte)
15. A sequence of characters beginning with a length value (typically a byte, but it could also be a half word, word, or other type)
16. A structure that describes a string object, typically containing length information and a pointer to the string
17. A sequence of objects (all the same type) appearing in consecutive memory locations
18. The address of the first element, typically at the lowest address of the array in memory
19. Here is a typical array declaration using Gas:

```
anArray .space 256, 0  // 256 bytes, all initialize to 0
```

20. You will typically use a directive such as `.word` with a list of the initial element values; here's an example:

```
initializedArray: .word 1, 2, 3, 4, 5, 6, 7, 8
```

You could also use the `.space` directive if you have an array of bytes and every byte is initialized with the same value.

21. (a) Multiply the index by 8 and add the base address of `A` to this product; (b) To access `W[i, j]`, use `address = base(W) + (i * 8 + j) * 4`; (c) To access `R[i, j, k]`, use `address = base(R) + ((i * 4) + j) * 6 + k) * 4`.
22. A mechanism for storing arrays in memory, where elements from each row appear in consecutive memory locations and the rows appear in consecutive blocks of memory
23. A mechanism for storing arrays in memory, where elements from each column appear in consecutive memory locations and the columns appear in consecutive blocks of memory
24. A typical two-dimensional array declaration for word array `W[4,8]` would take this form: `W: .space 4 * 8 * 4, 0`.
25. A composite data type whose elements (fields) need not all be the same type
26. Use statements like the following:

```
struct student
    byte  sName, 65 // Includes zero-terminating byte
    hword Major
    byte  SSN, 12   // Includes zero-terminating byte
    hword Midterm1
    hword Midterm2
    hword Final
    hword Homework
    hword Projects
ends student
```

27. Add the offset of a particular field to the base address of the structure.
28. A type of structure in which all fields occupy the same memory locations
29. For structs, each field is allocated a unique block of memory (according to its size), whereas for a union, all fields are allocated the same memory locations.

F.5 Chapter 5

1. The `bl` instruction copies the address of the next instruction into LR, then transfers control to the target address specified by the operand.
2. The `ret` instruction copies the value from the LR into the program counter.
3. The biggest problem with caller preservation is that it is hard to maintain. It also generates bigger object code files.

4. It saves registers, taking valuable CPU cycles, even if the caller doesn't require those registers to be saved.
5. Storage space in the stack, where a procedure maintains parameters, return addresses, saved register values, local variables, and possibly other data
6. The FP register (X29)
7. The standard entry sequence is as follows:

```
stp fp, lr, [sp, #-16]!   // Save LR and FP values.
mov fp, sp                // Get activation record ptr in FP.
sub sp, sp, #NumVars      // Allocate local storage.
```

8. The standard exit sequence is shown here:

```
mov sp, fp    // Deallocate storage for all the local vars.
ldp fp, lr, [sp], #16  // Pop FP and return address.
ret                    // Return to caller.
```

9. A variable that a procedure automatically allocates and deallocates storage for in an activation record
10. Upon entry into a procedure
11. The parameter's value
12. The parameter's address
13. X0, X1, X2, and X3
14. All parameters beyond the eighth parameter get passed on the stack.
15. Volatile registers can be used by an ARM procedure without preserving their values; nonvolatile registers' values must be preserved across a procedure call.
16. Registers X0, X1, . . . , X15
17. Registers X16 through X31 (SP)
18. A procedure accesses parameters passed in the code stream via the address passed in the LR register.
19. Large parameters (such as arrays and records) should be passed by reference, since the procedure is faster and shorter when using reference arguments.
20. The X0 register (X8 can contain a pointer to a large function return result)
21. The address of a procedure to call, passed as an argument to a procedure or function
22. Call procedural parameters (as well as any procedure via a pointer) by using the `br` instruction.
23. Set aside local storage for the registers and preserve those values in the local storage.

