

RISC-V Assembly Language Programming

Using the ESP32-C3 and QEMU



Warren Gay

RISC-V Assembly Language Programming

Using ESP32-C3 and QEMU



Warren Gay

● This is an Elektor Publication. Elektor is the media brand of Elektor International Media B.V.
PO Box 11, NL-6114-ZG Susteren, The Netherlands
Phone: +31 46 4389444

● All rights reserved. No part of this book may be reproduced in any material form, including photocopying, or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication, without the written permission of the copyright holder except in accordance with the provisions of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licencing Agency Ltd., 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's permission to reproduce any part of the publication should be addressed to the publishers.

● **Declaration**

The Author and Publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, and hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident, or any other cause.

All the programs given in the book are Copyright of the Author and Elektor International Media. These programs may only be used for educational purposes. Written permission from the Author or Elektor must be obtained before any of these programs can be used for commercial purposes.

● British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

● **ISBN 978-3-89576-525-4** Print
ISBN 978-3-89576-526-1 eBook

● © Copyright 2022: Elektor International Media B.V.
Editor: Alina Neacsu
Prepress Production: D-Vision, Julian van den Berg

Elektor is part of EIM, the world's leading source of essential technical information and electronics products for pro engineers, electronics designers, and the companies seeking to engage them. Each day, our international team develops and delivers high-quality content - via a variety of media channels (including magazines, video, digital media, and social media) in several languages - relating to electronics design and DIY electronics. www.elektormagazine.com

Contents

Chapter 1 • Introduction	13
1.1. The Joy of the Machine	13
1.2. Assembler Language.	14
1.3. Why RISC-V?	14
1.4. Base Instruction Sets Covered	15
1.5. Projects in this Book	15
1.6. What do you need?	15
1.6.1. ESP32-C3 Hardware.	16
1.7. Assumptions About the Reader.	18
1.8. Summary	18
Chapter 2 • ESP32-C3 Installation.	19
2.1. ESP32-C3 Device	19
2.2. Manual Installation (Linux and MacOS)	20
2.3. Windows Install	29
2.4. Summary	34
Chapter 3 • Installation and Setup of QEMU	35
3.1. Linux/MacOS Platforms:	35
3.2. Windows	36
3.3. Installing QEMU on MacOS.	36
3.4. Install QEMU on Devuan Linux	36
3.5. QEMU Package Search.	37
3.5.1. Building QEMU on Linux	37
3.5.2. Basic Build Steps	38
3.5.3. Linux/MacOS Setup of Fedora Linux	41
3.5.4. Linux/MacOS Boot Fedora Linux	42
3.5.5. Linux/MacOS Boot Test.	43
3.6. Installing QEMU on Windows	46
3.7. Summary	50
Chapter 4 • Architecture	51
4.1. Program Counter Register	51

4.2. Endianness	51
4.3. General Purpose Registers	52
4.4. Introducing Subsets	52
4.5. Register Specifics	53
4.5.1. No Flag Bits	54
4.5.2. Register x0 / Zero	55
4.5.3. Register x1 / ra	55
4.5.4. Register x2 / sp	55
4.5.5. Register x3 / gp	55
4.5.6. Register x4 / tp	55
4.5.7. Registers x5-x7 / t0-t2	55
4.5.8. Register x8 / s0 / fp	55
4.5.9. Register x9 / s1	55
4.5.10. Registers x10-x11 / a0-a1	55
4.5.11. Registers x12-x17 / a2-a7	56
4.5.12. Registers x17-x27 / s2-s11	56
4.5.13. Registers x28-x31 / t3-t6	56
4.5.14. Register Summary	56
4.6. Instruction Set Base Subsets/Extensions	56
4.7. ESP32-C3 Hardware:	58
4.8. QEMU RISC-V 64 Bit Emulator	58
4.9. RISC-V Privilege Levels	59
4.10. RISC-V is Huge.	60
4.11. Summary.	60
Chapter 5 • Getting Started	61
5.1. Memory Models & Data Types	61
5.1.1. RV32 Model.	61
5.1.2. RV64 Model.	62
5.2. The Impact of XLEN	62
5.3. First Exercise	63
5.3.1. The Main Program	63
5.3.2. Assembler Routine add3	64

5.3.3. Assembly Language Format	64
5.3.4. Pseudo Opcode <code>.global</code>	65
5.3.5. Pseudo Opcode <code>.text</code>	65
5.3.6. The <code>add</code> Opcode	66
5.3.7. Calling <code>add3</code>	66
5.3.8. RV64I Consideration	67
5.3.9. Running the Demonstration	67
5.4. First Exercise on ESP32-C3	68
5.5. Assembler Listings	72
5.5.1. ESP32-C3 Assembler Listing	72
5.5.2. Influencing Assembly Code	73
5.4.3. <code>Objdump</code>	75
5.6. Summary	75
Chapter 6 • Load and Store Memory	76
6.1. A Word About Word Sizes	76
6.2. Load Instructions	76
6.3. Load Program Example	77
6.4. The <code>.data</code> Section	80
6.5. Unsigned Values.	83
6.6. Memory Alignment	83
6.7. Experiment	85
6.8. Immediate Values.	85
6.9. The <code>li</code> Pseudo-Op	86
6.10. The <code>addi</code> Opcode.	87
6.11. Pseudo-Op <code>mv</code>	87
6.12. Loads under RV64I	88
6.1.3 The <code>.section</code> Pseudo-Op	89
6.14. Storing Data	90
6.15. Review	94
6.16. RISC-V Assembler Modifiers	95
6.17. Summary.	95

Chapter 7 • Calling Convention	96
7.1. Register Usage	96
7.2. Call Procedure	97
7.2.1. Opcode jal	97
7.2.2. Pseudo Opcode jr.	97
7.2.3. Pseudo Opcode ret.	98
7.2.4. General Call Procedure	98
7.2.5. Call to 32-bit Absolute Address	98
7.2.6. Revised Call Procedure	99
7.2.7. Concrete Call Example	99
7.2.8. Simple Call Experiment	100
7.2.9. Running in gdb	101
7.3. Argument Passing in Registers	104
7.4. The Stack	105
7.4.1. Prologue	105
7.4.2. Epilogue	106
7.4.3. Floating Point Arguments	106
7.4.4. A Big Call Experiment.	106
7.5. Calling printf()	111
7.6. Summary	114
Chapter 8 • Flow Control	116
8.1. Branching Instructions	116
8.1.1. Unconditional Transfers	116
8.1.2. Conditional Branches	117
8.2. Shift Opcodes.	117
8.3. ESP32-C3 Project	118
8.3.1. Function c_ones()	119
8.3.2. Main Test Program	119
8.3.3. Assembler Function ones()	120
8.4. Compare and Set	123
8.5. Odd Parity Example	123

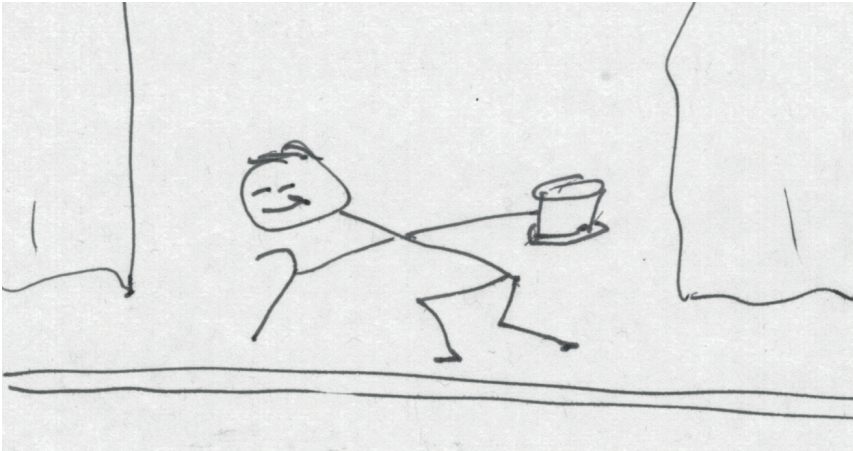
8.6. RV64I Odd Parity	125
8.7. Position Independent Code	128
8.8. Summary	129
Chapter 9 • Basic Opcodes	130
9.1. Arithmetic Opcodes.	130
9.1.1. add, addi and sub	130
9.1.2. lui	130
9.1.3. auipc	131
9.1.4. RV64 Arithmetic	131
9.2. Logical Opcodes	131
9.3. ESP32-C3 Rotate Left	132
9.4. RV64 Rotate Left	134
9.5. ESP32-C3 Rotate Right	136
9.6. Pseudo Opcodes	138
9.7. Unsigned Multi-precision Arithmetic	138
9.8. Signed Multi-precision Arithmetic	140
9.8.1. Signed Overflow	140
9.9. Summary	147
Chapter 10 • Multiply / Divide	148
10.1. Multiplication Operations	148
10.2. Division Operations	149
10.3. Opcode mul/mulu	149
10.4. Opcode mulhs/mulhu	149
10.5. Optimized Multiply	149
10.6. Unsigned Factorial	151
10.7. Opcode div/divu	153
10.8. Optimized Divide	153
10.9. Division By Zero	154
10.10. Divide Overflow	154
10.11. Safe Division	154
10.12. Greatest Common Divisor	158
10.13. Combinations	160

10.14. Summary	163
Chapter 11 • Addressing, Subscripting and Strings	164
11.1. Testing for Null Pointers	164
11.2. Sizeof Type for Pointers	164
11.3. Matrix Memory Layout	164
11.3.1. Subscript Calculation	165
11.4. Identity Matrix Example	165
11.5. String Functions	169
11.5.1. Function strlen()	169
11.5.2. Function strncpy32()	171
11.5.3. String to Integer Conversion	174
11.5.4. What if there is no Multiply?	178
11.5.5. Integer to String Conversion	178
11.6. Indexed Branching	181
11.7. Summary	184
Chapter 12 • Floating Point	185
12.1. Floating Point Registers	185
12.2. GNU Calling Convention	186
12.3. Floating-Point Control and Status Register (fcsr)	186
12.3.1. Rounding Modes fcsr.frm	187
12.3.2. Accrued Exception Flags fcsr.fflags	188
12.4. NaN Generation and Propagation	188
12.5. Opcodes and Data Formats	188
12.6. Load and Store	189
12.7. Floating Computation	189
12.8. Conversion Operations	190
12.8.1. Floating-Point Zero	190
12.8.2. Conversion Failures	191
12.9. Floating-Point Signs	191
12.10. Floating-Point Move	192
12.11. Floating-Point Compare	192
12.12. Classify Operation	192

12.13. Fahrenheit to Celsius Revisited	193
12.14. Summary.	196
Chapter 13 • Portability	197
13.1. C/C++ Pre-Processor	197
13.2. Testing for RISC-V Architecture	199
13.3. Testing For Integer Multiplication	199
13.4. RV32 vs RV64.	202
13.5. Assembler Macros.	204
13.6. Summary.	208
Chapter 14 • Determining Support	209
14.1. Privilege Levels.	209
14.1.1. Machine Level	210
14.1.2. Supervisor Level	210
14.1.3. User Level.	210
14.2. Control and Status Registers	210
14.2.1. Machine ISA Register	210
14.3. Opcodes	212
14.4. ESP32-C3	212
14.5. Reporting MISA	212
14.6. RV64 Platform	216
14.7. Counters	219
14.7.1 Project rdcycle	219
14.7.2. ESP32-C3 rdcycle Support	222
14.8. Summary.	225
Chapter 15 • JTAG Debugging	226
15.1. Espressif JTAG	226
15.2. Device Requirements.	226
15.3. Software Components	227
15.4. JTAG With No Serial Window	228
15.4.1. Starting OpenOCD	229
15.4.2. Problems with OpenOCD.	230
15.4.3. Terminating OpenOCD	230

15.4.4. Start gdb	230
15.5. Operating gdb	232
15.5.1. Abbreviations	232
15.5.2 GDB Walkthrough	233
15.5.3. Quitting gdb	240
15.6. JTAG With a Serial Window	241
15.6. Miscellaneous	244
15.7. Summary	244
Chapter 16 • Inline Assembly	246
16.1. Keyword Asm	246
16.2. Basic asm Form	246
16.2.1. Keyword volatile	248
16.2.2. Multiple Instructions	248
16.2.3. Behind the Scenes	249
16.3. Extended Asm	249
16.3.1. Assembler Template	249
16.3.2. Output Operands	250
16.3.2.1. Constraint	250
16.3.3. Input Operands	253
16.3.3.1. Clobbers	254
16.4. Bit Multiply	255
16.5. Example asm goto	257
16.6. Register Constraints	259
16.7. Summary	265
Index	266

Chapter 1 • Introduction



Introducing.... RISC-V!

With the availability of free and open-source C/C++ compilers today, you might wonder why someone would be interested in assembler language. What is so compelling about the RISC-V Instruction Set Architecture (ISA)? How does RISC-V differ from existing architectures? And most importantly, how do we gain experience with the RISC-V without a major investment? Is there affordable hobbyist hardware available?

The availability of the Espressif ESP32-C3 chip provides a student with one affordable way to get hands-on experience with RISC-V. The open-sourced QEMU emulator adds a 64-bit experience in RISC-V under Linux. These are just two ways for the student and enthusiast alike to explore RISC-V in this book.

1.1. The Joy of the Machine

In the earliest days of computing, there was great enthusiasm in working out the mechanical steps required to perform some computation or algorithm. The programmer was aware of every machine cycle, status bit and register available to him as resources to be exploited. In those early times debugging was often performed on a console decorated with lamps, buttons and switches. There was always great satisfaction in getting it right and making it run even faster than before.

I fear that today, we've lost some of that passion for exploiting the machine. Our CPU tower might possess only a power-on button and LED. But if we're lucky, it might provide a disk activity LED as well. Embedded systems are better, sporting multiple LEDs that can be utilized. Whether desktop or embedded device, we still program in high-level languages like C/C++. While there is still joy in that, the thrill of the hunt may be lacking for those thirsting for more. Getting that algorithm to execute even faster or more efficiently is not just a badge of honor, but a matter of pride.

One deterrent to assembly language has been the extremely cluttered instruction sets of today's architectures. The Intel and AMD instructions sets, for example, have been horribly extended in the name of compatibility. RISC-V allows the enthusiast to sweep all that clutter aside and start over with a clean slate. This makes things so much easier because there is less to be learned.

Getting back to assembler language will bring back that thrill of the hunt that you may be pining for. At the assembly level, the programmer directs every step of the machine. You decide how the registers are allocated and apply every binary trick in your toolbox to make that process slick. While we don't have hardware debug consoles anymore, we do have powerful debugging tools like the GNU gdb debugger. There's never been a better time to do assembly language than today.

1.2. Assembler Language

It might seem counterproductive to program in assembly language. While C/C++ languages will continue to be used for productive development, there frequently remain opportunities for optimizing code at the machine level. In these areas, you will be empowered to exploit the machine in otherwise difficult ways.

One attractive area is writing custom floating-point algorithms in assembly language. For each calculation, you can carefully evaluate which rounding method to use and check for special exception cases at strategic places in the code. In C/C++, the tendency is to simply code the formula.

Even if you decide that you have no need to *write* assembly language code, being able to *read* it can be extremely helpful when debugging. Compilers, often at higher optimization levels, can generate incorrect code (this happens more often than you think). Being able to verify that the generated code is correct can save you from a great deal of guessing when debugging. With a working knowledge, you can step through the code one instruction at a time in a debugger and pinpoint the problem. Once the problem is revealed, you can then decide on a work-around for the compiler or replace the errant code directly with some assembler language code.

In microcomputer solutions, performance is often paramount. Assembler language programming provides more options for meeting those performance goals. Finally, it is possible that one day RISC-V may become one of the few *standard* instruction sets in general use.

1.3. Why RISC-V?

Probably its most attractive feature is that it is designed to be free and open so that it is not hindered by restrictive licensing. This permits any manufacturer to create a RISC-V product without purchasing an expensive or restrictive license.

The RISC-V instruction set is also designed to be clean, unlike many existing architectures. Today's Intel ISA is a bewildering mess of adapted and extended instructions. To be fair, this was done to maintain code compatibility, but what a bewildering mess it has become.

Now that security is more important than ever, there is a pressing need for a simpler design.

Reduced Instruction Set Computer (RISC) architecture had its beginnings in research projects conducted by John L. Hennessy at Stanford University between 1981 and 1984. MIPS (Microprocessor without Interlocked Pipeline Stages) was developed with the IBM 801 and the Berkeley RISC projects. The Berkeley RISC project was led by David Patterson who coined the term RISC. The thrust of these efforts was to develop a design that was simpler and potentially faster than the CISC (Complex Instruction Set Computers) of that time.

Today, simplicity benefits the chip manufacturer in lowering the cost of chip development and verification. Simplicity means fewer transistors, which leads to lower power requirements. A simple instruction set also reduces the complexity that the software developers must face. Finally, the RISC-V ISA has provision to include vendor *extensions*, without requiring any special approval. This can lead to surprising innovations.

1.4. Base Instruction Sets Covered

RISC-V is now a large body of work. In chapter 4, Architecture, I'll discuss base instruction sets and extensions to RISC-V. The focus of this book, however, will be on the RV32 and RV64 subsets of RISC-V. This permits an ample study of the ESP32-C3 device (RV32) as well as Fedora Linux under QEMU (RV64). These two environments should provide the reader with a well-rounded tutorial.

1.5. Projects in this Book

In many books applying MCU (Microprocessor Computing Units) concepts, the project builds are the focus. In them, the emphasis is placed on using GPIO (General Purpose Input/Output), I2C, SPI and other built-in peripherals to build something fun.

This book has a different focus, but it is still fun! The projects in this book are boiled down to the barest essentials to keep the assembly language concepts clear and simple. In this manner, you will have "aha!" moments rather than puzzling about something difficult. The focus of this book is on *learning how to write* RISC-V assembly language code without getting bogged down. As you work your way through this tutorial, you'll build up small demonstration programs to be run and tested. Often the result is some simple printed messages to prove a concept. Once you've mastered these basic concepts, you will be well equipped to apply assembly language in larger projects.

1.6. What do you need?

This book uses the Espressif ESP32-C3 dev kit device for hardware. You will also emulate RISC-V in 64-bit mode using the QEMU emulator on your desktop computer. The emulator requires that you have at least 20 GB of free disk space available. For both of the ESP-IDF (ESP Integrated Development Framework) and QEMU instances, an internet connection for downloading is assumed.

A suitably modern desktop computer is required to program the ESP32-C3 and to run the QEMU emulator, running Fedora Linux. If you lack 20 GB of free disk space, then that can be remedied by plugging in a USB hard drive to add some disk space.

1.6.1. ESP32-C3 Hardware

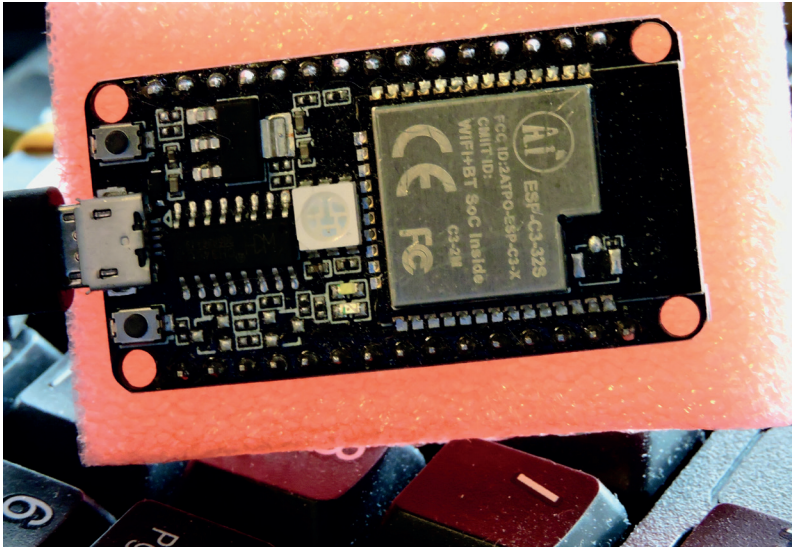
When purchasing hardware for use with this book, be sure to acquire the correct device type. The original ESP32 devices used the Xtensa processor and *are not RISC-V devices*. At the time of writing there are several types of ESP32:

- ESP32 (Xtensa dual-core and single-core 32-bit LX6 microprocessor)
- ESP32-S2 (Single-core Xtensa LX7 CPU)
- ESP32-S3 (Dual-core Xtensa LX7 CPU)
- ESP32-C3 Single-core 32-bit RISC-V (WiFi 2.4 Ghz IEEE 802.11b/g/n)
- ESP32-C6 Single-core 32-bit RISC-V

This book was developed specifically with the Espressif ESP32-C3 device (emphasis on the "C3").

When purchasing, don't make the mistake of just buying just a chip. Be sure to purchase a "devkit" that consists of a PCB with the GPIO breakouts, USB interface and the ESP32-C3 chip soldered onto it. You might also find it listed as "ESP-C3", but be careful. Make sure this refers to the ESP32-C3, and not some other ESP32 variants (using the Xtensa CPU). You'll also need an appropriate USB cable to flash and communicate with the devkit. The projects in this book can run off of the power from the USB cable.

Figure 1.1 illustrates an early ESP32-C3 dev board purchased from AliExpress. This version 1 dev board cannot support JTAG. Notice that these use a USB to serial interface chip (CH340C), which can be seen in the photo. If you aren't concerned about JTAG support, then these are otherwise suitable to use.



*Figure 1.1: An early version 1 dev board.
Note the USB to serial chip just right of the micro USB connector.*

As time went on, Espressif came out with ESP32-C3 devices with JTAG support (version 3 or later). Since these connect directly from the ESP32-C3 chip to the USB port, they don't use a serial to USB converter chip. This makes it easy to identify the boards that support JTAG. With a direct connection to the USB bus, the ESP32-C3 can perform JTAG functions over the USB bus, as well as the usual serial communications. These are the preferred ESP32-C3 devkits to obtain. Figure 1.2 illustrates a JTAG-capable ESP32-C3 dev board sitting on a breadboard.

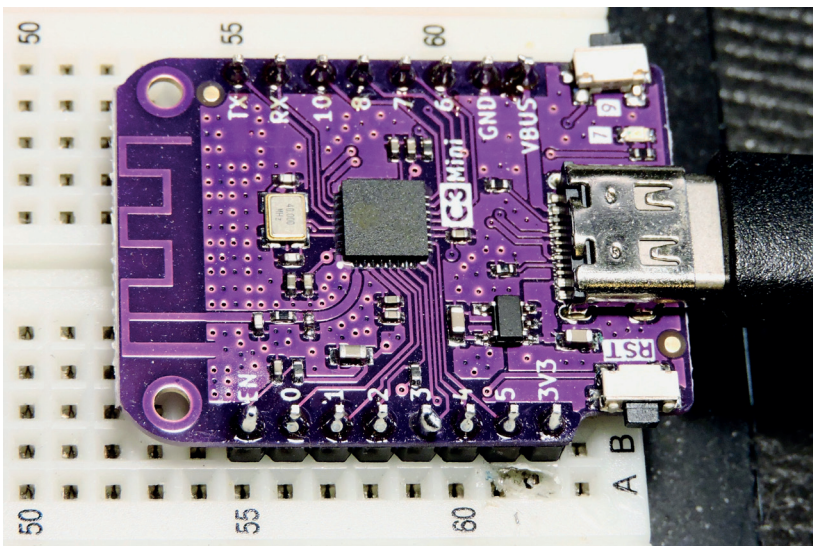


Figure 1.2: A revision 3 ESP32-C3 dev board supporting JTAG access.

1.7. Assumptions About the Reader

The reader is assumed to have a basic understanding of what a CPU is, what registers are and the rudiments of computer memory. Assembler language and debugging require knowledge of number systems, specifically binary and hexadecimal. The reader should also be familiar with endianness. Knowing how big-endian ordering differs from little-endian will be helpful.

The reader is expected to be familiar with basic file system navigation: changing directories, creating directories and copying files. In some cases, the editing of a script file may be necessary. The Windows user should also be familiar with the Linux file system in this regard when using QEMU (emulating Fedora Linux).

Knowledge of the C language is assumed. Code examples will consist of a C language main program calling an assembly language function.

The QEMU examples use a RISC-V version of downloadable Fedora Linux. Consequently, some familiarity with the Linux command line is an asset even for Windows users. The ESP32-C3 examples will use your native desktop environment for ESP development, whether Linux, MacOS or Windows. In all cases, simple commands are issued to build and run the test programs.

Finally, the reader is expected to do some software downloading and installation. Unfortunately, there is a fair amount of this initially, but the good news is that it is all free and open software. Once installed, the reader can then focus on the RISC-V concepts presented in this book.

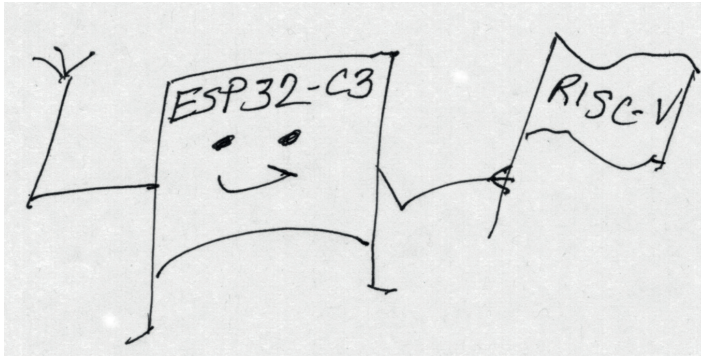
1.8. Summary

The next pair of chapters deal with installing your ESP-IDF (ESP Integrated Development Framework) and the QEMU emulation software. This bit of work is necessary to let the good times roll. So gather your disk space and enable your internet access and begin!

Bibliography

- [1] [The_RISC-V_Reader_An_Open_Architecture_Atlas_by_David_Patterson_Andrew_Waterman.pdf](#)

Chapter 2 • ESP32-C3 Installation



ESP32-C3 celebrates RISC-V

This chapter illustrates the steps necessary to get you up and running using the ESP32-C3 device. The first part of this chapter focuses on Linux and MacOS software installations. This is what Espressif calls "Manual Installation". Espressif also supports IDE installations for Eclipse or VSCode if you prefer. See their online documentation for that.

Windows users will want to skip to the later part of this chapter starting with the section heading "Windows Install". This will guide you through the use of the downloaded Espressif windows installer.

For either installation, you will need to plan for ample disk space and be connected to your internet. The ESP-IDF on the Mac (for device ESP32-C3 only) requires about 1.5 GB of disk. But the compilers and other tools will also require additional disk space as they are installed. I recommend that you allocate a minimum of 10 GB of free disk space before you proceed. The downloads are rather large and will take some time to complete. Choose a time to install where you are not rushed.

2.1. ESP32-C3 Device

Espressif markets and sells several devices under the ESP32 moniker, so make sure you purchase the correct device in order to enjoy RISC-V adventures. If your part says ESP32 but not C3, it is *not* the RISC-V version of the CPU. These devices are also sold as bare modules. So be sure to get a "dev kit" form of the product. One product that I am using is a ESP32-C3-DevKitM-1 clone. By the time you read this there may be newer versions of the ESP32-C3-DevKit and those are likely your best option.

Dev kits may include a USB to serial chip (USB-UART bridge) like the CP2102. My devices used the CHG340 chip. Either bridge chip is ok, since we are only interested in the RISC-V CPU in this book. Now you can purchase dev kits with JTAG support. These devices with revision 3 or later use the USB facilities directly. Figure 2.1 illustrates two early examples of revision 1 dev boards. The CH340C USB to serial converter chip is very conspicuous.

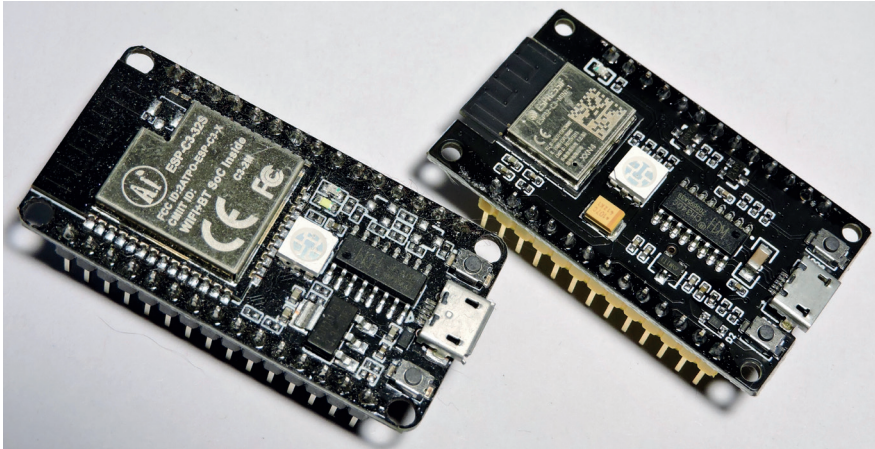


Figure 2.1: Two ESP32-C3 devices with revision 1 PCBs (using USB to serial converters).

2.2. Manual Installation (Linux and MacOS)

Locate the Espressif "Get Started" web page. If the website doesn't change too much, you should be able to arrive there directly using the following url:

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/get-started/index.html>

Otherwise search for "ESP-IDF Programming Guide" with your browser.

Once you arrive at that page, don't forget to choose the product type ESP32-C3 in the upper left. Espressif supports multiple device types, and we are primarily interested in the ESP32-C3 with RISC-V support. It should look something like Figure 2.2.

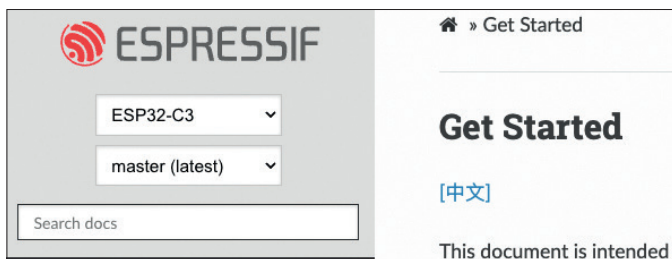


Figure 2.2: Set the device selection as ESP32-C3.

Scroll down the page until you see the link "Linux and MacOS". Click on the link to open a new page of instructions.

Linux and MacOS

On this page you will be guided through the following basic steps:

1. Install Prerequisites.
2. Get ESP-IDF.
3. Setup the tools.
4. Set up the environment variables.
5. First Steps for ESP-IDF.

Note: If the instructions differ at all from this text, do follow the instructions found on the website instead. Espressif may have made changes to the installation procedure by the time you read this.

Install Prerequisites

The prerequisites will vary somewhat with the Linux distribution you're using. Check the Espressif website for the latest updates by distro.

Ubuntu and Debian

For these distributions, the following packages should be installed (some may already be installed):

```
$ sudo apt install git wget flex bison gperf python3 python3-pip \
python3-setuptools cmake ninja-build ccache libffi-dev libssl-dev \
dfu-util libusb-1.0-0
```

You might wish to split these up into smaller steps, like the following:

```
$ sudo apt install git wget
$ sudo apt flex bison
$ sudo apt gperf python3
$ sudo apt python3-pip python3-setuptools
$ sudo apt cmake ninja-build
$ sudo apt ccache libffi-dev
$ sudo apt libssl-dev dfu-util libusb-1.0-0
```

CentOS 7 & 8

CentOS uses the yum installer, and the dependencies are listed as follows:

```
$ sudo yum -y update && sudo yum install git wget flex bison gperf \
python3 python3-pip python3-setuptools cmake ninja-build ccache \
dfu-util libusbx
```

Arch

Espressif has documented the following dependencies for the Arch distro:

```
$ sudo pacman -S --needed gcc git make flex bison gperf python-pip \  
  cmake ninja ccache dfu-util libusb
```

Espressif notes that CMake version 3.5 or newer is required. If you're running an older distribution, you may need to update your system packages. For other Linux distributions not listed here, use the above as a hint to the package names that you may need to add or update.

MacOS

MacOS users usually install HomeBrew (recommended) or the MacPorts open-source projects to add functionality to their Mac. If you've not done that yet, you need to do that now. Refer to the following sites for more information about this:

<https://brew.sh/> (HomeBrew)

<https://www.macports.org/install.php> (MacPorts)

Whether you use HomeBrew or MacPorts, you must install pip:

```
$ sudo easy_install pip
```

Next, CMake and Ninja are installed. For HomeBrew install:

```
$ brew install cmake ninja dfu-util
```

For MacPorts users, use:

```
$ sudo port install cmake ninja dfu-util
```

Espressif recommends that you also install ccache for faster build times. For HomeBrew, use:

```
$ brew install ccache
```

For MacPorts use:

```
$ sudo port install ccache
```

Note: Espressif indicates that if during the installation you encounter an error like the following:

```
xcrun: error: invalid active developer path (/Library/Developer/Command-  
LineTools), missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun  
Then you will need to install the Apple XCode command line tools in order to continue.  
Install these with:
```

```
$ xcode-select --install
```

MacOS Python 3

Check the version of python you have installed with:

```
$ python --version
```

If it is a version older than 3, or is not present, then check the following (notice that the command name is python3 this time):

```
$ python3 --version
```

If that fails, then you need to install it. For HomeBrew:

```
$ brew install python3
```

For MacPorts:

```
$ sudo port install python38
```

Get ESP-IDF

At this point, you need to decide where to place your ESP-IDF software (not as root). I'm going to assume in this book, that the directory will be named `~/espc3`, where the tilda (`~`) represents your home directory. You can choose a different directory name, by simply substituting `~/espc3` for the name you prefer.

First, create the subdirectory to house your files in (starting from your home directory):

```
$ mkdir -p ~/espc3
```

and then change to it:

```
$ cd ~/espc3
```

Now access the files from GitHub by performing:

```
$ git clone --recursive https://github.com/espressif/esp-idf.git
```

This git operation downloads several files and will take some time to complete. It is also a good opportunity to take a break for your favourite beverage.

When the git operation completes, you will have the Espressif software downloaded into the subdirectory `~/espc3/esp-idf`.

Set up the Tools

Now we need to install some tools used by the ESP-IDF framework, like the compiler and Python packages.

```
$ cd ~/espc3/esp-idf
$ ./install.sh esp32c3
```

Performing this step will result in more files being downloaded and installed. This step progresses much faster than the GitHub clone operation but can still require some time. Perhaps your beverage needs a refill?

At the end of this installation, you might encounter a message like:

```
WARNING: You are using pip version 21.2.1; however, version 22.0.3 is available.
```

```
You should consider upgrading via the
```

```
'~/espressif/python_env/idf5.0_py3.9_env/bin/python -m pip install --upgrade
pip' command.
```

This is optional, but I chose to do it.

At the end of the installation, your session should have completed with the message:

All done! You can now run:

```
$ . ./export.sh
```

At this point, you should run this script and watch for any error messages. Some errors you might see may include:

```
ERROR: tool xtensa-esp32-elf has no installed versions. Please run '/Users/joe/
espc3/esp-idf/install.sh' to install it.
ERROR: tool xtensa-esp32s2-elf has no installed versions. Please run '/Users/joe/
espc3/esp-idf/install.sh' to install it.
ERROR: tool xtensa-esp32s3-elf has no installed versions. Please run '/Users/joe/
espc3/esp-idf/install.sh' to install it.
```

In other words, these messages complain about missing support for:

- xtensa-esp32-elf
- xtensa-esp32s2-elf
- xtensa-esp32s3-elf

If you're only concerned about our ESP32-C3 device, which was not listed in error, you can ignore these messages. If you also want to support these other devices (regular ESP32,

ESP32-S2 and ESP32-S3), you can follow the Espressif advice and run the installation scripts suggested.

Set up Environment Variables

To run the Espressif tools to build your projects some environment changes must be applied each time you log in. In a fresh terminal session, you would need to do the following. Make sure that you type a space between the dot and the rest of the pathname:

```
$ . ~/espc3/esp-idf/export.sh
```

Assuming that no critical error messages are reported, this sets up your terminal session to build your ESP projects. Some users may wish to create a shorter way to do this. Espressif recommends creating an alias like `get_idf` as follows:

```
$ alias get_idf='. $HOME/espc3/esp-idf/export.sh'
```

Once that alias is defined, you can just type:

```
$ get_idf
```

to establish your build environment. Depending upon the shell you use for your terminal session, you might want to set the `get_idf` alias up in your `~/.profile` or `~/.bashrc` file, so that it is automatically defined each time you log in.

Once that `export.sh` script has run, your environment will have the variable `IDF_PATH` set. In this chapter, it would have the value `"~/espc3/esp-idf"`. This allows you to use the shell value `$IDF_PATH` in commands and scripts if you like.

First Steps for ESP-IDF

The installation procedure has covered a lot of ground so let's test it. Change to the example subdirectory shown:

```
$ cd $IDF_PATH/examples/get-started/hello_world
```

(or)

```
$ cd ~/espc3/esp-idf/examples/get-started/hello_world
```

Configure the Target Device

In order to build for the ESP32-C3 device, we need to tell the build framework about it:

```
$ idf.py set-target esp32c3
```

Build Example `hello_world`

This step adjusts the environment so that it knows that it is compiling for our RISC-V device (ESP32-C3). Now test the build process:

```
$ idf.py build
```

The first time this runs for a given project, it will compile a lot, but don't be concerned. Subsequent builds will not take so long. When it succeeds, the process should end with a message:

```
Project build complete. To flash, run this command:  
...snip...  
or run 'idf.py -p (PORT) flash'
```

Flash the Device

Before we can flash your RISC-V (ESP32-C3), we need to find out what port it appears on when you plug its cable into a USB port. But before you plug in your device's USB cable, list the cu devices under /dev as follows:

```
$ ls /dev/cu*  
/dev/cu.Bluetooth-Incoming-Port /dev/cu.usbserial-0001
```

This will list some that are already present. Don't be concerned if no devices show up. Now plug in your RISC-V device's USB cable and list the files again:

```
$ ls /dev/cu*  
/dev/cu.Bluetooth-Incoming-Port /dev/cu.usbserial-0001 /dev/cu.usbserial-1430
```

In this example, the device /dev/cu.usbserial-1430 was added. This is the device name we need for flashing our device. Define it in the shell variable named PORT, so that you won't have to type it each time:

```
$ PORT=/dev/cu.usbserial-1430  
$ export PORT
```

To flash the example program to your device, try it now (the example output has been abbreviated somewhat):

```
$ idf.py flash  
Executing action: flash  
Serial port /dev/cu.usbserial-1430  
Connecting....  
Detecting chip type... ESP32-C3  
Running ninja in directory /Users/joe/espc3/esp-idf/examples/get-started/  
hello_world/build  
Executing "ninja flash"...  
...  
Chip is ESP32-C3 (revision 3)  
Features: Wi-Fi  
Crystal is 40MHz
```

```
MAC: 7c:df:a1:b4:44:94
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Flash will be erased from 0x00000000 to 0x00004fff...
Flash will be erased from 0x00010000 to 0x00034fff...
Flash will be erased from 0x00008000 to 0x00008fff...
Compressed 19984 bytes to 12116...
Writing at 0x00000000... (100 %)
Wrote 19984 bytes (12116 compressed) at 0x00000000 in 0.7 seconds (effective
221.1 kbit/s)...
Hash of data verified.
Compressed 151072 bytes to 81515...
Writing at 0x00010000... (20 %)
Writing at 0x00019a2e... (40 %)
Writing at 0x00020360... (60 %)
Writing at 0x00027602... (80 %)
Writing at 0x0002dada... (100 %)
Wrote 151072 bytes (81515 compressed) at 0x00010000 in 2.7 seconds (effective
443.9 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.1 seconds (effective 262.5
kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done
```

If your session appeared similar to this, then congratulations are in order. You have flashed your first RISC-V program to the device.

Running `hello_world`

To see the `hello_world` program run, we *monitor* it as follows:

```
$ idf.py monitor
$ idf.py monitor
Executing action: monitor
Serial port /dev/cu.usbserial-1430
Connecting....
Detecting chip type... ESP32-C3
```

```

Running idf_monitor in directory /Users/joe/esp32/esp-idf/examples/get-started/
hello_world
...
--- idf_monitor on /dev/cu.usbserial-1430 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ESP-ROM:esp32c3-ap1-20210207
Build:Feb 7 2021
rst:0x1 (POWERON),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd6100,len:0x1750
load:0x403ce000,len:0x930
load:0x403d0000,len:0x2d3c
entry 0x403ce000
I (30) boot: ESP-IDF v5.0-dev-1730-g229ed08484 2nd stage bootloader
I (30) boot: compile time 19:50:12
I (30) boot: chip revision: 3
I (33) boot.esp32c3: SPI Speed      : 80MHz
I (38) boot.esp32c3: SPI Mode      : DIO
I (43) boot.esp32c3: SPI Flash Size : 2MB
I (48) boot: Enabling RNG early entropy source...
I (53) boot: Partition Table:
I (57) boot: ## Label                Usage            Type ST Offset   Length
I (64) boot:  0 nvs                  WiFi data        01 02 00009000 00006000
I (71) boot:  1 phy_init              RF data          01 01 0000f000 00001000
I (79) boot:  2 factory                factory app      00 00 00010000 00100000
I (86) boot: End of partition table
...
I (145) boot: Loaded app from partition at offset 0x10000
I (148) boot: Disabling RNG early entropy source...
I (165) cpu_start: Pro cpu up.
I (173) cpu_start: Pro cpu start user code
I (173) cpu_start: cpu freq: 160000000 Hz
I (173) cpu_start: Application information:
I (176) cpu_start: Project name:      hello_world
I (182) cpu_start: App version:       v5.0-dev-1730-g229ed08484
I (188) cpu_start: Compile time:      Mar 1 2022 19:50:05
I (194) cpu_start: ELF file SHA256:  d4ad172e8078f033...
I (200) cpu_start: ESP-IDF:          v5.0-dev-1730-g229ed08484
I (207) heap_init: Initializing. RAM available for dynamic allocation:
I (214) heap_init: At 3FC8C540 len 00033AC0 (206 KiB): DRAM
I (220) heap_init: At 3FCC0000 len 0001F060 (124 KiB): STACK/DRAM
I (227) heap_init: At 50000020 len 00001FE0 (7 KiB): RTCRAM
I (234) spi_flash: detected chip: generic
I (238) spi_flash: flash io: dio
I (242) sleep: Configure to isolate all GPIO pins in sleep state

```

```

I (249) sleep: Enable automatic switching of GPIO sleep configuration
I (256) cpu_start: Starting scheduler.
Hello world!
This is esp32c3 chip with 1 CPU core(s), WiFi/BLE, silicon revision 3, 2MB
external flash
Minimum free heap size: 328924 bytes
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...

```

After the device boots up, you will see the message "Starting scheduler". The next message shown is the "Hello world!" that we were waiting for. Then the program counts down and will reboot again until you terminate it. To stop monitoring, type Control-] (control plus the right square bracket) and you should be returned to your shell. Congratulations, you ran your first RISC-V program!

2.3. Windows Install

While it is possible to install the Windows Subsystem for Linux (WSL and WSL2) on Windows 10 and later, you may not be able to get the Linux USB access to work with the ESP32-C3 device. For this reason, I'll document the native Windows install procedure as provided by Espressif. Espressif provides the install documentation here:

```

https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/get-started/windows-setup.html

```

Limitations

Espressif lists the following limitations:

- The installation path of ESP-IDF and ESP-IDF Tools must not be longer than 90 characters.

From their web page:

[Installation paths that are too long] might result in a failed build. The installation path of Python or ESP-IDF must not contain white spaces or parentheses. The installation path of Python or ESP-IDF should not contain special characters (non-ASCII) unless the operating system is configured with "Unicode UTF-8" support.

System Administrator can enable the support via Control Panel. Change date, time, or number formats - Administrative tab - Change system locale - check the option "Beta: Use Unicode UTF-8 for worldwide language support" - Ok and reboot the computer.

ESP-IDF Tools Installer

Look for and click on the link "Windows Installer Download" provided on their website". Make certain that you have selected ESP32-C3 from the top left drop down of the ESP-IDF Programming Guide web page. At the time of writing, this downloads from:

```
https://dl.espressif.com/dl/esp-idf/?idf=4.4
```

They provide several different installers, which at the time of writing include:

- Universal Online Installer 2.13 for Windows 10, 11, size 4 MB
- Espressif-IDE 2.4.2 with ESP-IDF v4.4 for Windows 10, 11, size 1 GB
- ESP-IDF v4.4 Offline Installer for Windows 10, 11, size 600 MB
- ESP-IDF v4.3.2 Offline Installer for Windows 10, 11, size 570 MB
- ESP-IDF v4.2.2 Offline Installer for Windows 10, 11, size 376 MB
- ESP-IDF v4.1.2 Offline Installer for Windows 10, 11, size 353 MB

In this guide, I've chosen the Universal Online Installer. Once the installer has been downloaded and launched, you'll be prompted to "Select Setup Language", which will be "English" by default.

Next is the "License Agreement", where you want to click "I accept the agreement" and then click the "Next" button.

The installer does a "Pre-installation system check". If there were any warnings, click the "Apply Fixes" button. Likely it will complain that you need Administrator access, which the "Apply Fixes" button will correct. Once all is good, click on the "Next" button.

The next dialog "Download or use ESP-IDF" asks you to choose one of:

- Download ESP-IDF.
- Use an existing ESP-IDF directory.

Choose "Download ESP-IDF" and click the "Next" button.

In the "Version of ESP-IDF" dialog, choose the version of the software to download. At the time of writing, the most current version was v4.4 (release version). It is probably best to choose the latest version available. Click the "Next" button.

In the "Select Destination Location" dialog, you are asked where you want the software to be installed. By default, the text "C:\Espressif" is used. If you need it on another drive or a different directory, this is where you can choose it. Click the "Next" button.

In the "Select Components" dialog, you are presented with a tree of checkboxes. The one area that you might want to deliberate is the "Chip Targets", where it lists the following choices:

- ESP32 (optional)
- ESP32-C3 (make sure this remains checked for RISC-V)
- ESP32-S Series:
 - ESP32-S2 (optional)
 - ESP32-S3 (optional)

If you need to save disk space, then unselect the optional items listed above. Make certain however that ESP32-C3 remains checked. Click the "Next" button.

At this point, the "Ready to Install" dialog box appears with a summary of your choices so far. If the summary appears okay, then click the "Install" button.

If any prompt presents a "Do you want to allow this app to make changes to your device?" question, then answer with a click on the "Yes" button.

You will also get prompts from "Windows Security" to the effect "Would you like to install this device software?". Click on the "Install" button.

The software will then download and install for a considerable amount of time. Later on, the "Installing ESP-IDF tools" dialog will appear towards the end. If you're feeling a craving for coffee or tea, this would be a great time to make it. The download and installation may take about 45 minutes depending upon your system and internet.

After the above is completed, you are presented with a "Completing the ESP-IDF Tools Set-up Wizard" dialog. I suggest you leave all the options checked and click the "Finish" button. Click on "Yes" if you are prompted with "Do you want to allow this app to make changes to your device?".

Once all that has been completed, you should have two more start menu options:

- ESP-IDF 4.4 CMD
- ESP-IDF 4.4 PowerShell

as shown in Figure 2.3 (yours may differ slightly).

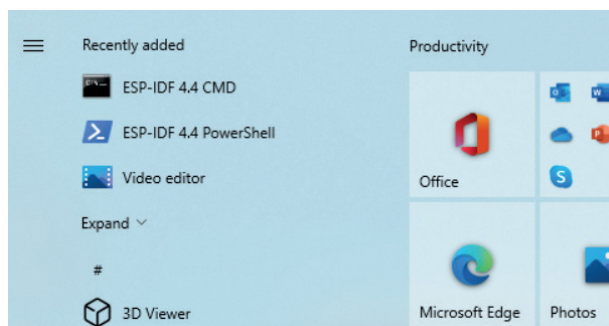


Figure 2.3: The created ESP-IDF startup links in the start menu.

Now it is time to test your installed ESP-IDF on a provided example program.

Windows First Steps on ESP-IDF

Using the ESP-IDF PowerShell (or CMD), let's use a test project to test the software with. When you start your shell, it will place you into a directory that depends upon your install choices. The prompt that I got was:

```
PS C:\Espressif\frameworks\esp-idf-v4.4>
```

In the text that follows, I will just show "PS C:>" as the prompt. Now change to the `hello_world` subdirectory:

```
PS C:> cd examples
PS C:> cd get-started
PS C:> cd hello_world
```

Choose your target device type by executing the following command:

```
PS C:> idf.py set-target esp32c3
```

This configures the build for this project to compile for your RISC-V device. Once that completes, you can build your `hello_world` software:

```
PS C:> idf.py build
```

The first time you build your project a lot of compiling of dependencies will occur. On successive builds however, the process only rebuilds what is needed and is much faster. If all went well, the command should complete with the message:

```
Project build complete. To flash, run this command:
...
or run 'idf.py -p (PORT) flash'
```

Now we must discover the name of your USB device when the ESP32-C3 is plugged in. Open the Device Manager and expand the "Ports (COM & LPT)" after plugging in your ESP32-C3 device. See Figure 2.4 for an example:

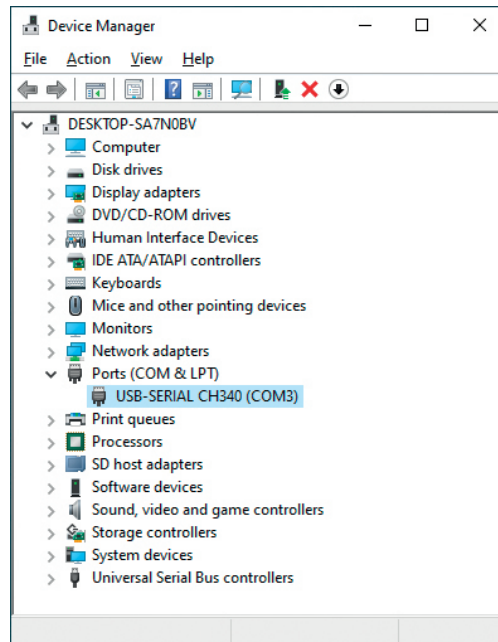


Figure 2.4: Locating the COM port for the ESP32-C3 plugged into the serial port.

In this example, we see that Windows has registered the ESP32-C3 as the device name COM3. Your device may use a different name depending on the device hardware. My device used a CH340 USB serial interface chip, and so the "CH340" shows up in the name.

Once you know the device name, you can flash it:

```
PS C:> idf.py -p COM3 flash
```

If all went well, you will see that it recognizes the device and uploads the compiled software to it. In order to see `hello_world` run, we monitor it with:

```
PS C:> idf.py -p COM3 monitor
```

Several message lines will appear, but eventually you will see some output similar to the following:

```
--- idf_monitor on com3 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
ESP-ROM:esp32c3-api1-20210207
Build:Feb 7 2021
rst:0x1 (POWERON),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd6100,len:0x16bc
```

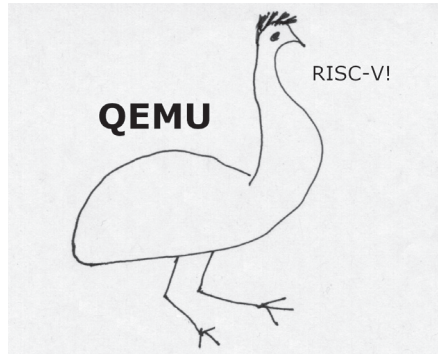
```
load:0x403ce000,len:0x930
load:0x403d0000,len:0x2d40
entry 0x403ce000
I (30) boot: ESP-IDF v4.4 2nd stage bootloader
I (30) boot: compile time 15:55:20
I (30) boot: chip revision: 3
I (32) boot.esp32c3: SPI Speed      : 80MHz
I (36) boot.esp32c3: SPI Mode      : DIO
I (41) boot.esp32c3: SPI Flash Size : 2MB
I (46) boot: Enabling RNG early entropy source...
...
I (228) spi_flash: detected chip: generic
I (232) spi_flash: flash io: dio
I (236) sleep: Configure to isolate all GPIO pins in sleep state
I (243) sleep: Enable automatic switching of GPIO sleep configuration
I (250) cpu_start: Starting scheduler.
Hello world!
This is esp32c3 chip with 1 CPU core(s), WiFi/BLE, silicon revision 3, 2MB
external flash
Minimum free heap size: 329676 bytes
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
Restarting in 6 seconds...
```

After booting up, it will eventually reboot and display "Hello world!" as often as you allow it to continue. To exit monitor mode, type Control-] (control key plus the right square bracket). Congratulations, you have successfully compiled, flashed and ran your first RISC-V program!

2.4. Summary

This may have been a tedious chapter for getting your Espressif software ready. The good news is that the difficult part is over and that your ESP32-C3 is now at your beckon call.

Chapter 3 • Installation and Setup of QEMU



QEMU speaks RISC-V!