F.6 Chapter 6

1. The `cmp` instruction sets the zero flag if the two operands are equal.
2. The `cmp` instruction sets the carry flag if one unsigned operand (left) is greater than or equal to the other unsigned operand (right).
3. The `cmp` instruction sets negative and overflow flags to opposite values if the left signed operand is less than the signed right operand; they are set to the same value if the left signed operand is greater than or equal to the right signed operand.
4. `x = x + y`:

```
ldr w0, [fp, #x]
ldr w1, [fp, #y]
add w0, w0, w1
str w0, [fp, #x]
```

`x = y - z`:

```
ldr w0, [fp, #y]
ldr w1, [fp, #z]
sub w0, w0, w1
str w0, [fp, #x]
```

`x = y * z`:

```
ldr w0, [fp, #y]
ldr w1, [fp, #z]
mul w0, w0, w1
str w0, [fp, #x]
```

`x = y + z * t`:

```
ldr w0, [fp, #y]
ldr w1, [fp, #z]
ldr w2, [fp, #t]
mul w1, w1, w2
sub w0, w0, w1
str w0, [fp, #x]
```

`x = (y + z) * t`:

```
ldr w0, [fp, #y]
ldr w1, [fp, #z]
add w0, w0, w1
ldr w1, [fp, #t]
mul w0, w0, w1
str w0, [fp, #x]
```

x = -((x * y) / z):

```
ldr  w0, [fp, #x]
ldr  w1, [fp, #y]
mul  w0, w0, w1
ldr  w1, [fp, #z]
sdiv w0, w0, w1
neg  w0, w0
str  w0, fp, #x]
```

x = (y == z) && (t != 0):

```
ldr  w0, [fp, #y]
ldr  w1, [fp, #z]
cmp  w0, w1
cset w0, eq
ldr  w1, [fp, #t]
cmp  w1, #0
cset w1, ne
and  w0, w0, w1
str  w0, [fp, #w]
```

5. x = x * 2:

```
ldr w0, [fp, #x]
lsl w0, w0, #1
str w0, [fp, #x]
```

x = y * 5:

```
ldr w0, [fp, #y]
lsl w1, w0, #2
add w0, w0, w1
str w0, [fp, #x]
```

x = y * 8:

```
ldr w0, [fp, #y]
lsl w0, w0, #3
str w0, [fp, #x]
```

6. x = x / 2:

```
ldr x0, [fp, #x]
lsr x0, #1
str x0, [fp, #x]
```

x = y / 8:

```
ldr x0, [fp, #y]
lsr x0, #3
str x0, [fp, #x]
```

x = z / 10:

```
ldr x0, [fp, #z]
ldr x1, =6554    // 65,536/10
mul x0, x0, x1
lsr x0, x0, #16  // Divide by 65,535.
str x0, [fp, #x]
```

7. x = x + y:

```
ldr  d0, [fp, #x]
ldr  d1, [fp, #y]
fadd d0, d0, d1
str  d0, [fp, #x]
```

x = y - z:

```
ldr  d0, [fp, #y]
ldr  d1, [fp, #z]
fsub d0, d0, d1
str  d0, [fp, #x]
```

x = y * z:

```
ldr  d0, [fp, #y]
ldr  d1, [fp, #z]
fmul d0, d0, d1
str  d0, [fp, #x]
```

x = y + z * t:

```
ldr  d0, [fp, #y]
ldr  d1, [fp, #z]
ldr  d2, [fp, #t]
fmul d1, d1, d2
fadd d0, d0, d1
str  d0, [fp, #x]
```

x = (y + z) * t:

```
ldr  d0, [fp, #y]
ldr  d1, [fp, #z]
fadd d0, d0, d1
ldr  d1, [fp, #t]
fmul d0, d0, d1
str  d0, [fp, #x]
```

x = -((x * y) / z):

```
ldr  d0, [fp, #x]
ldr. d1, [fp, #y]
fmul d0, d0, d1
```

```
ldr  d1, [fp, #z]
div  d0, d0, d1
fneg d0, d0
str  d0, [fp, #x]
```

8. `bb = x < y`:

```
ldr  d0, [fp, #x]
ldr  d1, [fp, #y]
fcmp d0, d1
cset x0, lo  // Less than, ordered
strb w0, [fp, #bb]
```

`bb = x >= y && x < z`:

```
ldr  d0, [fp, #x]
ldr  d1, [fp, #y]
fcmp d0, d1
cset x0, ge  // Greater than or equal, ordered (HS is unordered)
ldr  d1, [fp, #z]
fcmp d0, d1
cset x1, lo // Less than, ordered (LT is unordered)
and  x0, x1
strb w0, [fp, #bb]
```

F.7 Chapter 7

1. Use the `lea` macro to obtain the address of a symbol in the program.
2. `br` reg_{64}
3. A piece of code that keeps track of its execution history by entering and leaving certain states
4. A mechanism for extending the range of a branch instruction
5. Short-circuit Boolean evaluation might not execute code for all the conditions in an expression if it determines the result to be true or false without executing any additional code. Complete Boolean evaluation evaluates the entire expression, even if the result is known after a partial evaluation of the expression.
6. a.

```
ldr  w0, [fp, #x]
ldr  w1, [fp, #y]
cmp  w0, w1
cset w0, eq
ldr  w1, [fp, #z]
ldr  w2, [fp, #y]
cmp  w0, w1
cset w1, hi
```

```
orrs w0, w1
beq  skip

     Do something.

skip:
```

b.

```
ldr  w0, [fp, #x]
ldr  w1, [fp, #y]
cmp  w0, w1
cset w0, ne
ldr  w1, [fp, #z]
ldr  w2, [fp, #t]
cmp  w1, w2
cset w1, lo
ands w0, w1
beq  doElse

    then statements
b.al ifDone

doElse:
    else statements
ifDone:
```

7.

a.

```
ldrsh w0, [fp, #x]
ldrsh w1, [fp, #y]
cmp   w0, w1
bne   skip
ldrsh w1, [fp, #z]
ldrsh w2, [fp, #t]
bge   skip

   Do something.

skip:
```

b.

```
ldrsh w0, [fp, #x]
ldrsh w1, [fp, #y]
cmp   w0, w1
beq   doElse
ldrsh w1, [fp, #z]
ldrsh w2, [fp, #t]
cmp   w1, w2
bge   doElse
```

```
    then statements
b.al  ifDone

doElse:
    else statements
ifDone:
```

8. The following `switch` statements (assume all variables are unsigned 32-bit integers) become the assembly language code:
a.

```
ldr  x0, [fp, #t]
cmp  x0, #3
bhi  default
adr  x1, jmpTbl
ldr  x0, [x1, x0, lsl #3]
add  x0, x0, x1
br   x0

jmpTbl: .dword case0-jmpTbl, case1-jmpTbl, case2-jmpTbl, case3-jmpTbl
```

b.

```
ldr  x0, [fp, #t]
cmp  x0, #2
blo  default
cmp  x0, #6
bhi  default
adr  x1, jmpTbl
ldr  x0, [x1, x0, lsl #3]
add  x0, x0, x1
br   x0

jmpTbl: .dword case2-jmpTbl, default-jmpTbl, case4-jmpTbl
        .dword case5-jmpTbl, case6-jmpTbl
```

9. The following `while` loops get converted to the corresponding assembly code (assuming all variables are signed 32-bit integers):
a.

```
whlLp:
    ldr x0, [fp, #i]
    ldr x1, [fp, #j]
    cmp x0, x1
    bgt endWhl

    Code for loop body

    b.al whlLp
endWhl:
```

b.

```
do...while:

rptLp:
    Code for loop body

    ldr x0, [fp, #i]
    ldr x1, [fp, #j]
    cmp x0, x1
    bne rptLp
```

c.

```
   str  wzr [fp, #i]
forLp:
   ldr  x0, [fp, #i]
   cmp  x0, #10
   bge  forDone

   Code for loop body

   ldr  x0, [fp, #i]
   add  x0, x0, #1
   str  x0, [fp, #i]
   b.al forLp
forDone:
```

F.8 Chapter 8

1.

a.

```
ldp  x0, x1, [fp, #y]
ldp  x2, x3, [fp, #z]
adds x0, x0, x2
adc  x1, x1, x3
stp  x0, x1, [fp, #x]
```

b.

```
ldr  x0, [fp, #y]
ldr  w1, [fp, #y+8]
ldr  x2, [fp, #z]
adds x0, x0, x2
adc  w1, w1, wzr
str  x0, [fp, #x]
str  w1, [fp, #x+8]
```

c.

```
ldr  w0, [fp, #y]
ldrh w1, [fp, #y+4]
ldr  w2, [fp, #z]
ldrh w3, [fp, #z+4]
adds w0, w0, w2
adc  w1, w1, w3
str  w0, [fp, #x]
strh w1, [fp, #x+4]
```