In this chapter, instructions are provided for installing QEMU for the RISC-V emulator. Running the QEMU emulator allows us to explore the 64-bit RISC-V machine without having actual hardware for it. Additionally, it provides us debugger access to the machine in a friendly Linux environment.

When there is no QEMU binary install package available for your Linux desktop, it can be compiled from source code. The procedure for that is provided in this chapter. Otherwise, Windows and MacOS users can download and use prebuilt binary installs instead. Windows users should skip down to the section "Installing QEMU on Windows" near the end of this chapter.

Book Source Code

All users should check out the source code available for this book somewhere convenient. How and where you can download it will vary according to whether you are using Windows or not.

3.1. Linux/MacOS Platforms:

Throughout this book, I'll assume that the directory used for the checked-out code is under `~/riscv/repo`. Be sure to use the `-b master` option to checkout the correct branch.

```
$ git clone https://github.com/ve3wgg/risc-v.git -b master ~/riscv/repo
```

Copy the file:

```
$ cp ~/riscv/repo/boot.sh ~/riscv/boot.sh
```

This provides a copy of the checked-out file at the top of your `~/riscv` tree.

3.2. Windows

Windows makes the process a little more difficult, so follow these recommended steps:

1. Double-click your desktop ESP-IDF 4.4 CMD icon that was installed when you installed the ESP-IDF framework. Using this environment will give you instant access to the ESP-IDF installed git command.
2. Change to the C:\riscv after you create the directory.
3. Type the following git command using *forward slashes* to clone the repository.

```
C:> git clone https://github.com/ve3wwg/risc-v.git -b master /riscv/repo
```

After this is done, there should be files populated in your C:\riscv\repo subdirectory.

Basic Steps

There are two basic steps required to get our RISC-V system environment setup:

1. Install the QEMU RISC-V 64-bit emulator.
2. Install the Fedora Linux for RISC-V 64-bit (within the emulator).

In order to complete both of these steps, you need to determine if you have sufficient disk space. You will need at least 10 GB of space for the Fedora Linux system image file. But since it downloads as a compressed file, you need additional space to uncompress it. Plan on about 20 GB (some of this space is only temporarily needed). You will also need about 500 MB for the *installed* QEMU software. If you plan to build QEMU from source code on Linux, you will need about another 1 GB of disk space. These are rough estimates that should provide you with minimum guidelines.

3.3. Installing QEMU on MacOS

Detailed instructions for a MacOS binary installation can be found here:

```
https://www.qemu.org/download/#macos
```

These installations use the Homebrew or the MacPorts collections. If you don't use either of these yet, then you will need to install Homebrew or MacPorts first. I prefer the HomeBrew (<https://brew.sh>) collection myself.

3.4. Install QEMU on Devuan Linux

If you prefer to have a systemd free version of Linux and run the Devuan Chimaera release (<https://www.devuan.org>), then you can simply install QEMU from a binary package (as root):

```
# apt install qemu-system-misc
```

This installs all QEMU emulators, but check that the riscv64 version is the one installed with:

```
$ qemu-system-riscv64 --version
QEMU emulator version 5.2.0 (Debian 1:5.2+dfsg-11+deb11u1)
Copyright (c) 2003-2020 Fabrice Bellard and the QEMU Project developers
```

you should get a version confirmation for riscv64.

3.5. QEMU Package Search

Other Linux distros likely offer QEMU installable packages. Be sure to look for riscv64 when searching (unless it is bundled in `qemu-system-misc`). Package management varies depending upon the distro used. Many Debian based distros use the "apt" command for package management, for example. You can test for package availability with:

```
# apt search qemu
```

At the time of writing, for example, an old Devuan 32-bit Linux system supported only the following QEMU packages:

```
qemu-system/stable,stable-security 1:5.2+dfsg-11+deb11u1 i386
qemu-system-arm/stable,stable-security 1:5.2+dfsg-11+deb11u1 i386
qemu-system-mips/stable,stable-security 1:5.2+dfsg-11+deb11u1 i386
qemu-system-ppc/stable,stable-security 1:5.2+dfsg-11+deb11u1 i386
qemu-system-sparc/stable,stable-security 1:5.2+dfsg-11+deb11u1 i386
qemu-system-x86/stable,stable-security,now 1:5.2+dfsg-11+deb11u1 i386
```

In this example platform, there is no `qemu-system-riscv64` listed. If your Linux also lacks a package for riscv64, then don't despair. It can be compiled and installed from source code.

If you now have QEMU support for riscv64 installed, then skip the build instructions and resume at the section "Setup of Fedora Linux".

3.5.1. Building QEMU on Linux

In the Linux command examples shown later, command lines using a "\$" prompt are performed from your own developer account (not as root). Where the prompt is shown as "#", these commands must be executed from the root account or using the `sudo` command.

On the command lines shown, any text following the "#" character indicates a comment, which is ignored by the shell (from the point of the "#" character to the end of the line). These are comments about why the command is necessary etc. Don't type those in.

3.5.2. Basic Build Steps

When building QEMU from sources on Linux, we will perform the following steps:

1. Create a working directory with sufficient disk space.
2. Check out the QEMU source code from gitlab.com.
3. Configure it (and check for prerequisites).
4. Build (compile) it.
5. Install it.

The entire process is relatively painless but may require considerable time on slower platforms. Let your computer perform most of the work!

Create a Working Directory

For the purpose of building QEMU from source code, you need to create a working directory on a file system with at least 2 GB of storage. I'll assume that it is called "~/work" but you can choose any location you prefer.

```
$ mkdir ~/work
$ cd ~/work
```

Clone from gitlab.com

Check the source code out from can be checked out from gitlab.com as follows:

```
$ git clone https://gitlab.com/qemu-project/qemu.git
```

Depending upon your internet download speed, this might take a while. Once it has been completed, change to the checked-out source code directory:

```
$ cd ~/work/qemu
```

Open-source projects often experience significant changes. So that you can follow this text with fewer speed bumps, I strongly recommend that you use the same version as used in the book. To do this, change the version of the checked-out source code as shown:

```
$ git checkout v6.0.0-rc5 # Checkout this specific version
```

This will modify the source files to build the exact same version of QEMU that was used by the author here.

Configuration

The configuration and build are performed in a subdirectory named "build" that you must create. Create the subdirectory and then change to it:

```
$ mkdir -p ~/work/qemu/build
$ cd ~/work/qemu/build
```

If you ever need to start over because of problems, remove the build directory and recreate it.

Next, configure the build using the supplied configure script. Be sure to supply the "--target-list" option as shown because we only want the riscv64 version of the emulator. Otherwise, you will end up building many emulators, which can take a very long time.

```
$ cd ~/work/qemu/build
$ ../configure --target-list=riscv64-softmmu
...
                libdaxctl support: NO
                libudev: NO
                FUSE lseek: NO

Subprojects
                libvhost-user: YES

Found ninja-1.10.2.git.kitware.jobserver-1 at /usr/local/bin/ninja
```

If you don't have ninja installed, you may need to do so now (as root). For Devuan Linux for example, the following is installed:

```
# apt install ninja-build
```

There may be other dependency packages needed like:

```
# apt install pkg-config
# apt install libglib2.0-dev
# apt install libpixman-1-dev
```

The package names often vary by Linux distro. Always choose a "dev" (or devel) version of the package when it is available since it includes the header files needed for compiling. If these package names don't work for you, then copy the error message to your browser and find out what others used to solve the dependency.

After installing the missing dependencies, simply repeat the configure script as shown above. If all goes well, you should get to the end of the configure script successfully.

Build QEMU

After the configure step ends successfully, QEMU can be compiled from the build directory that you created.

```
$ cd ~/work/qemu/build
$ make
```

This build process can take a long time depending upon your Linux system. The compile can proceed without further user involvement so take a moment to make a coffee or tea or take your faithful dog for a walk in the park. At the end of the build, you should see messages similar to these:

```
[2137/2137] Linking target tests/qtest/qos-test
AS      multiboot.o
BUILD  multiboot.img
BUILD  multiboot.raw
SIGN   multiboot.bin
AS      linuxboot.o
BUILD  linuxboot.img
BUILD  linuxboot.raw
SIGN   linuxboot.bin
CC      linuxboot_dma.o
BUILD  linuxboot_dma.img
BUILD  linuxboot_dma.raw
SIGN   linuxboot_dma.bin
AS      kvmvapic.o
BUILD  kvmvapic.img
BUILD  kvmvapic.raw
SIGN   kvmvapic.bin
AS      pvh.o
CC      pvh_main.o
BUILD  pvh.img
BUILD  pvh.raw
SIGN   pvh.bin
$
```

Install QEMU

Once QEMU has been successfully compiled, you can install its components as follows:

```
$ cd ~/work/qemu/build
$ sudo make install
```

Congratulations! You have installed QEMU! Now verify that the emulator is available from the command line:

```
$ qemu-system-riscv64 --version
QEMU emulator version 5.2.95 (v6.0.0-rc5)
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
$
```

If that fails, check your PATH variable. Because you built QEMU from a specific version of the software, you should see the version number "v6.0.0-rc5".

Reclaiming Disk Space

If you're short of disk space and you were able to display the emulator version successfully after installation, then you can release the downloaded QEMU source code.

```
$ cd ~/work      # change to your work directory
$ rm -fr ./qemu # release all of the gitlab source code for QEMU
```

3.5.3. Linux/macOS Setup of Fedora Linux

In this section, I assume that you've successfully installed QEMU for RISC-V 64-bit, either from a binary package or from the source code. With that out of the way, let's turn our attention to downloading and setting up the RISC-V 64-bit version of Fedora Linux. Here, I'll assume that your working directory is ~/riscv:

```
$ mkdir ~/riscv
$ cd ~/riscv
```

Disk Space Requirements

In this section, we'll be downloading file system images for RISC-V Fedora Linux. For this, you're going to need a minimum of about 20 GB by the time the downloaded image is decompressed. Some of that space will be reclaimed.

Download Image Files

The Fedora Linux for RISC-V file system images that we're going to work with are available for download from this site:

<https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-images/images/>

At the time of writing, the following files of interest are available:

```
Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.elf
2020-01-13 16:48 592K
Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.elf.
CHECKSUM 2020-01-13 16:48 151
Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz
2020-01-13 16:55 1.4G
Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz.CHECKSUM 2020-01-13 16:58 125
```

If these particular files are no longer available when you read this, then choose the newer versions of these files. Keep in mind that disk space requirements may differ slightly.

Download Files

Download the .elf file required as shown below. On MacOS or Linux use the convenient wget command while in the ~/riscv directory:

```
$ wget 'https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-images/
images/Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.
elf'
```

You should also download the CHECKSUM file for verification, to make sure it has not been tampered with:

```
$ wget 'https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-images/
images/Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.
elf.CHECKSUM'
```

Now verify the downloads with the sha256sum command. Note the checksums reported, shown underlined here:

```
$ sha256sum 'Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-
smode.elf'
5ebc762df148511e2485b99c0bcf8728768c951680bc97bc959cae4c1ad8b053  Fedora-
Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.elf
$ cat 'Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.
elf.CHECKSUM'
SHA256 (Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.
elf) = 5ebc762df148511e2485b99c0bcf8728768c951680bc97bc959cae4c1ad8b053
```

Because the checksums match, we have confidence that the file has not been tampered with. You can now discard the CHECKSUM file if you wish.

We want the "developer" images of Fedora Linux to save us the trouble of installing compilers and other build tools. It also gives us an image with enough disk space to work in. So, download the following Fedora disk image file:

```
$ wget 'https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-images/
images/Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz'
```

and its matching CHECKSUM file. Once again, verify the checksum on the *compressed file*. When the checksums agree, decompress the file system image:

```
$ unxz Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz
$ ls -l Fedora-Developer-Rawhide-20200108.n.0-sda.raw
```

With the .elf and .raw files available, it is now time to boot into Fedora Linux using QEMU.

3.5.4. Linux/MacOS Boot Fedora Linux

There are a number of command line options needed by QEMU to successfully boot Linux. For this reason, we'll use a shell script so that you don't need to type them out. When you checked out the source code, you should already have copied this script:

```
$ cp ~/riscv/repo/boot.sh ~/riscv/boot.sh
```

The local copy of this script is available for your own customization. Review the section [Book Source Code](#) earlier, if necessary, if you've not yet checked out the source code.

The contents of the `~/riscv/boot.sh` script file should resemble this:

```
#!/bin/bash

exec qemu-system-riscv64 \
  -nographic \
  -machine virt \
  -smp 2 \
  -m 2G \
  -kernel Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-
smode.elf \
  -bios none \
  -object rng-random,filename=/dev/urandom,id=rng0 \
  -device virtio-rng-device,rng=rng0 \
  -device virtio-blk-device,drive=hd0 \
  -drive file=Fedora-Developer-Rawhide-20200108.n.0-sda.raw,format=raw,id=hd0 \
  -device virtio-net-device,netdev=usernet \
  -netdev user,id=usernet,hostfwd=tcp::10000-:22
```

Give the script file the appropriate execution permissions now:

```
$ chmod ug+rx ~/riscv/boot.sh
```

If your system fails to find the `qemu-system-riscv64` command, then its directory needs to be added to your `PATH` environment variable. MacOS for example, may have the emulator installed in `/usr/local/bin` when using HomeBrew. If this directory is not in your `PATH`, then add it now:

```
$ PATH="/usr/local/bin:$PATH"
```

Now it is time to test the boot into Fedora Linux.

3.5.5. Linux/MacOS Boot Test

Start the boot by invoking the script:

```
$ cd ~/riscv
$ ./boot.sh
```

The boot process *may seem to hang* when coming up:

```
[ 0.000000] printk: bootconsole [ns16550a0] enabled
```

But be patient and with time, it will progress beyond that point. The entire process can be quite lengthy on a little 32-bit Linux with one CPU. Modern systems should manage it better. Once Fedora comes up, you should be greeted with the following information:

```
Welcome to the Fedora/RISC-V disk image
https://fedoraproject.org/wiki/Architectures/RISC-V

Build date: Wed Jan  8 10:28:16 UTC 2020

Kernel 5.5.0-0.rc5.git0.1.1.riscv64.fc32.riscv64 on an riscv64 (ttyS0)

The root password is 'fedora_rocks!'.
root password logins are disabled in SSH starting Fedora 31.
User 'riscv' with password 'fedora_rocks!' in 'wheel' group is provided.

To install new packages use 'dnf install ...'

To upgrade disk image use 'dnf upgrade --best'

If DNS isn't working, try editing '/etc/yum.repos.d/fedora-riscv.repo'.

For updates and latest information read:
https://fedoraproject.org/wiki/Architectures/RISC-V

Fedora/RISC-V
-----
Koji:          http://fedora.riscv.rocks/koji/
SCM:          http://fedora.riscv.rocks:3000/
Distribution rep.: http://fedora.riscv.rocks/repos-dist/
Koji internal rep.: http://fedora.riscv.rocks/repos/
fedora-riscv login:
```

It is important to note the account and password provided in the greeting:

After the boot process is completed, you can log in using `riscv` as login and `fedora_rocks!` as the password.

You can log in at the console, but you may prefer to login using `ssh` instead (note that the account name is `riscv`):

SSH is enabled so you can `ssh` into the VM through port 10000 using the following command.

```
ssh -p 10000 -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no
-o PreferredAuthentications=password -o PubkeyAuthentication=no riscv@
localhost
```

You may find that a simple ssh command as follows is good enough:

```
ssh -p 10000 riscv@localhost
```

Root Access

You will need root access to administer your Fedora Linux system from time to time. To change to root from the riscv account, use sudo using the password fedora_rocks!:

```
[riscv@fedora-riscv ~]$ sudo -i

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for riscv:
[root@fedora-riscv ~]#
```

Check Disk Space

If you downloaded the suggested Fedora image file, you should have enough disk space available within this Fedora image to work on the projects for this book:

```
[riscv@fedora-riscv ~]$ df -k .
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/vda4        9539456 5345064   4080428   57% /
[riscv@fedora-riscv ~]$ bc -l <<<'4080428/1024/1024'
3.89139938354492187500
```

From the example shown, there are nearly 4 GB of free space. When you boot up a new instance of Linux there is a strong urge to perform system updates. *Resist that temptation* because by doing so, you could end up running out of space once the updates are applied.

SSH Authentication

Unfortunately, Fedora Linux requires a long password, which is very annoying in a hosted environment like this. If you want easier SSH access, follow the optional instructions found at <https://kb.iu.edu/d/aews> to set up public key authentication. This will allow you to log in without a password. Otherwise, simply choose a password of the required length that is easy enough for you to use.

Source Code Checkout for Fedora

Once you have logged into Fedora's riscv account, you'll want to check out the book's source code for use within Fedora Linux. After you log in, you will be in your home directory as shown:

```
[riscv@fedora-riscv ~]$ pwd
/home/riscv
```

While it's a little confusing, create a subdirectory named `riscv`:

```
[riscv@fedora-riscv ~]$ mkdir riscv
```

This creates `~/riscv/riscv` (or `/home/riscv/riscv`). Then check out the book source code using (don't forget the `-b` master option to check out the main branch):

```
$ git clone https://github.com/ve3wwg/risc-v.git -b master ~/riscv/repo
```

This places the book's source code in the directory `~/riscv/repo` (which is `/home/riscv/riscv/repo` in Fedora Linux). If you check it out exactly like this, the text references in the book will match.

Fedora Shutdown

Now that your RISC-V instance of Fedora Linux is running, let's review how to shut it down. If things go terribly wrong, like hanging at boot time, it is usually possible to `^C` (Control-C) out of it in the session where you launched `~/riscv/boot.sh`. If that fails, you can kill the process using the `kill` command.

A normal shutdown is performed in Fedora Linux from the root in the usual way:

```
# sudo /sbin/shutdown -h now
```

On the console session, you will eventually see the message:

```
[ 2425.326258] reboot: Power down
```

3.6. Installing QEMU on Windows

Installable QEMU executables are available from the following site for 32- or 64-bit versions of Windows:

<https://www.qemu.org/download/#windows>

First click on "Windows" and then you will see the text below:

Stefan Weil provides binaries and installers for both [32-bit](#) and [64-bit](#).

Click on the appropriate link for your version of Windows. Navigate through the directory of offered versions and then download and run the installer chosen. The download is approximately 192 MB in size.

QEMU Install

Once you have downloaded the installer, launched it, and answer Yes to the prompt "Do you want to allow this app from an unknown publisher to make changes to your device?" Select your language (English by default) and click OK.

In the "Welcome to QEMU Setup" dialog, click "Next". Agree to the license by clicking "I Agree". In the "Choose Components" dialog, leave everything checked that is checked and click "Next". Next, you will choose a folder into which you place the installed files. By default, it will be "C:\Program Files\qemu" (a recommended choice for this book). Change it or leave it as you wish and click "Install". Then click "Finish" when it completes the installation. Figure 3.1 illustrates the contrast in physical sizes of the two systems: ESP32-C3 sitting on top to the Dell Windows 10 tower below.



Figure 3.1: Size contrast of the ESP32-C3 dev board with a typical Windows-10 tower.

Download Fedora Linux

With the emulator installed we must now download Fedora Linux to run within it. Make a convenient directory named C:\riscv:

```
PS C:> mkdir \riscv
PS C:> cd \riscv
```

Now you can download the .elf file needed and the .raw image files:

```
PS C:> wget 'https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-  
images/images/Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-  
smode.elf'
```

After the .elf file is downloaded, download the raw image file.

```
PS C:> wget 'https://dl.fedoraproject.org/pub/alt/risc-v/repo/virt-builder-  
images/images/Fedora-Developer-Rawhide-20200108.n.0-sda.raw.xz'
```

Note: If you don't have the wget command, use your browser to download the image files and move them into the C:\riscv directory.

If you want to verify the checksums, see the Linux section about using sha256sum. You'll likely also need to install the sha256sum command under Windows. This step is optional.

The image file needs to be decompressed. There are freely downloadable apps like the ones at <https://tukaani.org/xz/>, or if you have PKZIP already installed, that will be able to uncompress it. After uncompressing the file, you should have the following two files ready for use:

```
C:\riscv\Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-smode.elf  
C:\riscv\Fedora-Developer-Rawhide-20200108.n.0-sda.raw
```

With the .elf and .raw files ready, it is time to boot into Fedora Linux using QEMU.

The boot.bat script

The options necessary to boot Fedora Linux under Windows differs somewhat from the Linux/MacOS version of QEMU. The file C:\riscv\boot.bat is a local copy from C:\riscv\repo\boot.bat/ This is available for you to customize, if necessary. The content of that file should resemble this:

```
@REM boot.bat  
  
PATH="C:\Program Files\qemu:%PATH%"  
  
cd C:\Program Files\qemu  
qemu-system-riscv64 -nographic -machine virt -smp 2 -m 2G ^  
-kernel \riscv\Fedora-Developer-Rawhide-20200108.n.0-fw_payload-uboot-qemu-virt-  
smode.elf ^  
-bios none ^  
-device virtio-blk-device,drive=hd0 ^  
-drive file=\riscv\Fedora-Developer-Rawhide-20200108.n.0-sda.raw,format=raw,id=hd0  
^  
-device virtio-net-device,netdev=usernet ^
```



```
-netdev user,id=usernet,hostfwd=tcp::10000-:22

@REM end boot.bat
```

The very long command line is broken into segments using the caret (^) character. Edit this file if you have different drive letters, directory, or file names. Once your .bat file is ready, launch QEMU to boot up Fedora Linux:

```
PC C:> cd \riscv
```

```
PC C:> .\boot.bat
```

If this fails to launch, check for spelling errors in the file names in the boot.bat file and correct them. The boot process *may seem to hang* when coming up but don't despair:

```
[ 0.000000] printk: bootconsole [ns16550a0] enabled
```

With time, it will progress beyond that point. Once it comes up, you should be greeted with the following information:

```
Welcome to the Fedora/RISC-V disk image
https://fedoraproject.org/wiki/Architectures/RISC-V

Build date: Wed Jan  8 10:28:16 UTC 2020

Kernel 5.5.0-0.rc5.git0.1.1.riscv64.fc32.riscv64 on an riscv64 (ttyS0)

The root password is 'fedora_rocks!'.
root password logins are disabled in SSH starting Fedora 31.
User 'riscv' with password 'fedora_rocks!' in 'wheel' group is provided.

To install new packages use 'dnf install ...'

To upgrade disk image use 'dnf upgrade --best'

If DNS isn't working, try editing '/etc/yum.repos.d/fedora-riscv.repo'.

For updates and latest information read:
https://fedoraproject.org/wiki/Architectures/RISC-V

Fedora/RISC-V
-----
Koji:          http://fedora.riscv.rocks/koji/
SCM:          http://fedora.riscv.rocks:3000/
Distribution rep.: http://fedora.riscv.rocks/repos-dist/
Koji internal rep.: http://fedora.riscv.rocks/repos/
fedora-riscv login:
```

See the prior section "Linux/macOS Boot Test". For more information about logging into Fedora Linux:

- Root Access
- Check Disk Space
- SSH Authentication
- Source Code Checkout for Fedora
- Fedora Shutdown

Related to this, I found that the Windows version of QEMU may sometimes stall at shutdown. Pressing RETURN at the console got it going again. Note also that the console may not be the best place to login since the terminal support may not support ANSI escape sequences, affecting its display. Use SSH for login, perhaps from PuTTY (<https://www.putty.org>) or from a Linux or macOS machine. Use the ipconfig command in the PowerShell to list your network address, if necessary. Once you know that, you can log in with SSH from any network connected machine using ssh:

```
$ ssh -p 10000 riscv@192.168.1.44 # Example IP number
```

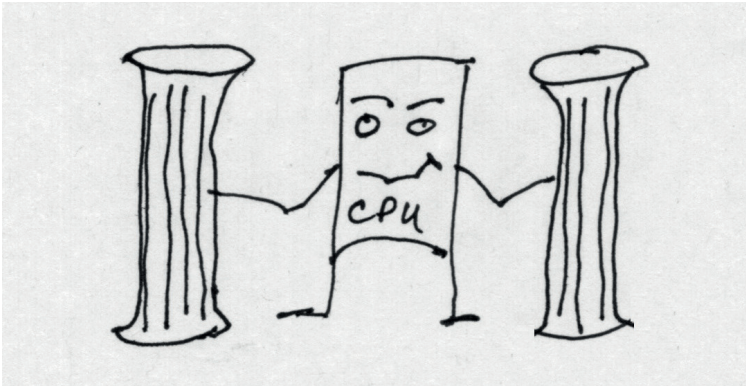
Make sure that you specify the port as *ten thousand* and not as one thousand, as the author is prone to do. If you still run into trouble, try the verbose version of the command below:

```
$ ssh -p 10000 -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o PreferredAuthentications=password -o PubkeyAuthentication=no riscv@192.168.1.44
```

3.7. Summary

This chapter has been an intense software setup chapter. But having this out of the way and the QEMU emulator running RISC-V in 64-bit mode under Linux, macOS or Windows opens many doors for our exploration in the remainder of the book. Fasten your seat belt.

Chapter 4 • Architecture



A CPU is known by its architecture

Before a programmer can program in assembly language, he/she needs to know something about the machine's architecture. What registers are available? What flag bits are evaluated? How is memory accessed? What are the available opcodes? In short, we need to know how the machine is organized.

Welcome to the Machine!

The best part of assembly language is the fact that you have full control over "the machine". You're not specifying C language code that is indirectly re-interpreted by the compiler into the machine's native language. No, you're specifying to the machine exactly what you want it to do. But to do that, you need to know about the resources that you have available.

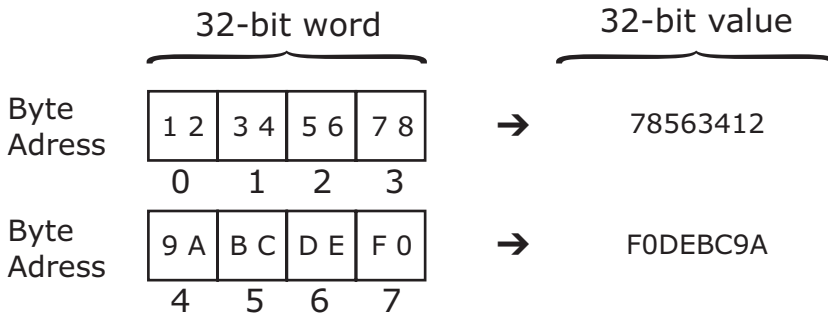
In the following sections, the RISC-V value of XLEN is the number of bits for the corresponding architecture. For the ESP32-C3 for example, XLEN is 32 (for 32 bits). For the QEMU emulated RISC-V 64-bit CPU, XLEN is 64 bits. So let's start with the program counter register.

4.1. Program Counter Register

Every CPU includes a program counter (PC) register. Unless an instruction is branching, this counter increments to form the address of the next instruction to perform. Standard RISC-V instructions have a 32-bit word length and must be aligned on 32-bit boundaries. There is provision for extending the instruction set to include variable length instructions based upon parcels of half-words (16-bits), which then must be half-word aligned. This allows for compressed instructions.

4.2. Endianness

The RISC-V memory architecture is little endian based and is *byte addressable*. Little endian means that a word (of 4 bytes) orders the bytes with least significant bytes first (lowest addressable unit). The last (highest addressed) byte of the word is the most significant byte. Figure 4.1 illustrates two 32-bit words with byte addresses and their word values at right.



(all numbers are hexadecimal)

Figure 4.1: RISC-V Byte Order (Little Endian). All numbers expressed in hexadecimal.

Given that the word length is 4 bytes, a normal instruction word fetch will increment the program counter by 4. For compressed instructions consisting of 2-byte parcels an instruction fetch will increment the program counter by some multiple of 2, depending upon the instruction's length.

4.3. General Purpose Registers

Most Central Processing Units (CPU) designs today include one or more operation registers. The operations occur between registers or between registers and memory. Registers are similar to memory words except that they are instantly available to the CPU and fast. Memory fetches or stores by contrast require more time to complete.

The width of the register is defined by constant XLEN in the given architecture. So, registers are 32-bits wide for the ESP32-C3 where XLEN=32. Registers are 64-bits wide for the QEMU emulated CPU where XLEN=64. For most architectural subsets (discussed in the next section) have 32 general purpose registers available. These are known as registers X0 through X31. We shall see that they can also be referred to by different names and that register X0 has a special talent.

4.4. Introducing Subsets

The RISC-V design breaks the architecture design into subsets with names like RV32I. Subsets allow manufacturers to implement only the portion of the architecture that they need, while leaving out other aspects of the design. An small embedded processor, for example, may implement the minimal RISC-V design to reduce component complexity, power consumption and overall cost. Alternatively, a general-purpose server design for Linux may implement most or all of the defined architecture subsets for maximum flexibility.

The most basic architectural subset is RV32I. This defines the most basic opcodes, register set and memory operations as a "base architecture". The "32" indicates that the architectural width, is 32 bits (XLEN=32). So, in RV32I we know that the register size is 32 bits wide.

A minimal RISC-V 64-bit implementation is defined as the RV64I subset. This expands upon RV32I by widening the registers to 64 bits (XLEN=64). A few additional operations are added to allow for loading and storing of 64-bit register values.

There also exists the RV32E subset for use in embedded processors where *minimum cost* and low power are the focus. In this unusual subset, with the number of registers reduced to 16 but keeping with the register width of 32 bits (XLEN=32). The reasoning for this subset is explained in [1]:

We have found that in the small RV32I core designs, the upper 16 registers consume around one quarter of the total area of the core excluding memories, thus their removal saves around 25% core area with a corresponding core power reduction.

There are other basic subsets such as the RV128I, which widens XLEN to 128. But we'll focus mainly on RV32I and RV64I in this book. It is easiest to learn when the abstract is replaced with the concrete.

There are other capabilities described by RISC-V such as RV32M for example, which defines the operations necessary for integer multiply and divide. RV32C describes the provision for compressed instruction opcodes, allowing for greater code density. The ESP32-C3 device implements RV32IMC, which indicates that the basic subset RV32I, multiply and divide RV32M and compressed opcodes RV32C subsets are supported (XLEN=32). We'll review the features of the QEMU 64-bit CPU later in this book.

4.5. Register Specifics

There are 32 basic registers provided by RV32I and RV64I. These are known by symbolic names x0 to x31. Because of the number of registers involved and their varied usage, the Application Binary Interface (ABI) names are preferred instead. For example, register a0 is often used as the first calling argument or return value rather than x10. The code used in this book will prefer the ABI register names shown in Table 4.1. The usage described in that table are according to the GNU compiler convention for RISC-V.

Register	ABI Name	Description	Saver
x0	zero	Hardwired to return zero	-
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	-
x4	tp	Thread Pointer	-
x5-x7	t0-t2	Temporary Registers	Caller
x8	s0/fp	Saved register / Frame Pointer	Callee
x9	s1	Saved Register	Callee
x10-x11	a0-a1	Function arguments / return value	Caller
x12-x17	a2-a7	Function arguments (continued)	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

Table 4.1: RISC-V Basic Registers, of XLEN bits.

The register x0 (zero) is special to RISC-V in that it always returns the value zero when used as a source register or discards a value when it is used as a destination. This permits some creativity in the opcode's effect. It is specified as x0 or simply as zero.

The remainder of registers, from x1 to x32 are each XLEN bits wide and can be used as source or destination. Each register has an ABI name assigned to make it easier to write code that uses the registers in a consistent way. For example, there are temporary registers that are named t0 through t6.

The rightmost column of Table 4.1 indicates who saves the register when saving is required, according to the GNU calling convention. Either the calling code (caller) or the called code (callee) must save the register, if it is modified. Registers marked with "-" like registers gp or tp, have no requirement for being preserved. The description field indicates how these registers are typically used.

4.5.1. No Flag Bits

One unique characteristic of the RISC-V architecture is that it does not define a status register containing any "flag bits". Many architectures like Intel, define a Z (zero), C (carry) and other flag bits that are updated during the execution of some opcodes. The benefit of flag bits is that a ready computed state is made available. But a disadvantage is that flag bits (in a status register) must be saved and restored when interrupts occur or across function calls. The RISC-V designers have decided that flag bits are a complication to be avoided. Some RISC-V instructions do, however, store a flag bit in the destination register when it is needed.

You might wonder how to handle multi-precision unsigned integers without a Carry flag bit, for example. This and similar problems will be solved later in this book.

4.5.2. Register x0 / Zero

The x0 or zero register is special in RISC-V. When used as a *source* register, it always supplies a binary value of zero. When used as a *destination* register, it discards the result. This is a hard-wired feature of the CPU and cannot be changed. This special talent of x0 will frequently allow creativity in the computed results.

4.5.3. Register x1 / ra

The x1 or ra register is traditionally used as the *return address* register. This is by GNU calling convention and is not hard-wired. When a function is called, traditionally the return address is put into register ra. There are opcodes used by the calling code that arrange this.

4.5.4. Register x2 / sp

According to the GNU calling convention, the x2 or sp register is used as the stack pointer. This is by convention only. The code that is called (callee) is responsible for establishing this value. The called code normally adds a value to the current stack pointer to adjust the stack.

4.5.5. Register x3 / gp

According to the GNU calling convention for RISC-V, this register serves as a global pointer. This is by convention only. The use of this value may vary according to the platform.

4.5.6. Register x4 / tp

By GNU convention, this register is used for thread local storage access. The establishment of this value may vary by the platform supported.

4.5.7. Registers x5-x7 / t0-t2

These are temporary registers. When calling other functions and subroutines, the caller is responsible for saving these values if they need to be preserved across the call.

4.5.8. Register x8 / s0 / fp

The x8 register, has two basic purposes: s0 is the first saved register value, while fp represents a stack frame pointer. This value must be saved and restored by the called (callee) code if the register value is modified.

4.5.9. Register x9 / s1

Register x9 or s1, is similar to register s0. It is a second saved register value. The called (callee) code must save and restore this value when it is modified.

4.5.10. Registers x10-x11 / a0-a1

When a function call has an argument, the first argument goes into register x10 / a0. For RV32 (RISC-V 32-bit) this means that arguments up to the size of 32-bits are placed into a0. A 64-bit argument would have the least significant 32 bits placed in a0, while the most significant 32 bits get placed into a1. For RV64, where the register width is 64-bits, the first argument goes into entirely into register a0 (argument sizes of up to 64-bits) and the next argument into a1. This applies to integers and pointers. When there is no hardware floating point (i.e. soft-float), this also applies to floating point values according to their size.

4.5.11. Registers x12-x17 / a2-a7

Additional arguments are passed in registers a2 to a7, depending upon their size like registers a0 and a1.

4.5.12. Registers x17-x27 / s2-s11

These registers must be preserved by the called function (callee) if the register's content is modified in any way. These registers can be used for various purposes.

4.5.13. Registers x28-x31 / t3-t6

Additional temporary registers t3 through t6 are available for temporary use. The caller is responsible for saving these registers if the values must be preserved through a function call.

4.5.14. Register Summary

With the names and functions of each register out of the way, it might seem to the reader that they must be used in a rather rigid scheme. For RISC-V, the only register with a hardwired talent is register x0. It can supply zero or discard a value. All other registers are general purpose and used as described only by *convention*. In other words, all remaining registers can be used any way you choose. However, using the GNU calling convention makes for safer, consistent and readable code.

4.6. Instruction Set Base Subsets/Extensions

RISC-V defines several instruction subsets and extensions. The most basic set of base instructions is the "I" Base Integer subset. This defines the most basic integer-type operations that a programmer must have. Two examples include the RV32I (XLEN=32) and RV64I (XLEN=64) subsets. The instructions are normally fixed in size at 32-bits. The instruction formats are described by the document "Volume I: RISC-V User-Level ISA V2.2". There is, however, provision for extending that format with the "C" extension to allow compressed instructions in 16-bit half words. This can apply to RV32, RV64 and RV128 ISA's, and thus that extension is often referred to as "RVC".

The Espressif ESP32-C3 device, for example supports the RV32IMC ISA. This indicates that the base integer subset "I", and the "C" compressed extensions apply. The "M" in the RC32IMC indicates that the "M" extension is also supported, providing hardware multiply and divide instructions. The "32" by way of review means that the registers are 32-bits wide (XLEN=32).

There is another subset designated as the RV32E, intended for small embedded systems. This is a reduced version of RV32I so that instead of 32 registers, only the first 16 registers are available. This reduces the number of transistors required and also reduces the power requirements. In chapter 3 of "*The RISC-V Instruction Set Manual*", it is stated that:

RV32E was designed to provide an even smaller base core for embedded microcontrollers. ... However, given the demand for the smallest possible 32-bit microcontroller, and in the interests of preempting fragmentation in this space, we have now defined RV32E as a fourth standard base ISA in addition to RV32I, RV64I, and RV128I. The E variant is only standardized for the 32-bit address space width.

They also warn that:

This change requires a different calling convention and ABI. In particular, RV32E is only used with a soft-float calling convention. Systems with hardware floating-point must use an I base.

Table 4.2 provides a list of most of the RISC-V base subsets and extensions. This list is incomplete because some standards are still evolving.

Base	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers
RV64I	Base Integer Instruction Set, 64-bit
RV128I	Base Integer Instruction Set, 128-bit

Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
G	Shorthand for the IMAFDZicsr Zifencei base and extensions, intended to represent a standard general-purpose ISA
Q	Standard Extension for Quad-Precision Floating-Point
L	Standard Extension for Decimal Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
J	Standard Extension for Dynamically Translated Languages
T	Standard Extension for Transactional Memory
P	Standard Extension for Packed-SIMD Instructions
V	Standard Extension for Vector Operations
K	Standard Extension for Scalar Cryptography
N	Standard Extension for User-Level Interrupts
H	Standard Extension for Hypervisor
S	Standard Extension for Supervisor-level Instructions

Table 4.2: Partial list of base subsets and extensions of the RISC-V ISA.

4.7. ESP32-C3 Hardware:

Focusing on the CPU resource, Espressif states that the ESP32-C3 device supports the RV32IMC ISA. Therefore, the following base and extensions are supported:

- I - Base Integer Instruction Set, 32-bit
- M - Standard Extension for Integer Multiplication and Division
- C - Standard Extension for Compressed Instructions

4.8. QEMU RISC-V 64 Bit Emulator

What base and extensions are supported by the QEMU emulator installed in the last chapter? When you boot Fedora Linux under the emulator, you can discover the level of support by listing the special kernel file `/proc/cpuinfo`:

```
[root@fedora-riscv ~]# cat /proc/cpuinfo
processor      : 0
hart         : 0
isa          : rv64imafdcsu
mmu         : sv48

processor      : 1
hart         : 1
isa          : rv64imafdcsu
mmu         : sv48

processor      : 2
hart         : 2
isa          : rv64imafdcsu
mmu         : sv48

processor      : 3
hart         : 3
isa          : rv64imafdcsu
mmu         : sv48
```

From this list, it is apparent that the following RISC-V base and extensions are supported:

- I - Base Integer Instruction Set, 64-bit
- M - Standard Extension for Integer Multiplication and Division
- A - Standard Extension for Atomic Instructions
- F - Standard Extension for Single-Precision Floating-Point
- D - Standard Extension for Double-Precision Floating-Point
- C - Standard Extension for Compressed Instructions
- S - Standard Extension for Supervisor-level Instructions

Finally, the "u" that is listed just means that "user mode" (vs "supervisor mode") is supported. The "S" extension is necessary for Unix/Linux/*BSD systems that protect independent processes from corrupting each other's memory.

The A extension is important to multi-threaded applications where it is necessary to read/update memory from multiple threads (CPU cores) in a safe manner. If you have full hardware support for floating point calculations, then you want both of the "F" and "D" extensions.

4.9. RISC-V Privilege Levels

The RISC-V specification identifies four classes of privilege levels for a CPU, which are listed in Table 4.3. The most basic of these privilege modes is the "m" for machine mode. This is what is used on the ESP32-C3 device, where all executing code has full access to the "machine". Machine mode is the only mandatory privilege mode for RISC-V implementations. This mode is normally used on embedded systems with unrestricted access to all resources. Some implementations may also support "u" mode in addition to "m" to provide a secured embedded solution.

The "m", "s" and "u" modes together will be used by hardware (and emulators) that support Unix/Linux/*BSD type systems. The supervisor mode separates the execution of specialized kernel code (mode "s") from the unprivileged user code ("u" mode). This protects one process from another and provides strict access to shared resources like memory and peripherals.

The column labeled MISA in Table 4.3 refers to a special register with bits that define the ISA and extensions supported.

ID	Mode	MISA
m	machine	
s	supervisor	Bit: 18, 0x00040000
u	user	Bit: 20, 0x00100000
d	debug	

Table 4.3: RISC-V Privilege Levels.

What does all of this mean for you? On the ESP32-C3, you will strictly operate in machine mode. Whatever is supported by that hardware will be open to you to use and abuse. Thus for the ESP32-C3, for example, you will be able to inquire of the MISA special register to determine the level of support available, among other things.

Under Fedora Linux using the QEMU emulator, however, you will be running code in the "u" user mode. This means that your code will be restricted in what it can do, including the reading of the MISA register. If you try to read the MISA register, your code will fail. To determine the level of support available, you must rely on the Linux kernel's special file `/proc/cpuinfo`.

4.10. RISC-V is Huge

There has been an incredible amount of effort put into the RISC-V specification and it continues to evolve. For this reason, we will be focusing on writing assembly language code for the machine and user modes for this book. This will keep the level of detail to an easily-managed tutorial.

4.11. Summary

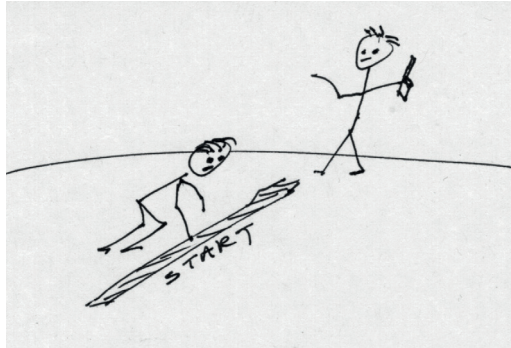
By this point in this chapter, I hope you are chomping at the bit to do some assembly-level code. With the basics of RISC-V out of the way, you are ready to write some code. So, let's get started!

Bibliography

[1] *The RISC-V Instruction Set Manual*.

<https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf>.

Chapter 5 • Getting Started



At the starting line

You've been introduced to the RISC-V ISA. You've installed the software. Now it is time to play! So let's get started with assembly language programming. To do that, we'll embark on a very small but digestible project. This will introduce several concepts without too much detail. When you complete this chapter, you will be ready to take on more advanced topics in assembly language programming.

5.1. Memory Models & Data Types

To begin our first assembler project, let's first discuss the different memory models used in our two RISC-V platforms. Recall that we're working with:

- ESP32-C3 device, which is an RV32 (32-bit) device
- qemu-system-riscv64 (QEMU), a 64-bit emulator for RV64

For the remainder of this book, I will simply use QEMU to refer to the qemu-system-riscv64 emulator. Now let's examine how the memory models differ between these two machine architectures in C language terms.

5.1.1. RV32 Model

If we were to write a short program to print out the C data type sizes in the RV32 model, we would get the following information:

```
sizeof(int) = 4
sizeof(long) = 4
sizeof(long long) = 8
sizeof(void*) = 4
```

These sizes are in bytes. We could summarize that the int, long integers and pointers share the same bit width of 32-bits (4 bytes). This fits the RV32I architecture where the registers are 32-bits in width. C programmers might be more familiar with the equivalent ILP32 model, which originates from the Solaris C language data model (Integers, Long and Pointers are 32-bits). While we didn't report it above, the sizeof(short) is 16-bits and the

character (byte) is 8-bits in size. The C/C++ compiler will also support the long long data type but multi-precision arithmetic is used on 32-bit platforms to achieve it.

5.1.2. RV64 Model

QEMU is fun to experiment with because it uses the RV64 architecture, which is a superset of RV32. The registers in this architecture are widened to 64-bits. When we use the same C program in QEMU to report the data type sizes, we would obtain the following:

```
sizeof(int)    = 4
sizeof(long)   = 8
sizeof(long long) = 8
sizeof(void*)  = 8
```

In other words, long integers and pointers are 64 bits wide (8 bytes), while sizeof(int) remains at 32 bits. This is equivalent to the LP64 Solaris model (Long and Pointers are 64-bit, while Integers are assumed to be 32 bits). Again, the sizeof(short) is 16 bits and the character 8 bits as before. The long long integer data type is also 64 bits, as it was for RV32 (or ILP32). However, the RV64 platform can naturally compute in 64 bits. Clint Eastwood in the movie Dirty Harry says that "a man's gotta know his limitations". Now that we are familiar with the two memory models, we know our data type limitations.

Table 5.1 summarizes the data type characteristics for the two different RISC-V profiles that we'll use.

C Type	ILP32 / RV32 (Bits)	LP64 / RV64 (Bits)
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64

Table 5.1: Memory Models.

5.2. The Impact of XLEN

Since we will be programming for both platforms, it is important to keep in mind how this impacts your code. It is critical when programming in assembler language because, unlike the C compiler, the assembler is not going to do automatic conversions between the two memory models.

While the register widths differ in these two models, the instruction set remains largely the same. When a 32-bit value is loaded into a 64-bit register, the value is automatically sign-extended into the 64-bit register. When the same register is stored into a 32-bit memory location, only the low order 32-bits are saved. In other words, the main impact of different register sizes will occur in the loading from and storing to memory. To accomplish

64-bit loads and stores, additional opcodes were added to RV64I. For now, just be aware of this and keep it in the back of your mind.

5.3. First Exercise

The work to be accomplished by this first exercise is trivial. We're going to simply add three 32-bit integer numbers in assembly language. This is something routinely performed in C/C++ but we're going to do this in assembly language to learn the ropes. This will bring together some concepts for calling assembler subroutines from C/C++.

5.3.1. The Main Program

Using the familiar C main program, an assembler program will be called upon to perform the actual addition. This will demonstrate three things:

- How to call an assembler function, instead of a C function.
- How to pass integer argument values to the assembler routine.
- How to return an integer result.

The main program is written as if it were calling another C function. The C compiler doesn't even know that the function `add3()` in this example is written in assembly language. The main program for this example is illustrated in Listing 5.1 and will be run from Fedora Linux under QEMU. Throughout this book, I'll use line numbers at the left of each listing for ease of reference. They are not part of the source file.

```
1 // main.c
2
3 #include <stdio.h>
4
5 extern int add3(int arg1,int arg2,int arg3);
6
7 int
8 main(int argc,char **argv) {
9     int a=23, b=24, c=25;
10    int r = 0;
11
12    r = add3(a,b,c);
13    printf("r = %d\n",r);
14    return 0;
15 }
```

Listing 5.1: Program `~/riscv/repo/05/add3/qemu64/main.c`.

Let's cover the highlights of the program:

- Line 3 is used to bring in support for the `printf()` function used in line 13.
- Line 5 is a declaration of our assembler subroutine, but in C language terms, where:
 - The function's (symbol) name is `add3`. The function is expecting three integer arguments of type `int`. The function will return an integer value.
 - The `extern` keyword informs the C compiler that the function `add3` is external to the current source file.
- Line 9 declares three integers `a`, `b` and `c` and initializes them with values.
- The function `add3` is invoked in line 12, with the result assigned to the variable `r` (declared in line 10).
- Line 13 reports the sum returned in `r`. And finally, line 14 just returns an exit code that Linux expects from the main program.

Keep in mind that the data type `int` is 32 bits in size. This applies to variables `a`, `b`, `c` and `r`.

5.3.2. Assembler Routine `add3`

The assembler routine is found in file `~/riscv/repo/05/add3/qemu64/add3.S`, which is illustrated in Listing 5.2.

```
1      .global add3
2      .text
3 add3:  add      a0,a0,a1      # a0 = a0 + a1
4      add      a0,a0,a2      # a0 = a0 + a2
5      ret                          # return value in a0
```

Listing 5.2: File `~/riscv/repo/05/ADD3/qemu64/add3.S`.

Now that you've seen the assembly language source file, let's discuss the general format used.

5.3.3. Assembly Language Format

The assembly language source file follows a general convention. Each source line consists of four fields, where fields 1, 3 and 4 are optional. All lines, except for comment only lines must contain an opcode or pseudo-op.

Field1	Field2	Field3	Field4
label:	opcode	operands	# comments...

Labels are optional but when used, start in the first column, followed by a colon. The label "add3:" is an example of a label found in line 3 of Listing 5.2.

The second field is the operation code (opcode), which can be an actual instruction opcode like "add" (see lines 3 and 4) or a pseudo operation like `.global` or `.text`. Pseudo opcodes influence the working of the assembler as it progresses from the start of the source file to its end.

Many opcodes require operands. These are listed in the third field and have subfields separated by commas.

Optionally, a fourth field starting with '#' marks the start of a comment on the line. A '#' in the first column indicates that the *entire* line is a comment. These lines are ignored by the assembler.

While the above represents the general convention for an assembler source line, GNU's assembler is somewhat forgiving. For example, you could have spaces preceding the label or spaces between the label and the colon character. I have used tab characters to separate the fields, but you can use spaces instead.

5.3.4. Pseudo Opcode `.global`

In Listing 5.2, line 1 indicates that there is a global symbol reference by the name of "add3". The GNU assembler also accepts the pseudo opcode `.globl` for historical reasons. This instruction to the assembler is important because it indicates that the symbol "add3" will be made known to the linker as an external symbol. The linker will link the main program's reference to `add3` to a symbol defined in this assembly source file. The symbol `add3` itself is defined by the label in line 3. That line defines the starting address for the `add3` function.

If the assembler source file was to call a function like `printf()` for example, the symbol `printf` should also be listed as a `.global` (there will be examples of this later in the book). References and defined symbols can be listed in the `.global` operands field. Multiple symbols can be given on one line, for example:

```
.global add3, printf
```

By default, all undefined symbols are assumed by the assembler to be external. However, it is best practice to list all global symbols defined or referenced in the program. As a result, it is easier to distinguish between an error of omission and an actual external symbol. This becomes critical in larger projects.

5.3.5. Pseudo Opcode `.text`

Line 2 of the assembler listing uses the `.text` pseudo code to indicate that the following code belongs to an object code section named `".text"`. This is the section normally used for executable code. The linker uses sections to group code and data into memory regions. An alternative way to indicate this is to use the `.section` pseudo code:

```
.section      .text
```

The `.text` section is assumed by default at the start of the assembly, but it is best to be specific. If nothing else, it helps the reader of your code know where the instructions are going to be assembled.

5.3.6. The add Opcode

Lines 3 and 4 of the listing use the RISC-V opcode `add`. This is an actual RISC-V instruction, which has three register operands: a destination register, followed by two source registers:

```
add3:  add a0,a0,a1
```

The label `"add3"` is defined as a global symbol because it was listed as a `.global` reference. The RISC-V instruction set architecture consists mainly of register-to-register and register load/store operations. The `"add"` instruction is a register-to-register operation that has three operands. Two registers are source registers (shown underlined) and one destination register. With the shown `add` instruction, register `a0` is added to `a1` with the result replacing the contents of register `a0`.

5.3.7. Calling add3

The C language compiler will arrange the call to our routine `add3` (review `main.c` line 12), with three arguments to be loaded:

1. Argument one is loaded into register `a0` (x10).
2. Argument two is loaded into register `a1` (x11).
3. Argument three is loaded into register `a2` (x12).

In this RV64I code, the `int` variables are still 32-bits in size (review Table 5.1). However, as these values are loaded into 64-bit registers, the values are sign extended to 64 bits.

The C language compiler will expect the 32-bit integer result to be returned in register `a0` (x10).

Note: Throughout this book, the friendly (ABI) register names are used. For example, we will refer to the register `a0` instead of `x10`. Either is legal in the source code but the friendly names are easier to work with. It is also kinder to the reader.

In line 3 (Listing 5.2), the value of the first argument arrives in register `a0`, with the second argument in `a1`. These two registers are added together, and the result replaces the value in `a0`. In C language terms, it amounts to:

```
a0 += a1;
```

Line 4 repeats the `"add"` instruction but this time referencing the third argument in register `a2`:

```
add          a0,a0,a2
```

This time the result of $a0 + a2$ replaces $a0$. In C language terms, the pair of instructions can be summarized as:

```
a0 += a1;
a0 += a2;
```

The calculated result is left in register $a0$ prior to the return to `main`. The opcode "ret" in line 5 causes the assembler routine to return to the caller. This works because the caller's return address has been placed in register `ra` (or `x1` – think "return address") when our `add3` routine was called. The `ret` instruction will be examined in more detail later in the book.

5.3.8. RV64I Consideration

While we declared the arguments and the return value as `int` type (32-bits) in the C program, everything works as expected, even for 64-bit RISC-V. This is because when the 32-bit values were loaded into registers `a0`, `a1` and `a2` by the calling C program, they were *automatically sign extended to 64 bits* to match the register size. The additions were also performed as 64-bit integers. While there is a 64-bit result produced in `a0`, the return value will be taken from the lowest 32-bits because the C program is expecting a 32-bit integer result.