2.

a.

```
ldp  x0, x1, [fp, #y]
ldr  x2, [fp, #y+16]
ldp  x3, x4, [fp, #z]
ldr  x5, [fp, #z+16]
subs x0, x0, x3
sbc  x1, x1, x4
sbc  x2, x2, x5
stp  x0, x1, [fp, #x]
str  x2, [fp, #x+16]
```

b.

```
ldr  x0, [fp, #y]
ldr  w1, [fp, #y+8]
ldp  x2, [fp, #z]
ldr  w3, [fp, #z+8]
subs x0, x0, x2
sbc  w1, w1, w3
str  x0, [fp, #x]
str  w1, [fp, #x+8]
```

3.

```
            ldr     x0, [fp, #y]
            ldr     x1, [fp, #y + 8]
            ldr     x2, [fp, #z]
            ldr     x3, [fp, #z + 8]
// X5:X4 = X0 * X2

            mul     x4, x0, x2
            umulh   x5, x0, x2

// X6:X7 = X1 * X2, then X5 = X5 + X7 (and save carry for later):

            mul     x7, x1, x2
            umulh   x6, x1, x2
            adds    x5, x5, x7

// X7 = X0 * X3, then X5 = X5 + X7 + C (from earlier):
```

```
            mul     x7, x0, x3
            adcs    x5, x5, x7
            umulh   x7, x0, x3
            adcs    x6, x6, x7  // Add in carry from adcs earlier.

// X7:X2 = X3 * X1


            mul     x2, x3, x1
            umulh   x7, x3, x1

            adc     x7, x7, xzr  // Add in C from previous adcs.
            adds    x6, x6, x2   // X6 = X6 + X2
            adc     x7, x7, xzr  // Add in carry from adds.

// X7:X6:X5:X4 contains 256-bit result at this point, ignore overflow:

            stp     x4, x5, [fp, #x]   // Save result to location.
```

4. The conversions are as follows:
   a.

```
ldp x0, x1, [fp, #x]
ldp x2, x3, [fp, #y]
cmp x0, x2
bne isFalse
cmp x1, x3
bne isFalse

 Code

isFalse:
```

   b.

```
ldp x0, x1, [fp, #x]
ldp x2, x3, [fp, #y]
cmp x1, x3
bhi isFalse
blo isTrue
cmp x1, x3
bhs isFalse

isTrue:
 Code

isFalse:
```

   c.

```
ldp x0, x1, [fp, #x]
ldp x2, x3, [fp, #y]
cmp x1, x3
```

```
 blo isFalse
 bhi isTrue
 cmp x1, x3
 bls isFalse

isTrue:
 Code

isFalse:
```

d.

```
 ldp x0, x1, [fp, #x]
 ldp x2, x3, [fp, #y]
 cmp x1, x3
 bne isTrue
 cmp x1, x3
 beq isFalse

isTrue:
 Code

isFalse:
```

5. The conversions are as follows:
   a.

```
 ldp  x0, x1, [fp, #x]
 subs x0, xzr, x0
 sbc  x1, xzr, x1
 stp  x0, x1, [fp, #x]
```

   b.

```
 ldp  x0, x1, [fp, #y]
 subs x0, xzr, x0
 sbc  x1, xzr, x1
 stp  x0, x1, [fp, #x]
```

6. The conversions are as follows:
   a.

```
 ldp x0, x1, [fp, #y]
 ldp x2, x3, [fp, #z]
 and x0, x0, x2
 and x1, x1, x3
 stp x0, x1, [fp, #x]
```

   b.

```
 ldp x0, x1, [fp, #y]
 ldp x2, x3, [fp, #z]
```

```
orr x0, x0, x2
orr x1, x1, x3
stp x0, x1, [fp, #x]
```

c.

```
ldp x0, x1, [fp, #y]
ldp x2, x3, [fp, #z]
eor x0, x0, x2
eor x1, x1, x3
stp x0, x1, [fp, #x]
```

d.

```
ldp x0, x1, [fp, #y]
not x0, x0
not x1, x1
stp x0, x1, [fp, #x]
```

e.

```
ldp  x0, x1, [fp, #y] // The easy way
adds x0, x0, x0
adc  x1, x1, x1
stp  x0, x1, [fp, #x]
```

f.

```
ldp x0, x1, [fp, #y]  // The easy way
ror x2, x1, #1
and x2, x2, #1 << 63
lsr x0, x0, #1
orr x0, x0, x2
lsr x1, x1, #1
stp x0, x1, [fp, #x]
```