5.3.9. Running the Demonstration

To run the demonstration (in QEMU), we must compile it first. Logged into your QEMU instance of Fedora Linux (see chapter 3 about running QEMU), compile this demonstration using the following command line:

```
$ cd ~/riscv/repo/05/add3/qemu64
$ gcc -O0 -g main.c add3.S
$ ls -l a.out
-rwxrwxr-x. 1 riscv riscv 13560 Apr 19 21:13 a.out
```

The compiled result is in file `a.out`, which can now be executed. Option `-O0` (dash capital oh zero) should be used to prevent any compiler optimization. Compilers today are very good at optimizing and might precompute the result without even calling the assembler routine at all. So we disable optimization to prevent that. The `-g` option is optional here, but I encourage you to use it in case you want to step through the code using a debugger like `gdb`. We'll make use of the debugger later.

The C compiler will compile `main.c` as a C program and `add3.S` as an assembler source module. After those successfully produce object modules, they are linked into a final executable named `a.out`. To execute our program under Linux use:

```
$ ./a.out
r = 72
```

This is the correct reported value for the sum.

Note: It is also possible to place assembly language into file `add3.s` (lowercase 's') but this causes the assembly to proceed *without the benefit of the C preprocessor*. Later in this book, we will want to use the macro capabilities of the preprocessor to make the assembly code portable. It is, therefore, recommended that you get used to using the capitalized suffix `.S` instead.

At this point, you can shut down your Fedora Linux instance (QEMU). We're now going to exercise the same code on the ESP32-C3 device.

5.4. First Exercise on ESP32-C3

In this exercise, we're going to apply the same exercise to the ESP32-C3. Recall that this device is a 32-bit platform. Use a separate terminal window because you'll need to initialize your ESP32 IDF to use the code specific to that MCU. Change to the subdirectory `05/add3`. The ESP32 source files are located in subdirectory `./main` but remain at this level for the build.

Note: Linux and MacOS users will want to initialize their ESP-IDF in a new terminal window with the alias established in chapter 2, or to do so manually as follows:

```
$ . ~/espc3/esp-idf/export.sh
```

Windows users can start a session just by double-clicking their ESP-IDF 4.4 CMD icon.

The ESP32 main program is somewhat different than what is used on Linux. Listing 5.3 illustrates:

```
1 #include <stdio.h>
2
3 extern int add3(int one,int two,int three);
4
5 void
6 app_main(void) {
7     int a=23, b=24, c=25;
8     int r = 0;
9
10    r = add3(a,b,c);
11    printf("r = %d\n",r);
12 }
```

Listing 5.3: ESP32-C3 main program `~/riscv/repo/05/add3/main/main.c`.

The main difference for ESP32 is that the main program is named `app_main`, taking *no arguments* and *returning no value*. The assembler file `~/riscv/repo/05/add3/main/add3.S` is otherwise identical to the file used for QEMU before.

It's always a good idea to start with a clean slate, so let's clean the project directory using the ESP-IDF:

```
$ cd ~/riscv/repo/05/add3
$ idf.py fullclean
Executing action: fullclean
Done
```

This guarantees that any partially built objects from experiments and failed compiles are removed so that your build will be done from scratch.

Now build your ESP32-C3 version of the add3 program. The first time you build after a fullclean, it can take a long time. But later builds will be swifter due to cached compiles:

```
$ idf.py build
Executing action: all (aliases: build)
Running cmake in directory /Users/ve3wwg/riscv/repo/05/add3/build
Executing "cmake -G Ninja -DPYTHON_DEPS_CHECKED=1 -DESP_PLATFORM=1 -DIDF_
TARGET=esp32c3 -DCCACHE_ENABLE=0 /Users/ve3wwg/riscv/repo/05/add3"...
...
Executing "ninja all"...
[10/955] Generating ../../partition_table/partition-table.bin
Partition table binary generated. Contents:
*****
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size, Flags
nvs,data,nvs,0x9000,24K,
phy_init,data,phy,0xf000,4K,
factory,app,factory,0x10000,1M,
*****
[502/955] Performing configure step for 'bootloader'
...
Bootloader binary size 0x4500 bytes. 0x3b00 bytes (86%) free.
[954/955] Generating binary image from built executable
esptool.py v3.2-dev
Merged 1 ELF section
...

Project build complete. To flash, run this command:
/Users/ve3wwg/.espressif/python_env/idf4.4_py3.9_env/bin/python ../../../../
esp32c3/esp-idf/components/esptool_py/esptool/esptool.py -p (PORT) -b 460800
--before default_reset --after hard_reset --chip esp32c3 write_flash --flash_
mode dio --flash_size detect --flash_freq 80m 0x0 build/bootloader/bootloader.bin
0x8000 build/partition_table/partition-table.bin 0x10000 build/add3.bin
or run 'idf.py -p (PORT) flash'
```

Now flash the device and monitor its output. In the example below, your path for the device will likely differ from the path `/dev/cu.usbserial-1430` shown (underlined). Windows users will specify a COM port instead.

```
idf.py -p /dev/cu.usbserial-1430 flash monitor
Executing action: flash
...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Executing action: monitor
Running idf_monitor in directory /Users/ve3wwg/riscv/repo/05/add3
...
ESP-ROM:esp32c3-apil-20210207
Build:Feb  7 2021
rst:0x1 (POWERON),boot:0xc (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fcd6100,len:0x15cc
load:0x403ce000,len:0x8ec
load:0x403d0000,len:0x25e8
entry 0x403ce000
I (30) boot: ESP-IDF v4.4-dev-2359-g58022f8599 2nd stage bootloader
I (30) boot: compile time 16:00:41
I (30) boot: chip revision: 3
...
I (256) cpu_start: Starting scheduler.
R = 72
```

When the program exits `app_main`, it just stalls within the ESP32 framework. Press Control-] (Control right square bracket) to exit the monitor process. Notice that the sum is reported correctly as 72. Congratulations, you have executed your first RISC-V instructions on the ESP32-C3!

Figures 5.1 and 5.2 illustrate a typical dev board PCB offering that includes a tiny OLED display purchased from AliExpress.

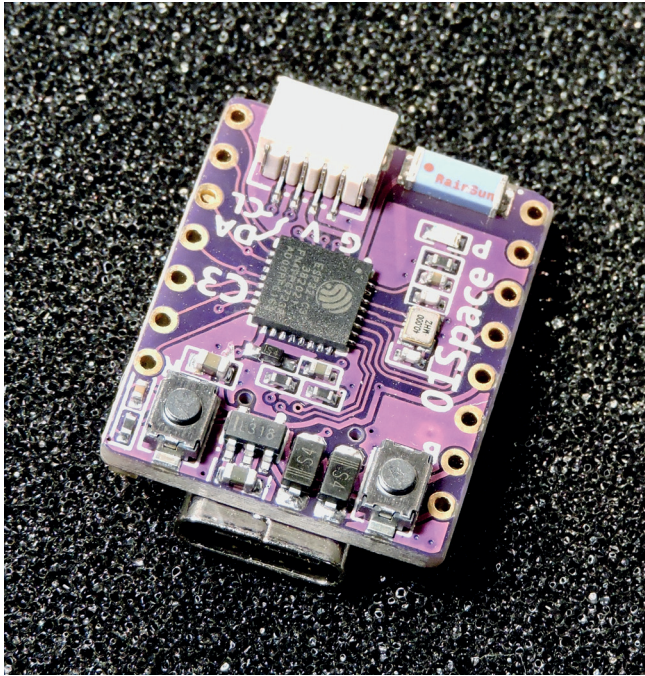


Figure 5.1: An ESP32-C3 device that includes an OLED (bottom side).

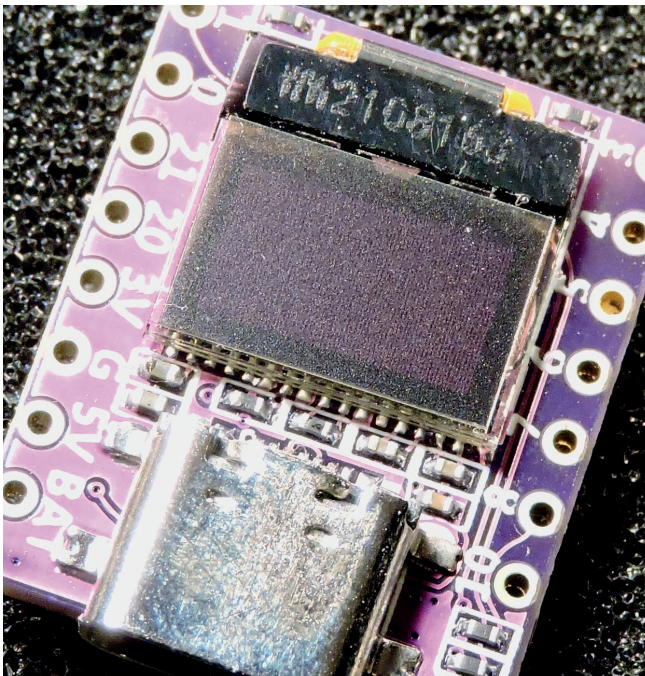


Figure 5.2: The topside of ESP32-C3 with OLED.

5.5. Assembler Listings

It might be an old-school thing, but I like to examine assembler language listings. There was a time when programmers printed assembler listings out and sat at a desk to check them. This practice has long fallen out of favour since computer time is now cheap, and tree huggers protest. Nevertheless, let's explore the assembler language listing for nuggets of useful information to view on a monitor.

5.5.1. ESP32-C3 Assembler Listing

Unfortunately, the way the listing is produced depends upon your environment. So, let's first examine a listing for the ESP32-C3 device using the provided script. Listing 5.4 illustrates how to generate a listing for the `add3.S` program in Linux/MacOS. Windows users will use the batch file `C:\riscv\repo\listesp.bat` instead.

```
$ cd ~/riscv/repo/05/add3
$ ~/riscv/repo/listesp main/add3.S
GAS LISTING /var/folders/jp/_ktnfn412kvbdznr4m9769tr0000gn/T//cc8FupVj.s
    page 1

1          # 1 "main/add3.S"
1          .global add3
0
0
2          .text
3 0000 2E95      add3:   add     a0,a0,a1      # a0 = a0 + a1
4 0002 3295      add     a0,a0,a2      # a0 = a0 + a2
5 0004 8280      ret
                    # return value in a0

GAS LISTING /var/folders/jp/_ktnfn412kvbdznr4m9769tr0000gn/T//cc8FupVj.s
    page 2

DEFINED SYMBOLS
    main/add3.S:3      .text:0000000000000000 add3

NO UNDEFINED SYMBOLS
```

Listing 5.4: Assembler Listing of `add3.S`.

The invoked script merely invokes `gcc` to produce an assembler listing using:

```
gcc -c -Wa,-a,-ad $*
```

The option `-Wa` indicates the further options are to be passed to the assembler, providing assembler options `-a` (high-level listing) and `-ad` (drop debug information from the listing). The `$*` is replaced with the name of the assembler source file.

The source line number is reported at the extreme left of the listing. These are useful references for error and informational messages. Lines starting with zero have no source file line.

The first listing line shows "# 1 "main/add3.S", indicating the source file that was assembled. Opcodes and pseudo-ops are shown in the center of each line. Line 3 shows the first assembled line of code:

```
3 0000 2E95          add3:  add    a0,a0,a1    # a0 = a0 + a1
4 0002 3295          add    a0,a0,a2    # a0 = a0 + a2
```

The four hex digits after the line number indicate the relative address of the assembled instruction. Notice how line 4 shows the relative address 0002, indicating that the add instruction in line 3 was only two bytes in length. Recall that the ESP32-C3 supports RV32IMC, with the "C" indicating support for compressed instructions of 16 bits instead of 32. We'll revisit this idea shortly.

The next four hex digits show the assembled instruction. For example, line 4 shows the instruction as 3295 in hexadecimal. In some cases, as we will see, the instruction length may be longer. There will also be times that the assembler will truncate what is shown there, because of the limits of the line length. In some cases, the assembler will show a temporary instruction code because of relocation performed by the linker.

The assembler listing will also show defined symbols. The following shows that line 3 of main/add3.S defines a symbol named add3, which is located in the .text section of memory. The value of the symbol will be a relative value in the listing since this is adjusted at link time by the linker.

```
DEFINED SYMBOLS
main/add3.S:3      .text:0000000000000000 add3
```

The next section will list any undefined symbols, if there are any. It's a good practice to quickly scan this looking for symbols that should not be undefined.

```
NO UNDEFINED SYMBOLS
```

5.5.2. Influencing Assembly Code

In Listing 5.4, it was shown that the compressed version of the add instructions was assembled because they were only 16 bits in length. The ESP32-C3 device does support RV32IMC base and extensions. The "C" indicates that compressed instructions are supported. What if we didn't want any compressed instructions to be used? The gcc compiler can be told not to:

Listing 5.5 shows the result of adding `-march=rv32im` to the `listesp` script. The option `-march=rv32im` is passed to `gcc`, which informs the assembler to support only the RV32IM base and extension, omitting the "C" extension.

Note: For Fedora Linux, change to `~/riscv/repo/05/add3/qemu64`, and then use the script at `~/riscv/repo/list`.

```
$ cd ~/riscv/repo/05/add3
$ ~/riscv/repo/listesp -march=rv32im main/add3.S
GAS LISTING /var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//cct6UCZb.s
  page 1

1          # 1 "main/add3.S"
1          .global add3
0
0
2          .text
3 0000 3305B500    add3:   add     a0,a0,a1      # a0 = a0 + a1
4 0004 3305C500    add     a0,a0,a2      # a0 = a0 + a2
5 0008 67800000    ret                                # return value in a0

GAS LISTING /var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//cct6UCZb.s
  page 2

DEFINED SYMBOLS
    main/add3.S:3      .text:0000000000000000 add3

NO UNDEFINED SYMBOLS
```

Listing 5.5: Assembler Listing of `add3.S`, with `-march=rv32im`.

The following assembly has changed in that listing:

```
3 0000    add3:      add     a0,a0,a1      # a0 = a0 + a1
4 0004 3305C500    add     a0,a0,a2      # a0 = a0 + a2
```

Here we see that both `add` instructions are now 4 bytes in length. The address of the second `add` instruction is now `0004`, rather than `0002`, since the first instruction is now `3305B500` in hex. So, no compressed instructions were generated. Even the `ret` instruction (return) is now 4 bytes in length.

Note: To perform the same listing in Windows, type:

```
\riscv\repo\listesp.bat "-march=rv32im" main/add3.S
```

Be sure to place the `-march=rv32im` option in quotes for the batch file processor.

5.4.3. Objdump

Sometimes as a developer, you want to disassemble the contents of a compiled or assembled object file. Other times, you may not have the source file for the object. The `objdump` takes many options, but in the ESP32-C3 environment, we can dump out the generated `add3.o` (this file was generated as part of producing the listing file) as follows:

```
$ riscv32-esp-elf-objdump -d add3.o

add3.o:      file format elf32-littleriscv

Disassembly of section .text:

00000000 <add3>:
   0: 00b50533          add    a0,a0,a1
   4: 00c50533          add    a0,a0,a2
   8: 00008067          ret
```

Here we see that the object file is reverse engineered by the `objdump` command, in a format similar to the listing file (this one used `-march=rv32im`). We are reminded by the output, that the format is little endian by the line:

```
add3.o:      file format elf32-littleriscv
```

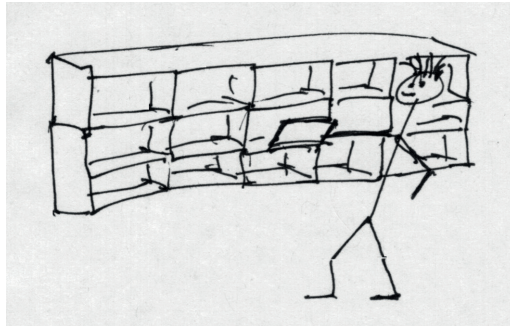
This is an important thing to remember since RISC-V machines are byte-addressable, and the word format is little endian. The `objdump` command simply lists the bytes `00b50533`, as the bytes increase in address. The assembler listing displays the *word* value `3305B500` instead. But with little endian addressing, byte `00` would be the lowest addressed byte, and byte `33` would be placed at the highest of the 4-byte sequence.

Note: For Fedora Linux, use `objdump` as the command name. In Windows you would use the `riscv32-esp-elf-objdump` command.

5.6. Summary

You've succeeded in assembling RISC-V code for the RV64 and RV32 platforms. You've linked your assembly code with a C main program and run it successfully. Knowing the organization of the assembly language source file and the optional listing report puts power into your hands. Finally, you've experienced the first instance of how the C program calls your assembly language code. In the next chapter, we'll examine the instructions needed to load from and store to memory.

Chapter 6 • Load and Store Memory



A pigeon hole is like a memory cell.

In the last chapter, we applied the "add" instruction to sum two registers and place the result into a third register. This is a register-to-register operation. But how do we get values into a register and how do we save them back to memory? This chapter examines the RISC-V operations used for loading and storing.

6.1. A Word About Word Sizes

The smallest addressable memory unit in RISC-V is the 8-bit byte. Integers, whether short, regular, or long are then multiples of bytes. Here's a list of the different word sizes:

- byte (1 byte, 8-bits, smallest addressable memory unit)
- half-word (2 bytes, 16-bits)
- word (4 bytes, 32-bits)
- double word (8 bytes, 64-bits)

Because RV32 is a 32-bit (XLEN=32) platform with 32-bit registers, the ESP32-C3 does not have instructions for loading or storing double words. The RV64 ISA (XLEN=64) does, however.

6.2. Load Instructions

A value is loaded into memory with the "load" instruction. But when we ask for a value to be loaded into a register, we need to specify the following:

- Destination register name (rd)
- The address (usually by assembler symbol)
- The word (unit) size

From this we can list the following basic assembler instructions for loading values:

```
lb    rd, symbol    # Load signed byte
lh    rd, symbol    # Load signed half word
lw    rd, symbol    # Load signed word
ld    rd, symbol    # Load signed double word: RV64I only
```

6.3. Load Program Example

Now let's demonstrate this. Change to the directory containing the 06/loads example:

```
$ cd ~/riscv/repo/06/loads
```

Windows users, use:

```
C:> cd \riscv\repo\06\loads
```

The example is illustrated in Listing 6.1, defining four assembler routines: `loadb()`, `loadh()`, `loadw()` and `loadd()`. This source file demonstrates how to provide multiple functions in one assembly file. Each of the symbols `loadb`, `loadh`, `loadw` and `loadd` can be thought of as multiple "entry points".

```

1      .global      loadb,loadh,loadw,loadd
2
3      .text
4 loadb:  lb      a0,byte      # Load a byte
5      ret
6 loadh:  lh      a0,hword    # Load half word
7      ret
8 loadw:  lw      a0,word     # Load a word
9      ret
10 loadd:  lw      a0,dword    # Load lower word of dword
11      lw      a1,dword+4    # Load upper word of dword
12      ret                # return value in a0
13
14      .data
15 byte:  .byte  1
16 hword: .half  0xF509
17 word:  .word  0x0708090A
18 dword: .dword 0xCCBAA99887766

```

Listing 6.1: The ~/riscv/repo/06/loads/main/loads.S source program.

The functions are defined in C language terms in Listing 6.2. There we see that function `loadb()`, `loadh()` and `loadw()` all return a 32-bit integer value. The last function, `loadd()` will return a 64-bit long long integer type, even on the RV32 platform.

```

#include <stdio.h>

extern int loadb(), loadh(), loadw();
extern long long loadd();

void
app_main(void) {

```

```

    printf("loadb() = %08X\n", loadb());
    printf("loadh() = %08X\n", loadh());
    printf("loadw() = %08X\n", loadw());
    printf("loadd() = %016lX\n", loadd());
}

```

Listing 6.2: Program ~/riscv/repo/06/loads/main/main.c.

The RISC-V 32-bit integer is always returned in register a0. So line 4 of Listing 6.1 loads a byte into the 32-bit register a0 and returns to the caller in line 5. The byte value loaded is 1 (defined in line 15). The half-word value is defined as the value 0xF509 in line 16. Think about what you expect that return value will be in the C program (and hold that thought). Finally, the entry point loadw loads the 32-bit integer value defined in line 17.

Notice that in the entry point loadd we had to perform two-word loads. This is because the registers are only 32 bits in size for the ESP32-C3, so that the upper half of the double word has to be returned in register a1 instead for XLEN=32 platforms.

Build, flash and execute this program on the ESP32-C3. Specify your device port (underlined> after the -p option according to the device port that it appears on (or Windows COM port).

```

$ idf.py build
...
$ idf.py -p <<<your-port>>> flash monitor
...
--- idf_monitor on /dev/cu.usbserial-146410 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
...
I (256) cpu_start: Starting scheduler.
loadb() = 00000001
loadh() = FFFFF509
loadw() = 0708090A
loadd() = 00CCBBAA99887766
(Type Control-] to exit)

```

What did you observe by running the program? The loadb() function reported as 00000001 as expected. So we know that the byte loaded into the 32-bit register as expected. The loadh() function however reported FFFFF509 rather than 0000F509 as you might have expected. If this surprises you, recall that the values are sign extended to the width of the register. Since the high order bit of the half-word was a 1-bit (line 16), that sign bit was extended to the full width of the register. Figure 6.1 illustrates the sign extension process. Finally function loadw() loaded and returned a 32-bit value just fine.

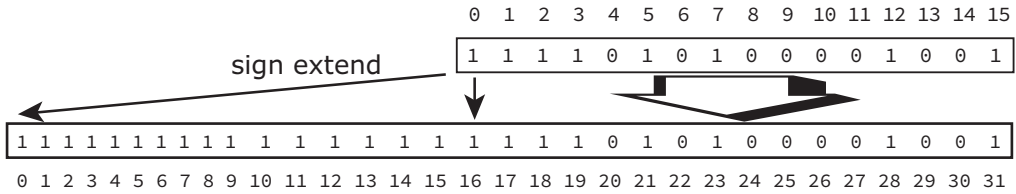


Figure 6.1: Sign extension of a half-word to a 32-bit register.

The `loadd()` function was designed to return a long long integer which is 64-bits in length. Since the registers are only 32-bits in size, the value was returned in the `a0` and `a1` register pair. The low order word in `a0`, and the high order word in `a1`.

Let's now examine the assembler listing for the program you just ran:

```
$ ~/riscv/repo/listesp main/loads.S
```

Windows users use:

```
C:> \riscv\repo\listesp.bat main/load.S
```

```

1          # 1 "main/loads.S"
1          .global loadb,loadh,loadw,loadd
0
0
2
3          .text
4 0000 17050000    loadb: lb      a0,byte      # Load a byte
4          03050500
5 0008 8280      ret
6 000a 17050000    loadh: lh      a0,hword     # Load half word
6          03150500
7 0012 8280      ret
8 0014 17050000    loadw: lw      a0,word      # Load a word
8          03250500
9 001c 8280      ret
10 001e 17050000   loadd: lw      a0,dword     # Load lower word of dword
10         03250500
11 0026 97050000   lw          a1,dword+4     # Load upper word of dword
11         83A50500
12 002e 8280      ret                      # return value in a0
13
14         .data
15 0000 01        byte: .byte 1

```

```

16 0001 09F5      hword:  .half  0xF509
17 0003 0A090807  word:    .word  0x0708090A
18 0007 66778899  dword:   .dword 0xCCBAA99887766
18      AABBC00

DEFINED SYMBOLS
main/loads.S:4   .text:0000000000000000 loadb
main/loads.S:6   .text:000000000000000a loadh
main/loads.S:8   .text:0000000000000014 loadw
main/loads.S:10  .text:000000000000001e loadd
main/loads.S:15  .data:0000000000000000 byte
main/loads.S:16  .data:0000000000000001 hword
main/loads.S:17  .data:0000000000000003 word
main/loads.S:18  .data:0000000000000007 dword
main/loads.S:4   .text:0000000000000000 .L0
main/loads.S:6   .text:000000000000000a .L0
main/loads.S:8   .text:0000000000000014 .L0
main/loads.S:10  .text:000000000000001e .L0
main/loads.S:11  .text:0000000000000026 .L0

NO UNDEFINED SYMBOLS

```

Listing 6.3: Listing for ~/riscv/repo/06/main/loads.S.

Notice the addresses shown left of the `.byte`, `.half` etc. data definitions. They increase starting from zero because they assemble in their own `.data` section (not `.text`, which is normally reserved for code). The defined symbols reported at the bottom also reflect this:

```

main/loads.S:15  .data:0000000000000000 byte
main/loads.S:16  .data:0000000000000001 hword
main/loads.S:17  .data:0000000000000003 word
main/loads.S:18  .data:0000000000000007 dword

```

Notice how the assembler has indicated that these are defined in the `.data` section.

6.4. The `.data` Section

I glossed over the `.data` pseudo-op earlier. As you've probably guessed, this places the data values in a different memory section belonging to data. Placing our data values in `.data`, was *almost* equivalent to declaring the following in C/C++:

```

static char byte = 1;
static short hword = 0xF509;
static int word = 0x0708090A;
static long long dword = 0xCCBAA99887766;

```


All of these values would wind up in SRAM on the ESP32-C3. Under Fedora Linux (QEMU), they are collected into `.data` virtual memory pages, that are read/write capable. The GNU C compiler prefers to gather these into the `.sdata` section, rather than `.data`. But the effect is the same.

We could have defined these particular values in `.text` since these values are never modified. These values would be protected as read-only. In C/C++ terms, defining those values in `.text` is *almost* equivalent to:

```
static char const byte = 1;
static short const hword = 0xF509;
static int const word = 0x0708090A;
static long const dword = 0xCCBAA99887766;
```

The ESP32-C3 device keeps the `.text` values in flash. Fedora Linux (QEMU) places `.text` into virtual memory pages that permit only read and execute permissions. The C/C++ statements shown above *actually* get placed into the section named `".srodata"`. The name of this section suggests Static Read-Only Data. This is preferred over the section `.text` since it indicates that no execute permission should be provided.

Note: The GNU compiler places C/C++ *static const* values into the section named `.srodata` for RV32I and RV64I. Non-const static C/C++ values likewise go into the `.sdata` section instead.