7.

```
ldp x0, x1, [fp, #y]  // The easy way
ror x2, x1, #1
and x2, x2, #1 << 63
lsr x0, x0, #1
orr x0, x0, x2
asr x1, x1, #1
stp x0, x1, [fp, #x]
```

8.

```
ldp  x0, x1, [fp, #x]  // The easy way
adcs x0, x0, x0
adcs x1, x1, x1
stp  x0, x1, [fp, #x]
```

F.9 Chapter 9

1. Four output digits
2. Call `qToStr` twice, the first time passing in the HO dword, the second time passing in the LO dword.
3. Take the input value and see if it is negative. If so, emit a dash (-) character and negate the value. Whether the number is negative or nonnegative, call the unsigned conversion function to do the rest of the work.
4. The `u64toSizeStr` function expects a pointer to the destination buffer in X0, the value to convert to a string in X1, and the minimum field width in X3.
5. The function will output however many characters are necessary to correctly represent the value.
6. The `r64ToStr` function expects the floating-point value to convert in D0, a pointer to the buffer in X0, the field width in X1, the number of digits after the decimal point in X2, the padding character in the LO byte of X3, and the maximum string length in X4.
7. A string `fWidth` characters long, containing the `#` character if it cannot properly format the output
8. D0 contains the value to convert; X0 contains the address of the output buffer; X1 contains the field width; X2 is the padding character; X3 contains the number of exponent digits; X4 is the maximum string width.
9. A character used to begin, end, and separate input values
10. Overflow and illegal input characters

F.10 Chapter 10

1. The set of legal input values
2. The set of possible output values
3.

a.

```
// Assume "input" passed in X0.
lea  x1, f        // Lookup table
ldrb w0, [x1, x0] // Function result is left in W0.
```

b.

```
// Assume "input" passed in X0.
lea  x1, f        // Lookup table
ldrh w0, [x1, x0, uxtw #1 ] // Function result is left in W0.
```

c.

```
// Assume "input" passed in X0.
 lea  x1, f        // Lookup table
 ldrb w0, [x1, x0 ] // Function result is left in W0.
```

d.

```
 // Assume "input" passed in X0.
 lea  x1, f        // Lookup table
 ldr w0, [x1, x0, uxtw #2 ] // Function result is left in W0.
```

4. The process of adjusting the input value to a function so that the minimum and maximum values are limited, in order to allow the use of smaller tables
5. Because memory access is so slow relative to computational performance

F.11 Chapter 11

1. A lane is an element of a byte, half-word, word, or dword array held within a vector register. When operating on a pair of vector registers, lanes are corresponding elements within the two vectors.
2. A scalar instruction operates on a single piece of data, while a vector instruction operates on multiple lanes (pieces of data) within a vector register.
3. The `fmov S`*d*`, W`*s* instruction
4. The `fmov D`*d*`, X`*s* instruction
5. The `tbl` or `tbx` instruction
6. The mov `V`*d*`.`*t*`[`*index*`], `*Rs* instruction (*Rn* = `X`*n* or `W`*n*)
7. The `shl V`*d*`.2D, V`*s*`.2D, #`*n* instruction
8. A vertical addition adds the corresponding lanes from two vector registers together, while a horizontal addition adds adjacent lanes in a single vector register together.
9. Use the `movi v0.16B, #0` instruction.
10. Use the `movi v0.16B, #0xff` instruction.