If you comment out the line containing the `.data` pseudo-op, the values will be defined in the `.text` section instead. Let's try it for fun. After you comment the `.data` line out, repeat the build, flash and run of the program. Does it still work? Now examine the listing for it, shown in Listing 6.4.

```
1          # 1 "main/loads.S"
1          .global loadb,loadh,loadw,loadd
0
0
2
3          .text
4 0000 17050000    loadb: lb      a0,byte      # Load a byte
4          03050500
5 0008 8280      ret
6 000a 17050000    loadh: lh      a0,hword     # Load half word
6          03150500
7 0012 8280      ret
8 0014 17050000    loadw: lw      a0,word      # Load a word
8          03250500
9 001c 8280      ret
10 001e 17050000   loadd: lw      a0,dword     # Load lower word of dword
10          03250500
```

```

11 0026 97050000          lw      a1,dword+4      # Load upper word of dword
11      83A50500
12 002e 8280             ret                          # return value in a0
13
14                          #      .data
15 0030 01               byte:  .byte  1
16 0031 09F5             hword: .half  0xF509
17 0033 0A090807         word:  .word  0x0708090A
18 0037 66778899         dword: .dword 0xCCBBAA99887766
18      AABBC00
18      00

DEFINED SYMBOLS
main/loads.S:4          .text:0000000000000000 loadb
main/loads.S:6          .text:000000000000000a loadh
main/loads.S:8          .text:0000000000000014 loadw
main/loads.S:10         .text:000000000000001e loadd
main/loads.S:15         .text:0000000000000030 byte
main/loads.S:16         .text:0000000000000031 hword
main/loads.S:17         .text:0000000000000033 word
main/loads.S:18         .text:0000000000000037 dword
main/loads.S:4          .text:0000000000000000 .L0
main/loads.S:6          .text:000000000000000a .L0
main/loads.S:8          .text:0000000000000014 .L0
main/loads.S:10         .text:000000000000001e .L0
main/loads.S:11         .text:0000000000000026 .L0

NO UNDEFINED SYMBOLS

```

Listing 6.4: the loads.S listing with the .data pseudo-op commented out.

Now notice the relative addresses of the data values. They all have addresses after your code and in the .text section. The last ret instruction in line 12 has a reported address of 002E, so the byte that follows in .text has an address of 0030.

```

main/loads.S:15         .text:0000000000000030 byte
main/loads.S:16         .text:0000000000000031 hword
main/loads.S:17         .text:0000000000000033 word
main/loads.S:18         .text:0000000000000037 dword

```

Isn't this fun?

6.5. Unsigned Values

You saw how the half-word value of F509 was sign extended to FFFF509 when it was loaded into a 32-bit register. But what if you didn't want the value to be sign extended? To address that need, some unsigned load instructions can be used instead (rd represents the destination register):

```
lbu   rd, symbol      # Load unsigned byte
lhu   rd, symbol      # Load unsigned half word
lwu   rd, symbol      # Load unsigned word: RV64I only
ldu   rd, symbol      # Load unsigned double word: RV128I only
```

Figure 6.2 illustrates the operation of an unsigned load of a half-word into a 32-bit register.

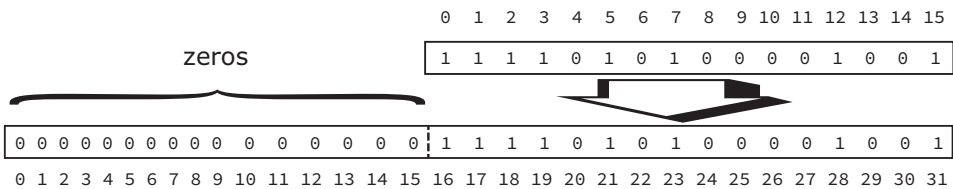


Figure 6.2: the process of loading an unsigned half-word into a 32-bit register.

Notice that RV32I does not have a "lwu" instruction because there is no need for it. The register is only 32-bits in size so there is no sign extension involved. But for RV64I, there is indeed a need because its registers are 64 bits wide. Loading a 32-bit value can be sign extended (lw) or not (lwu) for RV64I.

There is also an RV128I specification where registers are 128-bits wide, which needs the "ldu" instruction but we won't be concerned with RV128 in this book. If you learn RV32I and RV64I, then you will be well prepared for larger architectures.

6.6. Memory Alignment

The RISC-V specification makes a statement about memory alignment:

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity

From this, we can conclude that if performance isn't an issue, misaligned data loads and stores are supported. But if you are concerned about the best possible performance, then your data should be aligned. Further, if you are performing advanced programming where atomic values are needed, then you must align your data to guarantee that the operation is indeed atomic.

To assemble with aligned data, use the `.balign` pseudo-op:

```
.balign n, [fill], [max]
```

where `n` is 1, 2, 4 or 8 (or more). The optional fill parameter specifies what to fill each byte with and defaults to zero. The third optional parameter specifies the maximum number of bytes to pad with when specified. The location is padded with fill bytes until it is aligned to the value of `n`, or the maximum (when given) is reached. For example, to align a 32-bit value, use `n=4` (4-byte entity). To align a half-word, use 2, and for a 64-bit double-word use 8. When using `.balign` in the `.text` section, the fill parameter defaults to the `noop` instruction.

Change to the directory `~/riscv/repo/06/aligned`. This project is otherwise identical to the previous `loads.S` file, except for the added alignment directives in `main/aligned.S`:

```
14      .data
15 byte:  .byte  1
16      .balign 2
17 hword: .half  0xF509
18      .balign 4
19 word:  .word  0x0708090A
20      .balign 8
21 dword: .dword 0xCCBBAA99887766
```

Let's check the assembler listing now. The changes to the listing shown below:

```
14      .data
15 0000 01      byte:  .byte  1
16 0001 00      .balign 2
17 0002 09F5    hword: .half  0xF509
18      .balign 4
19 0004 0A090807 word:  .word  0x0708090A
20      .balign 8
21 0008 66778899 dword: .dword 0xCCBBAA99887766
21      AABCC00
```

From this, it is evident that the alignment directives had their desired effect. For example, the location was 0001 after the declaration of "byte". But after the `".balign 2"` directive, a pad byte was added to bring the location of "hword" to 0002. Similar alignments were unnecessary for the others, since they already had locations suitably aligned.

We can also see the effect in `objdump`, as shown below (after using the `listesp` script to generate a listing and object file):

```
$ riscv32-esp-elf-objdump -sj .data aligned.o

aligned.o:      file format elf32-littleriscv

Contents of section .data:
 0000 010009f5 0a090807 66778899 aabbcc00  ....fw.....
```

Notice that, before the F509 (in little endian format) is defined, the byte value 01 is followed by a 00 byte, to align it on a half word.

6.7. Experiment

Return and edit the program in `~/riscv/repo/06/main/loads.S`, so that instead of the "lh" instruction, it uses the "lhu" instruction instead. The affected loadh code should look like this after the edit:

```
loadh: lhu    a0,hword    # Load half word
      ret
```

Then rebuild, flash, and run the program again. You should get the following results:

```
loadb() = 00000001
loadh() = 0000F509
loadw() = 0708090A
loadd() = 00CCBBAA99887766
```

This time, the loadh() value was reported as 0000F509, without the sign extension. This is the special talent of the "lhu" opcode.

6.8. Immediate Values

Often a small constant is needed, and it is considered tedious and inefficient to have to reach out to memory to fetch it. If instead, that small constant could be embedded inside the instruction word itself, the CPU would already have the data it needed when the instruction is decoded. This concept is known as "immediate data".

Imagine that at some point in your algorithm, all you need to do is to increment the value of register a1 by 1. To do so without immediate data requires that you code something like this:

```
lb    t1,one    # t1 = 1
add   a1,t1,zero # a1 += t1
...

.section .srodata
one:  .byte 1
```

6.9. The li Pseudo-Op

Immediate data allows us to eliminate the need to define the data value of "one". For this, we use the assembler pseudo-op "li" as follows:

```
li    t1,1          # t1 = 1
add   a1,t1,zero    # a1 += t1 + zero
```

The assembler listing might look a little confused for pseudo-ops like "li". For example, assembling just the above code, produces a listing:

```
1          # 1 "t.S"
1          li    t1,1          # t1 = 1
0
0
2 0002 B3050300      add   a1,t1,zero    # a1 += t1
```

But the assembler listing did not report any code for line 1. Code was however, reserved by the evidence of the address of line 2 (the address is 0002, indicating that the prior instruction was two bytes in length). If we ask objdump about what was produced in the object file we get:

```
00000000 <.text>:
0: 4305          li    t1,1
2: 000305b3      add   a1,t1,zero
```

There is positive proof that the first instruction was 2 bytes, followed by a 4-byte instruction. You might be wondering why all the smoke and mirrors for the "li" opcode? It turns out that the assembler has to jump some hurdles for certain constants. For small constants like the number 1, the constant can be embedded into a compressed instruction (the keen student is encouraged to examine the RISC-V instruction formats in [1]). For larger constants, the pseudo-op is expanded by the assembler into multiple instructions as necessary. Consider the following code:

```
li    t1,0xFEEDBEEF # t1 = 0xFEEDBEEF
add   a1,t1,zero    # a1 += t1
```

The assembler in this case expands the "li" pseudo-op into two RISC-V opcodes:

```
00000000 <.text>:
0: feedc337      lui    t1,0xfecd
4: eef30313      addi   t1,t1,-273 # 0xfedbeef
8: 000305b3      add   a1,t1,zero
```

The first is generated as a "lui" opcode, with the upper portion of the constant (0xFEEDC). This new instruction is the "Load Upper Immediate" opcode. This immediate data is placed into the upper 20 bits of the destination register t1 so that the value becomes 0xFEEDC000

(the lower 12 bits are zeroed). The second generated instruction uses an "addi" (add immediate) instruction to add the signed value -273 to t1 (effectively subtracting 273). When that instruction completes, t1 will hold the value 0xFEEDBEEF as intended.

Let's check this in the gdb debugger. Tool gdb is a very useful for developers. It not only debugs but allows you to perform hexadecimal calculations. Try typing the underlined text into gdb as follows and type q to quit:

```
$ gdb
GNU gdb (GDB) 11.2
Copyright (C) 2022 Free Software Foundation, Inc.
...
(gdb) p /x 0xFEEDC000 - 273
$1 = 0xfeedbeef
(gdb) q
```

Note: Windows users must use the command name "riscv32-esp-elf-gdb" instead of "gdb".

Here we use the gdb "p" (print) command, "/x" to print the result in hexadecimal, subtracting 273 (decimal) from FEEDBEEF (in hexadecimal). The result is displayed as 0xfeedbeef. The printed result confirms the computation is indeed correct. This also proves that gdb is indeed your friend. We'll see more of gdb later in this book.

Notice that the "addi" instruction is an add instruction with some "immediate data" capability. Many of the basic RV32I and RV64I instructions have an immediate data version to allow a signed constant to be applied instead of a second source register.

6.10. The addi Opcode

As previously introduced, many instructions of the RV32I/RV64I have an immediate data counterpart. The example that we began with for immediate data can now be boiled down to just one instruction for incrementing a1 by 1:

```
addi    a1,a1,1      # a1 += 1
```

This is far more convenient and efficient than the earlier attempts. What is assembled, however, is going to depend upon the size of the constant used. In some cases, the assembler may report an error for cases that cannot be directly encoded.

6.11. Pseudo-Op mv

Register values sometimes need to be copied or moved from one register to another. The general form of this opcode is this:

```
mv      rd, rs
```

The source register *rs* is copied to the destination register *rd*, where the destination register is not *x0* (zero). When, however, the destination register *is* *x0* (zero), that operation performs as a "noop" (no operation).

The "mv" is a pseudo-op because, with a small trick, it is possible to do this without creating a new instruction. The pseudo-op also permits the assembler to optimize the move request. The following two instructions are equivalent, except for their size:

```
add    t0,t1,x0    # t0 = t1 + x0 (effectively copies t1 to t0)
mv     t0,t1       # better (compressed opcode when RV*C)
```

The effect of moving register *t1* to *t0* can be achieved by adding *t1* to zero (*x0*) and placing the result in *t0*, in the first example. However, if you use the "mv" pseudo-op, the assembler can substitute a compressed opcode (16 bits) in place of a 32-bit opcode, when it is permitted to do so.

6.12. Loads under RV64I

If you're interested in the RV64I experience, start up your QEMU emulator and log in to your Fedora Linux instance. Change to the following directory and compile the loads.S assembler program:

```
$ cd ~/riscv/repo/06/loads/qemu64
$ gcc -O0 -g loads.S main.c
$ ./a.out
loadb() = 00000001
loadh() = FFFFF509
loadw() = 0708090A
loadd() = FFFFFFFF99887766
```

The loads.S program for RV64I is illustrated in Listing 6.4. This program is identical to the ESP32-C3 version except that line 10 uses the *64-bit opcode* "ld" to load the double word into the 64-bit register *a0*, where the value is returned.

```
1      .global loadb,loadh,loadw,loadd
2
3      .text
4 loadb: lb      a0,byte      # Load a byte
5      ret
6 loadh: lh      a0,hword    # Load half word
7      ret
8 loadw: lw      a0,word     # Load a word
9      ret
10 loadd: ld     a0,dword    # Load lower word of dword
11      ret                # return value in a0
12
13      .data
```



```

14 byte:   .byte   1
15 hword:  .half   0xF509
16 word:   .word   0x0708090A
17 dword:  .dword  0xCCBAA99887766

```

Listing 6.4: the RV64I version of the loads.S program.

Run the program to satisfy yourself that it worked.

For testing alignment, perform the following:

```

$ cd ~/riscv/repo/06/aligned/qemu64
$ gcc -O0 -g aligned.S main.c
$ ./a.out

```

In Fedora Linux, produce a listing with the `~/riscv/repo/list` script. Use command name "objdump" to dump an object file.

6.1.3 The .section Pseudo-Op

Before we discuss storing data, let's revisit the `.section` pseudo-op and fully explore the `.section` directive. The attributes of a section will determine whether or not you can store a value in that memory region. The full format for the pseudo-op is:

```
.section name[, "flags"[, @type] ]
```

The name, as we've seen before is a section name like `.data`, `.sdata` or `.srodata` etc. Pre-defined sections with these names have default attributes associated with them. The flags argument, when present, must be enclosed in double-quotes and consist of one or more of the following flag characters:

- "a" – the section is allocatable
- "w" – the section is writable
- "x" – the section is executable

The allocatable attribute (a) means that space can be reserved but is not otherwise initialized. This means that the region is *not necessarily zeroed* or otherwise initialized (some platforms may zero this before calling the main entry point however). The writable attribute (w) is used for data areas that may be updated, like static data variables in a C program. Finally, the execute attribute (x) is used to indicate executable code. Under Linux, this permits the execution of code in the region and the lack of this permission prevents the execution of data.

Once a section has its flags defined (with some very special exceptions), they cannot be changed. It is thus very important that sections be defined consistently.

The @type argument, when present, may be one of the following values:

- @progbits – the section contains data
- @nobits – the section only occupies space and does not contain data

We can create our own section like .srodata, and name it .precious as follows:

```
.section .precious, "a", @progbits
```

This declares a memory section that can be allocated, but lacks write and execute permission. Because of the "@progbits" type, there can be initialized data within it, resulting in a section of read-only data.

Table 6.1 contains the most commonly encountered predefined section names. There are other specialized names used by C++ that are not shown for use in constructors and destructors, for example. You as the application developer are free to create section names of your own.

Section Name	Flags	Type	Usage
.text	"ax"	@progbits	Read-only executable object code
.bss	"wa"	@nobits	Read/Write uninitialized data
.data	"wa"	@progbits	Read/Write initialized data
.sdata	"wa"	@progbits	Read/Write initialized short data
.srodata	"a"	@progbits	Read-only data
.rodata	"a"	@progbits	Read-only data (literals)
.comment	" "	@progbits	Comments embedded in the object file

Table 6.1: Common predefined section names and their attributes.

Notice that the ".bss" section just allocates space for data, but does not define or initialize any data values for it. Some platforms may zero these sections prior to invoking the main function.

6.14. Storing Data

After a calculation or register manipulation is performed, there is normally a need to save that result into memory for later use or reference. This is done with the corresponding sb, sh, sw or sd opcode. The register "rs" below is the source register in this case because the memory is the destination. However, the store address is somewhat complicated, which will be explained shortly. The opcode defines the unit size of the store. For example, the "sb" instruction only updates a single byte in memory. Alignment is also important for performance reasons and is critical for atomic operations. There is no unsigned counterpart to these since sign extension never applies to storing in memory. In the following list, rx refers to a base or index register, while the offset is a fixed offset from the value in rx.

```

sb    rs, offset(rx)    # Store byte
sh    rs, offset(rx)    # Store half-word
sw    rs, offset(rx)    # Store word
sd    rs, offset(rx)    # Store double word: RV64I only

```

Since global addresses can be large, a store is usually performed in reference to a base register and a fixed offset. To help us in this regard, the assembler (and indirectly the linker) provides the use two special functions:

- `%hi(symbol)` – returns the high order 20 bits for absolute symbol
- `%lo(symbol)` – returns the low order 12 bits for the absolute symbol

Consequently, for global memory store operations, the following pattern is often used:

```

lui    t0,%hi(symbol)    # t0 = high order 20 bits of symbol
sw    a0,%lo(symbol)(t0) # address = t0 + low order 20 bits

```

The "lui" operation loads the high order 20 bits of the address symbol into t0 in this example, setting the lower 12 bits to zero. Then the "sw" opcode stores register a0 into the address computed by t0 plus the 12-bit offset returned from the assembler function `%lo(symbol)`.

We're now equipped for an example program. Change to the directory:

```
$ cd ~/riscv/repo/06/celcius
```

Listing 6.5 illustrates the main program that is going to call upon our Fahrenheit to Celsius assembler conversion using integer arithmetic. In this example, we are using global integers to pass and return the values to illustrate the load and store operations in the assembler. The calculations are performed in an integer, so the input value `temp_f10` (line 5) is the temperature in Fahrenheit multiplied by 10. The resulting value `temp_c10` (line 6) is likewise the integer result times ten. The `printf()` statement (lines 13 to 17) reports the values with the necessary format adjustments.

```

1  #include <stdio.h>
2
3  extern void convtemp();
4
5  int temp_f10 = 400;    // Fahrenheit degrees * 10
6  int temp_c10 = 0;     // Computed result: Celsius * 10
7
8  void
9  app_main(void) {
10
11      convtemp();      // Convert temp_f10 to temp_c10
12

```

```

13     printf("%.2d F -> %.2d C\n",
14           temp_f10 / 10,
15           temp_f10 % 10,
16           temp_c10 / 10,
17           temp_c10 % 10);
18 }

```

Listing 6.5: The main program ~/riscv/repo/06/celsius/main/main.c for converting Fahrenheit to Celsius.

The assembler routine is shown in Listing 6.6.

```

1     .global convtemp,temp_f10,temp_c10
2
3     .text
4 convtemp:
5     lw     t0,temp_f10           # t0 = F * 10
6     addi   t0,t0,-320           # t0 = (F * 10) - (32 * 10)
7     li     t1,10                # t1 = 10
8     mul    t0,t0,t1             # t0 *= 10
9     li     t1,18                # t1 = 1.8 * 10
10    div    t0,t0,t1             # t0 = F * 100 / 1.8 * 10
11    lui    t1,%hi(temp_c10)
12    sw     t0,%lo(temp_c10)(t1) # t0 = Celsius * 10
13    ret

```

Listing 6.6: Assembly language routine ~/riscv/repo/06/celsius/main/celsius.S.

This function does not return a value, so our calculation is performed in temporary register t0. Line 5 loads the global int value temp_f10. The signed immediate value -320 is added to t0 before it is multiplied by 10 in lines 7 and 8. We've not looked at "mul" and "div" yet, but they take two source operands and produce a result in the destination register. These are both signed integer computations.

Once the temp result in t0 is multiplied by 10, the value is divided by 18 (1.8 times 10), to produce a result in t0 (line 10). This will be in degrees Celsius times 10. Then temporary t1 is loaded with the high order 20 bits of the global integer temp_c10 (line 11). Finally, in line 12, we can store the register t0 into the global variable using the offset of %lo(temp_c10) added to t1. After the routine returns in line 13, the C program reports the answer:

```
40.0 F -> 4.4 C
```

Now let's re-examine the "lw" opcode in line 5 of Listing 6.6. The objdump utility will report something like the following:

```

0: 00000297          auipc   t0,0x0
4: 0002a283          lw      t0,0(t0) # 0 <convtemp>

```

The value is not known until the link step is performed, so the immediate constant shown is simple 0x0, which is fixed up later by the linker. By dumping out build/celsius.elf for the ESP32-C3 device, we see that the instruction gets converted to the following instruction:

```

420051d2:      8341a283      lw      t0,-1996(gp) # 3fc89834 <temp_f10>
420051d6:      ec028293      addi   t0,t0,-320

```

In other words, the ESP32-C3 linker script has established a global pointer in register `gp`, and the word is loaded using an offset of -1996 from it. If your brain is feeling a little bit of hurt right now, don't despair. We can use a more friendly bit of code to do the same thing. Listing 6.7 contains the source code for `celsius2.S`.

```

1      .global convtemp,temp_f10,temp_c10
2
3      .text
4 convtemp:
5      lui    t0,%hi(temp_f10)
6      lw     t0,%lo(temp_f10)(t0)    # t0 = F * 10
7      addi   t0,t0,-320              # t0 = (F * 10) - (32 * 10)
8      li     t1,10                   # t1 = 10
9      mul    t0,t0,t1                # t0 *= 10
10     li     t1,18                   # t1 = 1.8 * 10
11     div    t0,t0,t1                # t0 = F * 100 / 1.8 * 10
12     lui    t1,%hi(temp_c10)
13     sw     t0,%lo(temp_c10)(t1)    # t0 = Celsius * 10
14     ret

```

Listing 6.7. program ~/riscv/repo/06/celsius2/main/celsius2.S.

The cryptic load is now replaced with the more familiar "lui" and "lw" instructions in lines 5 and 6. Line 5 loads the high order 20 bits of the absolute address for global `temp_f10` into `t0`. Then the low order 12 bits are added to that to form an address of the word to be loaded. That word replaces `t0` with the contents of the memory word that we were after. If you build and flash that project, then running it on your ESP32-C3 will confirm that this still works.

RV64I Run

Start up your Fedora Linux instance in QEMU, and perform the following:

```

$ cd ~/riscv/repo/06/celsius2/qemu64
$ gcc -O0 -g celsius2.S main.c
$ ./a.out
40.0 F -> 4.4 C

```

Your result should match what you got for the ESP32-C3 run.

6.15. Review

We've covered a lot of ground in this chapter, wading into the various options for loading from and storing to global memory, immediate values, moving register values, memory alignment and section attributes. Let's summarize this and review the instructions that you've learned in this chapter.

To load values from memory, you can use the following opcodes using a memory offset and base register *rx*, to load into the destination register *rd*:

```
lb    rd, offset(rx)    # Load signed byte
lbu   rd, offset(rx)    # Load unsigned byte
lh    rd, offset(rx)    # Load signed half-word
lhu   rd, offset(rx)    # Load unsigned half-word
lw    rd, offset(rx)    # Load signed word
lwu   rd, offset(rx)    # Load unsigned word: RV64I
ld    rd, offset(rx)    # Load signed double word: RV64I
ldu   rd, offset(rx)    # Load unsigned double word: RV128I only
```

Likewise, the following opcodes may be used to store register *rs* into memory using a memory offset and base register *rx*:

```
sb    rs, offset(rx)    # Store byte
sh    rs, offset(rx)    # Store half-word
sw    rs, offset(rx)    # Store word
sd    rs, offset(rx)    # Store double word: RV64I only
```

The assembler also supports the load immediate form, for loading constants:

```
li    rd, immediate
```

Register contents can be copied to another register using the *mv* pseudo-opcode:

```
mv    rd, rs
```

Additionally, several instructions like *add* also have an immediate form:

```
addi  rd, rs, immediate
```

We also saw the use of the special "*lui*" and "*auipc*" instructions:

```
lui   rd, immediate    # Loads 20-bits of immediate data to upper rd
auipc rd, immediate    # Loads rd with the current PC + signed immediate 12-bits
```

Note that some of these instructions like "li" and "mv" for example, are adjusted according to the assembler based upon the size of the address/immediate data involved. Finally, we briefly saw and used the multiply and divide instructions, which will be explored later in the book.

It might at first appear that the load and store opcodes are rather clumsy because of the need to prepare a base register and then use an offset from that. This is true perhaps for global memory locations. Most memory accesses today, however, are relative to a stack frame, for which an offset and a stack frame pointer are ideal. We'll explore stacks and stack frames later in this book.

6.16. RISC-V Assembler Modifiers

In connection with memory accesses, the need frequently occurs for the evaluation of the 20-bit upper portion of a constant or address and the lower 12-bits of the same. We've already seen these two assembler functions used in this chapter:

- `%hi(symbol)` – provides high order 20 bits of the symbol
- `%lo(symbol)` – provides the low order 12 bits of the symbol

There are two more useful functions for the programmer. They are:

- `%pcrel_hi(symbol)` – the high 20 bits of relative address between pc and *symbol*
- `%pcrel_lo(symbol)` – the low 12 bits of relative address between pc and *symbol*

These functions are used to provide a relative address constant. When we examine branches, we will see more relative addresses from the current instruction counter (PC).

6.17. Summary

This chapter has introduced you to the critical operations of loading from and storing to memory. It was shown that immediate data can provide for more optimal code. Mastering these approaches is an essential start to successful RISC-V assembly language programming. The next chapter will take a detailed look at the essentials of the GNU calling convention.

Bibliography

- [1] *The RISC-V instruction set manual volume I: User-level Isa* (n.d.). Retrieved April 23, 2022, from <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>, 2.2 Base Instruction Formats

Chapter 7 • Calling Convention



Standing is a calling convention

In order for compilers, assemblers and linkers to produce a final working executable, an agreement must exist between them on the use of the stack and the registers used. Since we're using the GNU compiler collection in this book, let's discuss the GNU calling convention.

7.1. Register Usage

One of the first considerations of the convention is the use of available registers and who's responsible for saving and restoring them. Registers are limited in number and sometimes have architectural limitations. RISC-V restricts us to using registers x1 to x31, since x0 has a hardwired zero or discard talent. The remaining registers are fully general purpose. Table 7.1 lists the registers that are available, their intended function and who saves them when necessary.

Register	ABI Name	Description	Saver
x1	ra	Return Address	Caller
x2	sp	Stack Pointer	Callee
x3	gp	Global Pointer	-
x4	tp	Thread Pointer	-
x5-x7	t0-t2	Temporary Registers	Caller
x8	s0/fp	Saved register / Frame Pointer	Callee
x9	s1	Saved Register	Callee
x10-x11	a0-a1	Function arguments / return value	Caller
x12-x17	a2-a7	Function arguments (continued)	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

Table 7.1: Registers for the GNU calling convention.

In chapter 4, Architecture, the function of these registers has already been described. In this chapter, we'll focus on the call and return.

7.2. Call Procedure

In order for a subroutine to return to the caller where it left off, a return address must be provided as part of the call. In RISC-V this is performed by the Jump and Link "jal" or the Jump and Link Register "jalr" instructions:

```
jal    rd,offset    # rd = PC + n, PC = PC + n + offset
jalr   rd,offset(rx) # rd = PC + n, PC = rx + offset
```

The value of n is 2 when the instruction is compressed as a half-word, or else it is 4 for a word. The value of $PC + n$ is the address of the instruction following the current one.

7.2.1. Opcode jal

The "jal" instruction permits the programmer to code a "call" to a subroutine. It performs two things when it executes:

1. Copies the current $PC + n$, which is the address following the current instruction, to the destination register rd .
2. Adds the relative offset to the address $PC + n$ to begin execution at the called location.

Normally the destination register is ra ($x1$), but any other register can be used, including $x0$. When $x0$ is specified, the return address is discarded, and the operation becomes a jump instead. When rd is not $x0$, the return address is saved there, and execution resumes at the call address.

```
jal    rd,symbol
```

The destination register rd can be omitted, if you're using the standard ra ($x1$) register for this call:

```
jal    symbol
```

In both cases, the symbol is converted by the assembler into a half-word offset from the current instruction address. This offset is used to jump to the new routine address as part of the call.

7.2.2. Pseudo Opcode jr

In order to return from the call, the "jr" pseudo-opcode can be used to return to the address found in the specified register.

```
jr    rs
```

The operation of this instruction sets the PC to the value in register `rs`. In effect, it is a jump through register operation. This causes execution to resume after the point of the call. Why is "jr" a pseudo-instruction? The assembler expands this to the instruction:

```
jalr    x0,rs,0
```

Effectively this instruction sets the PC to the value of `rs + 0` and, because no return value is saved, it simply becomes a jump.

7.2.3. Pseudo Opcode `ret`

We've seen the "ret" opcode before in the previous chapters. It was simply coded as:

```
ret
```

But how does that differ from the operations we've just reviewed? The "ret" is a pseudo-opcode that is expanded by the assembler into:

```
jalr    x0,ra,0
```

By the GNU calling convention, the return address is saved in register `ra` (`x1`). This allows us to return to the caller by jumping to the address in `ra` (`x1`).

7.2.4. General Call Procedure

Let's now review the general call procedure:

1. The caller performs a "jal" or "jalr", causing the return address to be placed in `rd`, which is normally `ra` (`x1`).
2. The PC jumps to the offset + `rs` to start executing the called code.
3. The called code, returns by "ret", or "jr `ra`", or "jalr `x0,ra,0`".

Unless you're using a different register than `ra` (`x1`), the "ret" pseudo-opcode is recommended since this is easily understood and clear. We'll see later that there are sometimes reasons to use a different register.

7.2.5. Call to 32-bit Absolute Address

When using the "jal" opcode, the half-word offset is computed by the assembler to be relative to the current instruction. But when the target symbol is too far away for a relative branch, another approach must be used. The "call" pseudo-op comes to the rescue for calling, for example, a function named `foo`:

```
call    rd, foo      # specified rd register, or...
call    foo          # ra (x1) is assumed
```

This pseudo-op is expanded into a pair of instructions "auipc" and "jalr" to accomplish the far call. The expansion of the normal case of "call `foo`", would be:

```

1:    auipc  ra,%pcrel_hi(foo)
      jalr   ra,ra,%pcrel_lo(1b)

```

Recall that the assembler function `%pcrel_hi(symbol)` returns the high order 20 bits of the symbol and `%pcrel_low(symbol)` returns the lower 12 bits. The "auipc" step loads the high order 20 bits of the foo address into ra (x1) temporarily, which is the word offset from the current address in PC. The "jalr" instruction then branches to the temporary value in x1 *plus* the low order 12 bits of foo's address provided by `%pcrel_lo(1b)`. This arrives at the PC for the far away subroutine. Upon completion of the "jalr" opcode, the return address (of PC) is saved in ra (x1).

Numeric Labels

Before we can fully explain the "`%pcrel_lo(1b)`" part of this code, we need to explain the numeric label of "1" and the reference to it as "1b". GNU assembler permits numeric labels like "1" and references to these are either "1b" or "1f", indicating the first "1" back, or the first "1" forward. This clever system permits labels to be reused without having to invent unique names. This is especially useful in assembler macros. These numeric labels never register as external symbols.

Returning to the "auipc" instruction, we can see that it loads the high order 20 bits of foo's address into register ra (x1), with the PC added to it. The x1 register is used temporarily for this. The "jalr" instruction which follows then adds the low order 12 bits of foo's address to ra (x1) to compute the target subroutine address. The assembler function `%pcrel_lo(1b)` must reference the address of the "auipc" instruction where `%pcrel_hi(foo)` was used. Otherwise, there is a chance that the calculation might be incorrect.

This might seem like a lot to digest right now. The important concept here is to understand that a "call" is expanded into a pair of instructions to make a relative far call possible, when necessary.

7.2.6. Revised Call Procedure

With the vulgarities of near and far symbols out of the way, the procedure call can now be summarized as:

1. The caller performs a "call", causing the return address to be placed in rd, which is normally ra (x1).
2. The PC jumps to the subroutine address to start executing the called code.
3. The called code, eventually returns by "ret" (or "jr rs", when a different register is used).

7.2.7. Concrete Call Example

Sometimes the technical details are mind-numbing in the abstract. So let's review a long call with actual addresses to visualize how this works. In this example, we're going to call function `foo()`, which is assumed to be far away from the caller's current PC. In the following, the PC for the "auipc" instruction is `0x42005e16` (this information comes from an actual debug session):

```

#                call foo                # located at 0x427F0000

42005e16 007ea097  auipc  ra,0x7ea  # ra = 0x42005e16 + 0x7ea000
42005e1a 1ea080e7  jalr   490(ra)   # ra = 0x42005e1a + 4, PC = 0x427f0000

```

Now verify this calculation using gdb:

```

(gdb) p /x 0x42005e16 + 0x7ea000 + 490 # Remember 0x7ea is shifted up 12 bits
$2 = 0x427f0000

```

The current address of the "auipc" instruction 0x42005e16 is added to the constant 0x7EA000 (which is derived from 0x7EA placed in the upper 20 bits) and the result is temporarily stored into ra (x1). Then the "jalr" instruction adds the constant 490 (decimal) to ra (x1), to arrive at 0x427F0000 for foo(). As part of the "jalr" instruction, the return address of 42005E1A + 4 is now stored into ra (x1), while the PC register is set to foo's entry point address of 0x427F0000.

The good news is that the programmer doesn't need to worry about this much. The assembler and the linker do all the dirty work to make a long or short call as necessary.

7.2.8. Simple Call Experiment

Before we dig deeper into the calling convention, let's just prove to ourselves that the call and return mechanism works as advertised. The assembler routine in Listing 7.1 illustrates a simple function named "callme" for ESP32-C3 that performs the following:

1. Loads the value of 1 into the register a0 (x10) in line 4.
2. Calls an internal subroutine named "intern" from line 5, using temporary register t0 (x5).
3. Adds the value 2 to the value in a0 in line 8.
4. Returns to the caller from line 9, via register t0 (x5), to line 6.
5. Control returns to app_main() from line 6, using register ra (x1).

Part of the calling convention is that the return address is saved into register ra (x1). If, however, you need to call some mini-routine(s), the convention is that t0 (x5) is used. This avoids having to save and restore ra (x1) from memory.

```

1      .global callme
2
3      .text
4 callme: li    a0,1          # a0 = 1
5        call  t0,intern     # Call internal
6        ret
7
8 intern: addi  a0,a0,2      # a0 += 2
9        jr   t0            # Return to retn

```

Listing 7.1: An inner call example, `~/riscv/repo/07/jal/main/jal.S`.

Listing 7.2 illustrates the ESP32-C3 main program used for this experiment. When the function `callme()` is called the register `a0` is loaded with the value of 1 (line 4). After the internal function `intern()` is called (at line 8) the value of 2 is added resulting in `a0` holding the value of 3.

```

1 #include <stdio.h>
2
3 extern int callme();
4
5 void
6 app_main(void) {
7
8     printf("callme() returned %d\n",callme());
9 }
```

Listing 7.2: Main program for jal.S ,~/riscv/repo/07/jal/main/main.c.

When you build and flash this code, the run output should report the following (substitute the appropriate path for your USB device):

```

$ idf.py build
...
$ idf.py idf.py -p <<<your-port>>> flash monitor
...
callme() returned 3
```

This is the expected return value (1 + 2).

The same program exists for QEMU. You can build and run it as follows:

```

$ cd ~/riscv/repo/07/jal/qemu64
$ gcc -O0 -g jal.S main.c
$ ./a.out
callme() returned 3
```

7.2.9. Running in gdb

To get introduced to `gdb`, let's walk through the program in QEMU. Start the `gdb` session as follows:

```

$ gdb ./a.out
GNU gdb (GDB) Fedora 9.0.50.20191119-2.0.riscv64.fc32
Copyright (C) 2019 Free Software Foundation, Inc.
...
Reading symbols from ./a.out...
(gdb)
```

This prepares to debug the program executable `a.out` and pauses for your next command. As long as you compiled the code with the `-g` option, you should be able to list your code with the `"list"` command:

```
(gdb) list
1      #include <stdio.h>
2
3      extern int callme();
4
5      int
6      main(int argc,char **argv) {
7
8          printf("callme() returned %d\n",callme());
9          return 0;
10     }
(gdb)
```

Now let's set a "breakpoint" for the function `callme()`:

```
(gdb) b callme
Breakpoint 1 at 0x1048e: file jal.S, line 4.
(gdb)
```

Now when the program is running, it will stop when `callme()` is invoked. Since the program is not running yet, let's start it:

```
(gdb) r
Starting program: /home/riscv/riscv/repo/07/jal/qemu64/a.out
glibc-2.30.9000-29.fc32.riscv64
Breakpoint 1, callme () at jal.S:4
4      callme: li      a0,1          # a0 = 1
(gdb)
```

Because we have the breakpoint set, the execution paused as soon as it entered the assembler routine `callme()`. So far, so good. Now we can trace one instruction at a time. Let's "step" one instruction:

```
(gdb) s
5          call    t0,intern      # Call internal
(gdb)
```

We see the execution has performed the assembler statement of line 4. But let's examine the register contents of `a0`:

```
(gdb) info reg a0
a0          0x1      1
(gdb)
```

If we leave the register name out, all registers would be reported. We see that the register a0 has been loaded with the value 1, as coded. Let's step one more instruction:

```
(gdb) s
intern () at jal.S:8
8      intern: addi    a0,a0,2      # a0 += 2
(gdb)
```

Now we've called the intern() function by use of t0. Step once again and display registers:

```
(gdb) s
9      jr      t0          # Return to retn
(gdb) info reg t0 a0
t0          0x10494 66708
a0          0x3       3
(gdb)
```

The register a0 now has the value 3, and the register t0 has the return address from the call in line 5. Let's step from line 9:

```
(gdb) s
callme () at jal.S:6
6      ret
(gdb)
```

From this, we see the control has returned to after the point of the call in line 6. Stepping once again should get us back to the main() program, which called us:

```
(gdb) s
callme() returned 3
main (argc=1, argv=0x3fffffff2a8) at main.c:9
9      return 0;
(gdb)
```

From this session, we see that the printf() call was also completed as part of the return, so that gdb could return a complete statement executed, returning to the statement following in line 9. Since we don't care about tracing the rest of the program you can just "continue" it as follows:

```
(gdb) c
Continuing.
[Inferior 1 (process 843) exited normally]
(gdb) q
```

The program then exits in the normal way. You could just "quit" at that point also.

What did we learn? Using the gdb debugger (under QEMU), we were able to trace the code from `main()`, to `callme()`, and step through each assembler instruction until we returned to the main program. Along the way we, were able to report register contents. This is debugging in luxury!

7.3. Argument Passing in Registers

We've already seen that arguments are passed in registers starting with `a0` (with exceptions for *hardware* floating point). And that return values are returned starting with `a0`. Arguments and return values less than or equal to `XLEN` bits in size are sign-extended into the register for signed types. Otherwise, the register is zero-filled in the high order bits for unsigned types.

When a value like a 64-bit integer must be passed or returned on an `XLEN=32`-bit platform, the low-order `XLEN` bits are loaded into `a0` (or even numbered argument register). The high-order bits are then passed into the next odd-numbered register depending. Figure 7.1 illustrates how a long long int is passed or returned for the ESP32-C3.

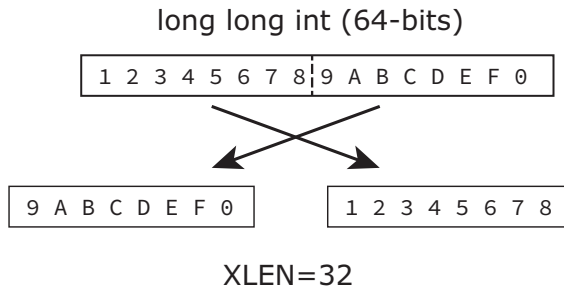


Figure 7.1: How a long long int is loaded into two 32-bit registers on ESP32-C3.

When an `int` followed by a long long is passed, the `int` goes into register `a0` (the `int` fits the register). However, the long long argument has its low order word in `a2` (the even-numbered register), and the high order word is passed in `a3`. Very large arguments (more than twice the size of a pointer) are passed by reference!

This convention is used for all arguments until all eight of the argument registers are used up. When registers `a0` through `a7` are allocated, what happens to the remaining arguments? These are placed on the stack. After the prologue executes, the stack pointer `sp` (`x2`) points to the first argument that was not passed in a register.

7.4. The Stack

I've avoided discussing the stack until this point. It is most efficient to pass and return values by registers alone when that is possible. However, when arguments that exceed registers or local variables are necessary, then the stack is needed.

The stack pointer value is found in register `sp` (`x2`) and is set to the high address of the stack and grows downward. The stack pointer is maintained at an alignment boundary, which for ESP32-C3 (where `XLEN=32`) is a double word address (modulo 8 bytes). For `XLEN=64` platforms (QEMU), this is modulo 16 bytes.

7.4.1. Prologue

At the start of the function, the prologue serves to perform any functions necessary to preserve the integrity of the call. This involves adjusting the stack pointer and saving to the stack as needed. The general procedure is:

1. Decrement `sp` by the stack frame size for register saves, including any local variable space (the size is round up to mod 8 for `XLEN=32`, or mod 16 for `XLEN=64`).
2. Store registers to be saved in the allocated stack frame (this optionally includes the `s0/fp` (`x8`) register).
3. Save register `ra` (`x1`) when the called routine calls or reuses register `ra`.

The `fp/s0` (`x8`) register is often setup by the C compiler so that negative offsets refer to values saved on the stack. By doing that, the stack can grow or shrink without losing track of local variable addresses or saved register values. For example:

```

4      addi    sp,sp,-32    # Allocate 32 bytes of stack space
5      sd     s0,24(sp)    # Save s0/fp at 24(sp)
6      addi    s0,sp,32    # Set s0/fp to original sp value

```

Then, to save one byte to a local stack byte variable, it might use:

```

8      sb     a5,-17(s0)   # Store byte at -17(s0)

```

Note: It is important to adjust the `sp` (`x2`) as the first step. This keeps your stack frame from being corrupted by intervening interrupts should they occur.

When the function is first called, the `sp` (`x2`) points to the first overflow argument (the calling program arranges this). After the function prologue completes, register `s0/fp` (`x8`) points to that first overflow argument instead. Positive offsets from `s0/fp` point to excess calling arguments. Since the caller arranges these arguments, the called function never needs to worry about releasing them.

Reviewing the prologue steps:

1. Allocate space on the stack by subtracting from the current `sp` (x2) and maintain stack alignment (this keeps the stack frame interrupt safe).
2. Optionally save the current `s0/fp` (x8) and other registers as necessary.
3. Optionally save register `ra` (x1), when necessary.
4. Optionally setup stack frame register `s0/fp` (x8) when required. This is normally required when all of the arguments did not fit into the available registers.

While the C compiler might use negative offsets from register `fp/s0` to access local variables, it does not have to be done that way. The same access to local variables can be had with positive offsets from the stack pointer register `sp`.

7.4.2. Epilogue

When it is time to return to the caller, the stack changes must be undone, and the necessary registers restored. This procedure consists of:

1. Reload saved registers (including optional `s0/fp`).
2. Reload `ra` (if necessary).
3. Increment `sp` by the amount it was subtracted in the prologue.
4. Return to the caller's address in register `ra`.

Note that the called function does not concern itself with freeing excess arguments that were passed on the stack. The calling program takes care of that for us.

7.4.3. Floating Point Arguments

When floating-point hardware is supported, they are passed in floating-point registers `fa0` to `fa7` similar to the way that integer values are passed. Returned floating-point values are returned in `fa0`.

When the floating-point data type is handled by software, those values are passed in the usual integer registers instead.

7.4.4. A Big Call Experiment

To practice our knowledge of the calling convention, let's exercise a C main program calling the assembler routine `bigcall()`, with nine arguments of type `int32_t` and `int64_t` using the `ESP32-C3`. Since this is an `XLEN=32` platform, any `int64_t` values will be split across two registers. The simple main program is illustrated in Listing 7.3. The routine `bigcall()` will return a simple sum of the arguments, with the limitation that only the lower 32 bits of each argument will be summed.

The `pragma` was added to optimize the C code to make examining its main program listing less confusing. Sometimes in unoptimized code, values are copied from register to register in a confusing and unnecessary manner.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern int bigcall(
7      int32_t one,
8      int64_t two,
9      int32_t three,
10     int64_t four,
11     int32_t five,
12     int64_t six,
13     int32_t seven,
14     int64_t eight,
15     int32_t nine
16 );
17
18 void
19 app_main(void) {
20     int rc = bigcall(101,102,103,104,105,106,107,108,109);
21
22     printf("bigcall() returned %d\n",rc);
23 }

```

Listing 7.3: Main C Program, ~/riscv/repo/07/call/main/main.c.

The code of interest is found in Listing 7.4, for the assembler routine bigcall().

```

1      .global bigcall
2
3      .struct 0
4  svfp: .space 4      # Save register fp/s0
5  svra: .space 4      # Save register ra
6  var1: .space 4      # Sample stack variable
7      .balign 8      # Make sz mod 8
8  sz    =      . - svfp
9
10     .struct 0
11  sixpt2: .space 4    # High order arg 6 (int64)
12  seven:  .space 4    # Arg 7 (int32)
13  eight:  .space 8    # Arg 8 (int64)
14  nine:   .space 4    # Arg 9 (int32)
15
16     .text
17  bigcall:
18     addi  sp,sp,-sz   # Set sp for stack frame

```

```

19     sw    fp,svfp(sp)    # Save fp/s0
20     sw    ra,svra(sp)   # Save ra
21     addi  fp,sp,+sz     # Set fp = original sp
22
23     add   a0,a0,a1      # Add low order arg2 (int64) to arg1
24     add   a0,a0,a3      # Add arg3 to arg1
25     add   a0,a0,a4      # Add low order arg4 (int64) to arg1
26     add   a0,a0,a6      # Add arg5 to arg1
27     add   a0,a0,a7      # Add low order arg6 (int64) to arg1
28
29     lw    t0,seven(fp)  # Load arg7
30     add   a0,a0,t0      # Add arg7 to arg1
31
32     lw    t0,eight(fp)  # Load arg8 (low order of int64)
33     add   a0,a0,t0      # a0 += arg8
34
35     lw    t0,nine(fp)   # Load arg8
36     add   a0,a0,t0      # Sum is return value
37
38     sw    x0,var1(sp)   # Zero sample stack variable
39
40     lw    ra,svra(sp)   # Restore ra
41     lw    fp,svfp(sp)   # Restore fp/s0
42     addi  sp,sp,+sz     # Restore sp
43     ret

```

Listing 7.4: `~/riscv/repo/07/call/main/bigcall.S`, assembler module for `bigcall()`.

This source file introduces a few new concepts. We could hard code the stack offsets for the overflow arguments and the saved register storage. But this is tedious and error-prone. So we make use of the ".struct" pseudo-op in lines 3 and 10. Focusing on the group starting on line 3, this starts an absolute definition, starting at address 0. This is *different* than defining a memory section because it is not actually allocated to any memory. These definitions appear in the listing as a section named "`*ABS*`", which is not an actual section. For example, a listing of `bigcall.S` would report:

```

DEFINED SYMBOLS
main/bigcall.S:17      .text:0000000000000000 bigcall
                      *ABS*:0000000000000000 svfp
                      *ABS*:0000000000000004 svra
                      *ABS*:0000000000000008 var1
main/bigcall.S:3      *ABS*:0000000000000010 sz
                      *ABS*:0000000000000000 sixpt2
                      *ABS*:0000000000000004 seven
                      *ABS*:0000000000000008 eight
                      *ABS*:0000000000000010 nine

```

Unlike the symbol `bigcall`, which is the name of our global entry point in the `.text` section, the symbol `svfp` is registered to an absolute area starting at absolute offset 0. References to memory section symbols get relocated by the linker, while these *absolute* symbols *do not*. This is important since we want the offsets to be computed but *not* relocated.

Looking at the structure definition:

```

3      .struct 0
4  svfp: .space 4          # Save register fp/s0
5  svra: .space 4          # Save register ra
6  var1: .space 4          # Sample stack variable
7      .balign 8          # Make sz mod 8
8  sz   =      . - svfp

```

we see that the symbol `svfp` is assigned an address (offset) of 0. The `".space"` pseudo-op in line 4 tells the assembler to reserve 4 more bytes before looking at line 5. This effectively moves the current location without defining any content.

Line 5 defines the symbol `svra` as offset 4, while line 6 defines `var1` as offset 12. Before we compute the size of the stack frame in line 8, we apply the `".balign"` pseudo-op to make the stack frame size aligned to an 8-byte boundary (for `XLEN=32`). For RV64, we would use 16 instead. Finally, `sz` is computed in line 8, which is the current location minus the start (`svfp`).

Figure 7.2 illustrates the layout of the saved registers and the excess arguments that are passed in the call after the function prologue has completed.

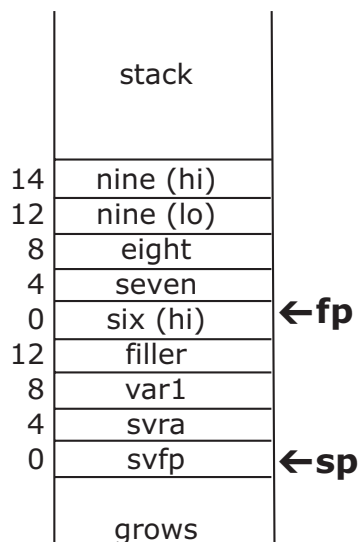


Figure 7.2: Diagram of stack layout for `bigcall.S`.

The saved word `svfp` is at offset 0 from the stack pointer (address `0(sp)`). The next word for saving `ra` at offset `svra` is at address `4(sp)`. Our example stack variable `var1` is at `8(sp)`, which we will later just zero to demonstrate accessing local variables. The saved calling arguments are available as offset from `fp/s0`. For example, argument seven is available as `4(fp)` (or `4(s0)`).

With these save offsets symbolically defined by the assembler, we can use them in the following prologue code:

```
18      addi    sp,sp,-sz      # Set sp for stack frame
19      sw     fp,svfp(sp)    # Save fp/s0
20      sw     ra,svra(sp)    # Save ra
21      addi    fp,sp,+sz     # Set fp = original sp
```

Line 18 adjusts the stack pointer by the size of the stack frame (16 bytes in this case). This maintains the alignment of the stack pointer. Lines 19 and 20 save registers `fp/s0` and `ra` to the stack. In this example, I am assuming that `ra` is going to be modified (perhaps by another call) so it must be saved. We modify the value of `fp/s0` in line 21, so it too must be saved. We don't have to initialize `var1`, so that isn't done in this prologue.

The first group of arguments that fit into the registers `a0` to `a7`, can be accessed directly:

```
23      add    a0,a0,a1      # Add low order arg2 (int64) to arg1
24      add    a0,a0,a3      # Add arg3 to arg1
25      add    a0,a0,a4      # Add low order arg4 (int64) to arg1
26      add    a0,a0,a6      # Add arg5 to arg1
27      add    a0,a0,a7      # Add low order arg6 (int64) to arg1
```

This code sums those arguments into `a0` (our return register) for arguments one through six. It's a little confusing with `a7` representing argument six but keep in mind that some of the arguments were 64 bits in size and required the use of a register pair.

The remaining arguments were placed on the stack by the calling C program. Review Figure 7.2 again. We use the frame pointer (`fp/s0`) register to access these arguments according to the offset symbols we defined in the structure starting in line 10. Since temporary registers like `t0` don't need to be preserved, we use it temporarily to load and sum the extra argument values:

```
29      lw     t0,seven(fp)  # Load arg7
30      add    a0,a0,t0      # Add arg7 to arg1
31
32      lw     t0,eight(fp)  # Load arg8 (low order of int64)
33      add    a0,a0,t0      # a0 += arg8
34
35      lw     t0,nine(fp)   # Load arg9
36      add    a0,a0,t0      # Sum is return value
```

The argument seven word is loaded into temporary register t0 (x5) and then added to register a0. The same is done for arguments eight and nine.

Just for fun, we zero our stack variable var1 in line 38. This example demonstrates how you might allocate and use stack variables when required.