F.12 Chapter 12

1. The `and` and `bic` instructions
2. The `bic` instruction
3. The `orr` instruction
4. The `eor` instruction
5. The `tst` instruction

6. The `bfxil` (or `bfm`) instruction
7. The `bfi` (or `bfm`) instruction
8. The `clz` instruction
9. You could reverse the bits in the register, invert all the bits, and then use the `clz` instruction to find the first nonzero bit.
10. The `cnt` instruction

F.13 Chapter 13

1. Compile-time language
2. During assembly (compilation)
3. `#warning`
4. `.warning`
5. `#error`
6. `.error`
7. `#define`
8. `.equ`, `.set`, and `=`
9. `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`, and `#endif`
10. The main Gas conditional assembly directives are `.if`, `.elseif`, `.else`, and `.endif`. The secondary assembly directives are `.ifdef`, `.ifb`, `.ifc`, `.ifeq`, `.ifeqs`, `.ifge`, `.ifgt`, `.ifile`, `.iflt`, `.ifnb`, `.ifnc`, `.ifndef`/`.ifnotdef`, `.ifne`, `.ifnc`, and `.ifnes`.
11. The CPP `map` macro
12. `.rept`, `.irp`, `.irpc`, and `.endr`
13. `.irpc`
14. `#define`
15. `.macro` and `.endm`
16. Specify the macro name at the place in the file where an instruction mnemonic is expected.
17. Use functional notation. For example: `mymacro( p1, p2 )`.
18. Specify Gas macro parameters as operands in the instruction operand field. For example: `lea x0, label` (`x0` and `label` are the parameters to the `lea` macro).
19. Put `:req` after the parameter in the macro declaration.
20. Specify the parameter name in the macro declaration without a suffix (`:req`, `:varargs`, or *`=expression`*). By default, Gas macro parameters are optional.
21. Use `...` as the last (or only) argument in a `#define` macro definition.
22. Put `:varargs` after the last (or only) argument in a Gas macro definition.
23. Use the `.ifb` (if blank) conditional assembly directive.
24. `.exitm`

F.14 Chapter 14

1. A sequence of zero or more characters in memory ending with a byte containing 0
2. Because the program must often scan the entire string to determine its length
3. Because this string assembly language type (a) encodes the string's length as part of the data type, (b) aligns the string data on a 16-byte boundary, and (c) guarantees that the storage for the string is a multiple of 16 bytes long. This allows algorithms to fetch additional data beyond the end of the string, as long as all the data fits within a block of 16 bytes aligned on a 16-byte boundary.
4. Because the starting index argument can be any value
5. Because they must deal with variable-length characters

F.15 Chapter 15

1. `#ifndef` or `.ifndef`
2. The assembly of a source file plus any files it directly or indirectly includes
3. `.global`
4. `.extern`. Technically, using this directive is optional, as Gas assumes all undefined symbols are external.
5.

```
target: dependencies
    commands
```

6. A makefile-dependent file is a file that must be built or updated in order to properly build the current file (that is, the current file depends on the makefile-dependent file in order to be built).
7. Delete all executable and object-code files produced by a `make` operation.
8. A collection of object modules that the linker can use to extract (only) those object modules it needs

F.16 Chapter 16

1. The operating system typically uses `svc` to call an OS API function.
2. `#0`
3. `#0x80`

INDEX

B

C

D

E

F

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

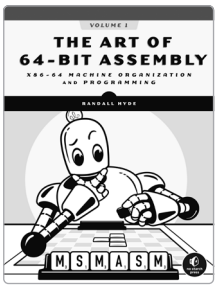
Z

The Art of ARM Assembly is set in New Baskerville, Futura, Dogma, and TheSansMono Condensed.

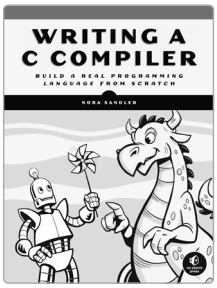
RESOURCES

Visit *https://nostarch.com/art-arm-assembly* for errata and more information.

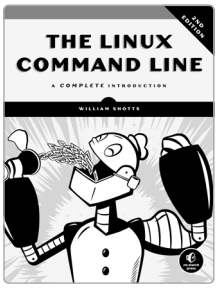
More no-nonsense books from **NO STARCH PRESS**



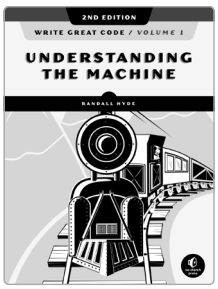
THE ART OF 64-BIT ASSEMBLY, VOLUME 1
x86-64 Machine Organization and Programming
BY RANDALL HYDE
1,032 PP., $79.99
ISBN 978-1-7185-0108-9



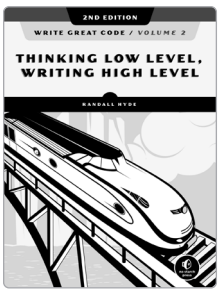
WRITING A C COMPILER
Build a Real Programming Language from Scratch
BY NORA SANDLER
792 PP., $69.99
ISBN 978-1-7185-0042-6



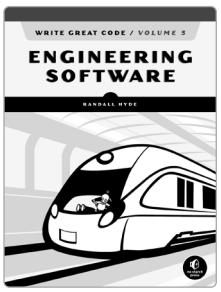
THE LINUX COMMAND LINE, 2ND EDITION
A Complete Introduction
BY WILLIAM SHOTTS
504 PP., $39.95
ISBN 978-1-59327-952-3



WRITE GREAT CODE, VOLUME 1, 2ND EDITION
Understanding the Machine
BY RANDALL HYDE
472 PP., $49.95
ISBN 978-1-7185-0036-5



WRITE GREAT CODE, VOLUME 2, 2ND EDITION
Thinking Low Level, Writing High Level
BY RANDALL HYDE
656 PP., $49.95
ISBN 978-1-7185-0038-9



WRITE GREAT CODE, VOLUME 3
Engineering Software
BY RANDALL HYDE
376 PP., $49.95
ISBN 978-1-59327-979-0

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

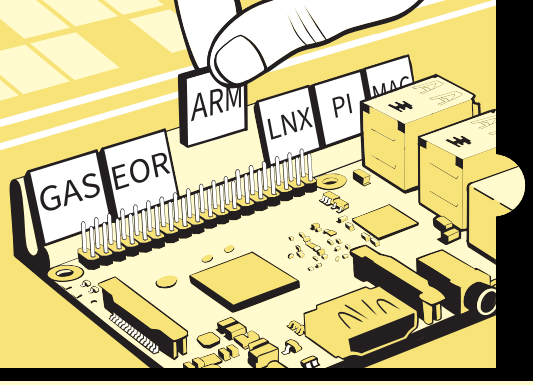
WEB:
WWW.NOSTARCH.COM

Never before has the world relied so heavily on the Internet to stay connected and informed. That makes the Electronic Frontier Foundation's mission—to ensure that technology supports freedom, justice, and innovation for all people—more urgent than ever.

For over 30 years, EFF has fought for tech users through activism, in the courts, and by developing software to overcome obstacles to your privacy, security, and free expression. This dedication empowers all of us through darkness. With your help we can navigate toward a brighter digital future.

ELECTRONIC FRONTIER FOUNDATION

LEARN MORE AND JOIN EFF AT EFF.ORG/NO-STARCH-PRESS



Modern Instructions for 64-Bit ARM CPUs

Building on Randall Hyde's iconic series, *The Art of ARM Assembly* delves into programming 64-bit ARM CPUs—the powerhouses behind iPhones, Macs, Chromebooks, servers, and embedded systems.

Following a fast-paced introduction to the art of programming in assembly and the GNU Assembler (Gas) specifically, you'll explore memory organization, data representation, and the basic logical operations you can perform on simple data types. You'll learn how to define constants, write functions, manage local variables, and pass parameters efficiently. You'll explore both basic and advanced arithmetic operations, control structures, numeric conversions, lookup tables, and string manipulation—in short, you'll cover it all.

You'll also dive into ARM SIMD (Neon) instructions, bit manipulation, and macro programming with the Gas assembler, as well as how to:

- Declare pointers and use composite data structures like strings, arrays, and unions
- Convert simple and complex arithmetic expressions into machine instruction sequences
- Use ARM addressing modes and expressions to access memory variables
- Create and use string library functions and build libraries of assembly code using makefiles

This hands-on guide will help you master ARM assembly while revealing the intricacies of modern machine architecture. You'll learn to write more efficient high-level code and gain a deeper understanding of software-hardware interactions—essential skills for any programmer working with ARM-based systems.

ABOUT THE AUTHOR

Randall Hyde is an embedded software engineer who has worked in the medical, nuclear, consumer electronics, and entertainment industries. He taught assembly language programming at the university level for over 10 years. He is the author of *The Art of Assembly Language*, *The Art of 64-Bit Assembly*, *The Book of I²C*, and the Write Great Code series, all from No Starch Press.

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™

nostarch.com