```
38      sw      x0,var1(sp)    # Zero sample stack variable
```

After the sum is computed and var1 is zeroed, we are ready to return to the calling program. This is where the epilogue is applied:

```
40      lw      ra,svra(sp)    # Restore ra
41      lw      fp,svfp(sp)    # Restore fp/s0
42      addi    sp,sp,+sz      # Restore sp
43      ret
```

The offsets svra and svfp are relative to the current stack pointer (sp). These offsets are used with the stack pointer to restore values for register ra and fp. After that, we must also restore the sp itself by adding to it the same offset that was subtracted from it upon entry (line 18). The last step in this epilogue is to "ret", which returns control to the caller (register ra (x1) is assumed by the pseudo-op by default).

When the program is flashed and executed, you should see the sum printed:

```
$ idf.py build
...
$ idf.py -p <<<your-port>>> flash monitor
...
bigcall() returned 945
```

Success!

This may seem like a lot of effort to orchestrate this function call and indeed it was. But do keep in mind that most functions do not have so many arguments, which force the overflow onto the stack. When the C code is calling, you only have to worry about where to find the arguments. They are placed on the stack for you by the C compiler before the call is made and automatically released after the return.

7.5. Calling printf()

So far, we have relied upon the C code to do the reporting through printf(). Our last example can be amended slightly to perform the printf call from within the function instead as an example. Our function prologue already saves and restores the fp/s0 and the ra registers, so we can simply add a call to printf from within the assembler module. Listing 7.5 illustrates our usual main program, except that it only invokes bigcall2(), without saving a return value this time (declared to return void). Our new example code is found in the ~/riscv/repo/07/bigcall2 directory.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern void bigcall2(
7      int32_t one,
8      int64_t two,
9      int32_t three,
10     int64_t four,
11     int32_t five,
12     int64_t six,
13     int32_t seven,
14     int64_t eight,
15     int32_t nine
16 );
17
18 void
19 app_main(void) {
20
21     bigcall2(101,102,103,104,105,106,107,108,109);
22 }

```

Listing 7.5: Main program, ~/riscv/repo/07/bigcall2/main/main.c ESP32-C3.

The program illustrated in Listing 7.6 is much the same as before with the following changes:

1. Line 36 moves the sum to register a1, to act as a second int argument to the printf() call.
2. Line 40 uses the pseudo-op "la" to load an address into a0. This is the pointer to the format string to be passed to printf().
3. Line 42 calls printf(), with the arguments in registers a0 and a1. Recall that the call (by default) clobbers register ra (x1) with the printf() return address, which is why the register was saved in the function prologue.
4. Line 43 handles the value returned by printf(), and saves it in our stack variable var1. We don't actually use it here but it demonstrates the use of a stack-based variable.
5. Lines 52 and 53 define the printf() format string in a read-only section used for literals.

```

1      .global bigcall2, printf
2
3      .struct 0
4  svfp: .space 4          # Save register fp
5  svra: .space 4          # Save register ra

```



```

6  var1:  .space 4           # Example stack variable
7         .balign 8         # Keep stack size mod 8
8  sz     =      . - svfp
9
10        .struct 0
11  sixpt2: .space 4         # High order arg 6 (int64)
12  seven:  .space 4         # Arg 7 (int32)
13  eight:  .space 8         # Arg 8 (int64)
14  nine:   .space 4         # Arg 9 (int32)
15
16        .text
17  bigcall2:
18      addi  sp,sp,-sz       # Set sp for stack frame
19      sw    fp,svfp(sp)    # Save fp/s0
20      sw    ra,svra(sp)    # Save ra
21      addi  fp,sp,+sz       # Set fp = original sp
22
23      add   a0,a0,a1        # Add low order arg2 (int64) to arg1
24      add   a0,a0,a3        # Add arg3 to arg1
25      add   a0,a0,a4        # Add low order arg4 (int64) to arg1
26      add   a0,a0,a6        # Add arg5 to arg1
27      add   a0,a0,a7        # Add low order arg6 (int64) to arg1
28
29      lw    t0,seven(fp)    # Load arg7
30      add   a0,a0,t0        # Add arg7 to arg1
31
32      lw    t0,eight(fp)    # Load arg8 (low order of int64)
33      add   a0,a0,t0        # a0 += arg8
34
35      lw    t0,nine(fp)     # Load arg8
36      add   a1,a0,t0        # Sum to be printed
37
38  #      Print the result
39
40      la    a0,fmt          # Pointer address to format string
41      # a1 already has the int to print
42      call  printf
43      sw    a0,var1(sp)     # Save return value to var1 from print
44
45  #      Epilogue
46
47      lw    ra,svra(sp)     # Restore ra
48      lw    fp,svfp(sp)    # Restore fp/s0
49      addi  sp,sp,+sz       # Restore sp
50      ret
51

```

```

52         .section .rodata
53  fmt:   .string "bigcall2() computed a sum of %d\n"

```

Listing 7.6: Assembler routine, ~/riscv/repo/07/bigcall2/main/bigcall2.S.

The "la" opcode in line 40 is a pseudo-op that results in the destination register receiving the desired *address* (not its value). Like the "call" pseudo-op, "la" can result in one or two actual opcodes to accomplish this. The printf() call needs a pointer to a format string in a0, so the "la" opcode accomplishes this for us. If you were to objdump the build/bigcall2.elf executable that was linked, you would find that it does indeed turn into two opcodes.

```

42005e62:    fa01f517    auipc    a0,0xfa01f
42005e66:    9df50513    addi    a0,a0,-1569 # 3c024841 <fmt>

```

When you flash and run the program, you should get the message:

```
bigcall2() computed a sum of 945
```

indicating success. The format of the message was deliberately changed so that you can be assured that it was the assembler program printing the message this time.

7.6. Summary

I hope you are feeling confident now about one of the more difficult aspects of RISC-V assembly language. It is vital that the register convention be understood for saving and restoring registers. You witnessed how the register passing works including those that were passed on the stack itself. The presented example programs illustrate the function prologue and epilogue.

Using the assembler ".struct" pseudo-op, you learned how to define symbolic offsets to stack frame components. This is an important tool because it saves you from having to use the brittle hard coding of offsets. Allow the assembler to do the mental arithmetic for you. It is also more amenable to changes later on.

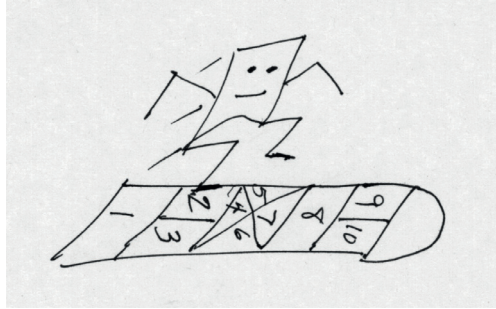
The new opcode "la" for load address was introduced without much fanfare. But its usefulness for loading a pointer value into a register will become more relevant in the chapters ahead.

There will sometimes exist a temptation to violate the call convention in the name of efficiency. Some advocate "just don't".^[1] Certainly recognize that there is a risk when you do. Accept that it is probably ill-advised for medical or safety-critical systems. Be aware that interrupts will also use the stack while your code executes so your abnormal convention must be able to tolerate that. If you are stuck debugging something weird, this is probably one of the first things to be checked.

Bibliography

- [1] *RISC-V reference* - Simon Fraser University. (n.d.). Retrieved May 11, 2022, from https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf

Chapter 8 • Flow Control



CPU jumping a over a hopscotch court

To this point we haven't explored branching other than calling and returning from a sub-routine call. This chapter examines some fun programs to exercise branching instructions. Branching is a fundamental part of a loop, which allows repeated operations.

8.1. Branching Instructions

The RISC-V documents like to refer to these as control transfer instructions. Of these, there are two basic categories:

- Unconditional transfers
- Conditional branches

8.1.1. Unconditional Transfers

We've already seen the "jal" and "jalr" opcodes for calling a subroutine. When the destination register is set to zero/x0, either explicitly or by a pseudo-op like "j", they become simple unconditional transfers because there is no return address produced. The "jal" and "j" opcodes have the ability to jump +/- 1 MB from the current program counter (pc) since the offset is encoded as a signed half-word offset. The "jalr" opcode on the other hand encodes the offset as a 12-bit signed byte offset and therefore has a more restricted range.

The following are variations of the unconditional transfer. For easy-to-read code, the "j" or "jr" pseudo-opcodes are recommended:

jal	x0,offset	# pc += offset
j	offset	# same as jal x0,offset
jr	offset(rs)	# rd is assumed to be x0, pc += rs + offset
jr	rs	# same as jr 0(rs)
jalr	x0,offset(rs)	# same as jr offset(rs)

When coding these transfers, there is no need to compute word or byte offsets. The assembler knows which offset to provide when you provide a branch label. However, if you provide numeric offsets, you will need to use the correct form.

8.1.2. Conditional Branches

With conditional branches, the comparison and the branch are combined into one operation. The following are signed comparison branches (the operation in C language terms is provided in the comment):

```

beq    rs1,rs2,offset    # branch when rs1 == rs2
bne    rs1,rs2,offset    # branch when rs1 != rs2
blt    rs1,rs2,offset    # branch when rs1 < rs2
bge    rs1,rs2,offset    # branch when rs1 >= rs2

```

Depending upon the result of the comparison between registers `rs1` and `rs2`, the branch is made by modifying register `pc`, or allowing the execution to resume at the next instruction. There are no status flags saved from the comparison in RISC-V.

Since the comparison to zero comes up frequently, the following pseudo-ops are also available:

```

beqz   rs,offset        # branch when rs == 0
bnez   rs,offset        # branch when rs != 0
bltz   rs,offset        # branch when rs < 0
bgez   rs,offset        # branch when rs >= 0

```

These are just encodings of the previous opcodes. For example, "bnez" is the same as:

```

bne    rs1,x0,offset    # branch when rs1 != x0

```

If you need unsigned comparisons, then the following additional opcodes are available:

```

bltu   rs1,rs2,offset    # branch when rs1 < rs2
bgeu   rs1,rs2,offset    # branch when rs1 >= rs2

```

Branch instructions make a CPU smarter than a simple calculator, by giving it the ability to make decisions.

8.2. Shift Opcodes

So far, we've exercised a restricted set of opcodes for computing values. Let's expand that with the addition of shift operators before we embrace the exercises in branching. The following opcodes are available in RV32I and RV64I:

```

sll    rd,rs1,rs2        # rd = rs1 << rs2 (shift left logical)
slli   rd,rs1,imm        # rd = rs1 << imm (shift left logical immediate)
srl    rd,rs1,rs2        # rd = rs1 >> rs1 (shift right logical)
srli   rd,rs1,imm        # rd = rs1 >> imm (shift right logical immediate)
sra    rd,rs1,rs2        # rd = rs1 >> rs2 (shift right arithmetic)
srai   rd,rs1,rs2        # rd = rs1 >> imm (shift right arithmetic)

```

In all of the above, the result goes into the destination register `rd`. The value that is shifted originates in register `rs1`. The shift count originates from either source register `rs2` or from an immediate constant. Finally, the opcode determines the type of shift – its direction and whether it is logical or an arithmetic shift. The following is an example:

```
slli    a0,a1,1        # a0 = a1 << 1 (shift a1 logically left 1 bit)
```

In the C language, you often don't consider whether it involves a logical or arithmetic shift operation. That is because it is determined by the data type. When unsigned, it requires a logical shift. When signed, it requires an arithmetic shift, when shifting right. The arithmetic right shift preserves the sign while the remaining bits are shifted right. All shift-left operations are logical shifts, which is why you don't see a "sla" opcode. Logical shifts always populate the shifted-out bit position with a zero.

RV64I Shift Opcodes

In addition to the above, RV64I adds these:

```
slliw   rd,rs1,imm    # rd = rs1 << imm (shift left logical immediate)
srliw   rd,rs1,imm    # rd = rs1 >> imm (shift right logical immediate)
sraiw   rd,rs1,rs2    # rd = rs1 >> imm (shift right arithmetic)
```

These shift operations take the *lower 32-bit value* in `rs1`, perform the shift and then *sign extend* the 32-bit result into the destination register for 64 bits. For example, if register `a0` holds the value:

```
a0      0x7fffffffffffffff
```

After the instruction:

```
sraiw a0,a0,1        # a0 = a0 << 1 (32-bits)
```

executes, `a0` would hold the result value:

```
a0      0xfffffffffffffffe
```

In other words, the sign was taken from bit 31 after the shift was performed and then extended out to the full 64 bits in the destination register. This permits an easy interchange of signed 32-bit values into the 64-bit environment.

8.3. ESP32-C3 Project

We now have enough resources available to test our knowledge of conditional branches. In this project, we'll write both an optimized C language program and an assembler language equivalent to count the number of 1-bits in a 32-bit integer, for the ESP32-C3. Let's then compare the two routines in this bake-off to see if the compiler can out-perform our assembly language code.

8.3.1. Function `c_ones()`

Listing 8.1 illustrates our test C language program, optimized at GCC level `-O3` for good performance (line 4). Our assembler routine has the function prototype defined in line 6. The C version of the same routine is defined in lines 8 to 16. In it we start out with a zeroed count (line 9) and keep testing for the int (32-bits) value of bits to be less than zero, indicating a 1-bit in bit 31, the sign bit (line 12). If the value is negative, we increment the count (line 13). At the end of each loop, we shift the value of bits left by one bit (line 11). If along the way, the value of bits becomes zero, we exit the loop and return the accumulated count (line 15).

8.3.2. Main Test Program

The main program first invokes the assembler routine `ones()` in line 27 and saves the bit count in variable `bcount`. Then the C language function is called in line 31 with the returned value saved in variable `ccount`. If there is any discrepancy between these two results, the discrepancy is reported in lines 33 and 34. Otherwise, we repeat the test with different test values from the static array defined in lines 20 to 22.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern int ones(int bits);      // Assembler routine
7
8  static int c_ones(int bits) {
9      int count = 0;
10
11     for ( ; bits != 0; bits <<= 1 ) {
12         if ( bits < 0 )
13             ++count;
14     }
15     return count;
16 }
17
18 void
19 app_main(void) {
20     static int tests[] = { 0, 1, 2, 3, 5, 7, 9, 15,
21                          31, 63, 64, 127, 1023, 1024, 2047, 2048,
22                          9999 };
23     int bcount, ccount, bits;
24
25     for ( unsigned ux=0; tests[ux] < 9999; ++ux ) {
26         bits = tests[ux];
27         bcount = ones(bits);
28         printf("ones(%4d) (0x%04X) returned %d\n",
29              bits, bits,

```

```

30             bcount);
31             ccount = c_ones(bits);
32             if ( ccount != bcount )
33                 printf("c_ones(%4d) did not agree with %d!\n",
34                        ccount,bcount);
35         }
36         printf("Done.\n");
37     }

```

Listing 8.1: Main program with c_ones() function, ~/riscv/repo/08/ones/main/main.c.

8.3.3. Assembler Function ones()

Our assembly language function ones() is illustrated in Listing 8.2. In this routine we initialize temporary register t0 with zero (line 3) then we drop into the loop starting at line 5. There we test if the argument in a0 is already zero or has become zero. If so, we branch to the label done.

When the value a0 is non-zero, we continue in line 6, testing if the sign bit is positive using "bge". If it is positive, we skip over the instruction in line 7. Otherwise, we continue execution in line 7 and increment the count in t0 (line 7). Execution continues in line 8 where the value of a0 is locally shifted left 1 bit. At the end of the loop in line 9, we unconditionally branch back to the top of the loop at line 5.

When the branch is finally taken to the label done (line 11), the count in register t0 is moved to a0 in order to return the bit count. Finally, we return to the caller in line 12.

```

1     .global ones
2     .text
3 ones:  mv     t0,zero      # t0 = 0
4
5 loop:  beq    a0,zero,done # Branch if a0 == 0
6        bge   a0,zero,shift # Skip next if sign is positive
7        addi  t0,t0,1      # t0 += 1
8 shift: slli  a0,a0,1      # a0 <<= 1
9        j     loop        # Repeat loop
10
11 done:  mv     a0,t0       # a0 = count in t0
12        ret

```

Listing 8.2: Assembly language function ones(), ~/riscv/repo/08/ones/main/ones.S.

Run it

Build, flash and monitor the result from the ESP32-C3 device:

```
$ cd ~/riscv/repo/08/ones
$ idf.py build
...
idf.py -p <<<your-port>>> flash monitor
...
ones( 0) (0x0000) returned 0
ones( 1) (0x0001) returned 1
ones( 2) (0x0002) returned 1
ones( 3) (0x0003) returned 2
ones( 5) (0x0005) returned 2
ones( 7) (0x0007) returned 3
ones( 9) (0x0009) returned 2
ones( 15) (0x000F) returned 4
ones( 31) (0x001F) returned 5
ones( 63) (0x003F) returned 6
ones( 64) (0x0040) returned 1
ones( 127) (0x007F) returned 7
ones(1023) (0x03FF) returned 10
ones(1024) (0x0400) returned 1
ones(2047) (0x07FF) returned
ones(2048) (0x0800) returned 1
Done.
```

How did we do? No complaints were issued, so this confirms that both the assembly language function and the C language functions agreed. Examination of the return values indicates that the results are correct.

Routines Compared

Was our assembler routine better than the C language one? Let's first examine the C language listing in assembly language:

```
$ ~/riscv/repo/listesp main/main.c
```

The extract of the portion of the listing for the `c_ones()` function is shown in Listing 8.3.

```
6          c_ones:
7 0000 AA87          mv      a5,a0
8 0002 0145          li      a0,0
9 0004 99C7          beqz   a5,.L4
10         .L4:
11 0006 13A70700      slti   a4,a5,0
12 000a 8607          slli   a5,a5,1
13 000c 3A95          add    a0,a0,a4
```

```

14 000e e5ff          bnez   a5, .L4
15 0010 8280          ret
16                .L5:
17 0012 8280          ret

```

Listing 8.3: Extract of the C Language `c_ones()` function listing.

It's clear from this listing that the C compiler chose to generate this function a little differently than our own assembler routine. Let's break it down and compare.

Line 7 moves the argument value "bits" to a5 as its first step. Then it sets a0 to zero in line 8. If the value of a5 is already zero, control passes to .L5 (line 16) where the routine returns to the caller (with zero held in a0).

Otherwise, the top of the loop in lines 10 and 11 is entered. Line 11 uses an opcode "slti" that we've not yet covered. When the value in a5 is less than 0 (the immediate value), then the value placed into a4 is the value 1. Otherwise, a4 receives the value zero (line 11). After that, it shifts the bits in a5 left 1 bit (line 12). Line 13 adds the value of a4 to a0. We know that a4 will be the value 0 or 1 from line 11. Finally, if a5 is not equal to zero in line 14, execution resumes at the top of the loop at .L4. Otherwise, the execution falls through to line 15, returning the bit count in a0.

```

3 0000 93020000      ones:  mv     t0,zero      # t0 = 0
4
5 0004 11c5          loop:  beq     a0,zero,done    # Branch if a0 == 0
6 0006 63530500          bge     a0,zero,shift    # Skip next if sign is +
7 000a 8502          addi   t0,t0,1           # t0 += 1
8 000c 0605          shift: slli   a0,a0,1     # a0 <<= 1
9 000e ddbf          j      loop              # Repeat loop
10
11 0010 1685          done:  mv     a0,t0        # a0 = count in t0
12 0012 8280          ret

```

Listing 8.4: Extract listing of `main/ones.S`.

It is readily apparent that both functions are the same number of bytes of object code (20 bytes). So which routine is faster? The `c_ones()` routine must execute the "addi" instruction even when the register a4 is zero:

```

13 000c 3a95          add    a0,a0,a4

```

In this routine's favour however, there is one less branch involved in each loop. The `ones.S` program has a "bge" branch in line 6, whereas the `c_ones()` function uses the "slti" in line 11 instead. The practical difference is going to boil down to the silicon used.

Because of pipelining and potential branch prediction, it is best to avoid unnecessary branches. With that in mind, the optimized GCC code is an improvement over our assembly code. Espressif states that the ESP32-C3 has a "4-stage, in-order, scalar pipeline" but nothing is said about branch prediction. On the ESP32-C3 therefore, there might not be any difference between these. On more advanced silicon however, branches are best avoided where possible.

8.4. Compare and Set

As reviewed in the examination of the `c_ones()` function, the C compiler used the "slti" opcode to set the destination register to a 1 or a 0 based upon the result of a comparison. There are other compare and set opcodes similar to it. These are listed with their C language equivalent expression in the comments:

```

slt    rd,rs1,rs2    # rd = rs1 < rs2 ? 1 : 0
slti   rd,rs1,imm   # rd = rs1 < imm ? 1 : 0
sltu   rd,rs1,rs2   # rd = (unsigned)rs1 < (unsigned)rs2 ? 1 : 0
sltiu  rd,rs1,imm   # rd = (unsigned)rs1 < (unsigned)imm ? 1 : 0

seqz   rd,rs         # rd = rs == 0 ? 1 : 0

snez   rd,rs         # rd = rs != 0 ? 1 : 0
sltz   rd,rs         # rd = rs < 0 ? 1 : 0
sgtz   rd,rs         # rd = rs > 0 ? 1 : 0

```

It should be clear that these operations have direct application for C/C++. Consider the C++ code:

```

bool zflag;
int x;

...
zflag = !x;           // Set zflag true if x == 0 else false

```

In this example, if register `t0` holds the value of `zflag` and `a0` holds the value of `x`, then the C compiler could simply emit:

```

seqz   t0,a0         // Set t0=1 if x in a0 is zero else t0=0

```

8.5. Odd Parity Example

Let's try another example, that is perhaps a little more practical. Our assembler routine named `odd_parity()` will determine if there is an odd or even number of 1 bits. The return value will be 1 if the parity is odd, else it returns 0 for even parity.

```

1 .global odd_parity
2     .text
3 odd_parity:
4     li    t0,0        # t0 = 0
5
6 loop:  beq    a0,zero,done # Branch if a0 == 0
7         sltz   t1,a0      # t1 = a0 < 0 ? 1 : 0
8         xor    t0,t0,t1   # t0 = t0 ^ t1
9         slli   a0,a0,1    # a0 <<= 1
10        j     loop       # Repeat loop
11
12 done:  mv     a0,t0      # a0 = count in t0
13        ret

```

Listing 8.5: The `odd_parity()` function, `~/riscv/repo/08/parity/main/parity.S`

In this program, we chose to use "li" in line 4 this time to initialize t0 to zero just for fun. The execution continues to the top of the loop in line 6, where a0 is tested for the value of zero. If it is zero, control moves to line 12, where the parity value is placed in a0, and then returned to the caller (line 13).

Otherwise, when line 6 fails to branch, it means we still have 1 bits to test for parity. Line 7 sets the temporary register t1 to a 1 if the value is less than zero (indicating that the sign bit is true), else t1 is cleared to zero. The "xor" opcode in line 8 applies an exclusive-or between the value in t0 and t1. Since either register only has the low order bit set (or not), the end result is a 1 or a 0, indicating the current parity value. Recall that an exclusive-or operation flips bits when one or the other bit is a 1-bit, but not if both are the same. So, every time through the loop, as we encounter more 1-bits, the value of t0 (parity) is inverted.

At the end of the loop in lines 9 and 10, we shift the value in a0 logically left by the 1-bit position, populating a new bit in the sign bit (bit 31 for the ESP32-C3) and repeat the loop.

Listing 8.6 illustrates the main program for driving the test, which is much the same as before. The function `odd_parity()` is called with various values from the array `tests[]` and reported in the `printf()` call in lines 18 and 19.

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 #pragma GCC optimize("-O3")
5
6 extern int odd_parity(int bits); // Assembler routine
7
8 void
9 app_main(void) {
10     static int tests[] = { 0, 1, 2, 3, 5, 7, 9, 15,

```

```

11         31, 63, 64, 127, 1023, 1024, 2047, 2048,
12         9999 };
13     int oddpar, bits;
14
15     for ( unsigned ux=0; tests[ux] < 9999; ++ux ) {
16         bits = tests[ux];
17         oddpar = odd_parity(bits);
18         printf("odd_parity(%4d) (0x%04X) returned %d\n",
19             bits, bits, oddpar);
20     }
21     printf("Done.\n");
22 }

```

Listing 8.6: The main test program, ~/riscv/repo/08/parity/main/main.c.

Build, flash and monitor this ESP32-C3 project as follows:

```

$ cd ~/riscv/repo/08/parity
$ idf.py build
...
$ idf.py -p <yourport> flash monitor
...
odd_parity( 0) (0x0000) returned 0
odd_parity( 1) (0x0001) returned 1
odd_parity( 2) (0x0002) returned 1
odd_parity( 3) (0x0003) returned 0
odd_parity( 5) (0x0005) returned 0
odd_parity( 7) (0x0007) returned 1
odd_parity( 9) (0x0009) returned 0
odd_parity(15) (0x000F) returned 0
odd_parity(31) (0x001F) returned 1
odd_parity(63) (0x003F) returned 0
odd_parity(64) (0x0040) returned 1
odd_parity(127) (0x007F) returned 1
odd_parity(1023) (0x03FF) returned 0
odd_parity(1024) (0x0400) returned 1
odd_parity(2047) (0x07FF) returned 1
odd_parity(2048) (0x0800) returned 1

```

Checking the results, you can verify that the odd parity was computed for each test value.

8.6. RV64I Odd Parity

For those itching to put Fedora Linux to work, let's do a 64-bit version of the `odd_parity()` function. Listing 8.7 lists the main program that drives this test. In this example, notice that `odd_parity()` now accepts a 64-bit integer argument for parity tests. The main program has added a few more test values but is otherwise is the same as before:

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern int odd_parity(int64_t bits); // Assembler routine
7
8  int
9  main(int argc, char **argv) {
10     static int64_t tests[] = { 0, 1, 2, 3, 5, 7, 9, 15,
11                               31, 63, 64, 127, 1023, 1024, 2047, 2048,
12                               65535, 65536, 0xF0F0900000000011ll, 999999 };
13     int64_t bits;
14     int oddpar;
15
16     for ( unsigned ux=0; tests[ux] != 999999ll; ++ux ) {
17         bits = tests[ux];
18         oddpar = odd_parity(bits);
19         printf("odd_parity(%20lld) (0x%016llx) returned %d\n",
20              bits, bits, oddpar);
21     }
22     printf("Done.\n");
23 }

```

Listing 8.7: Main program listing, ~/riscv/repo/08/parity/qemu64/main.c.

The parity.S assembly language program shown in Listing 8.8 is exactly the same program that we used on the ESP32-C3. The difference in this instance, however, is that the registers are 64 bits in width (XLEN=64). So, the 64-bit argument is received in register a0 (line 6). The 64 bits are shifted left (logically) in line 9 (vs 32 bits as it was on the ESP32-C3).

```

1      .global odd_parity
2      .text
3  odd_parity:
4      li      t0,0          # t0 = 0
5
6  loop:  beq    a0,zero,done  # Branch if a0 == 0
7      sltz   t1,a0          # t1 = a0 < 0 ? 1 : 0
8      xor    t0,t0,t1      # t0 = t0 ^ t1
9      slli   a0,a0,1       # a0 <<= 1
10     j      loop          # Repeat loop
11
12  done:  mv     a0,t0       # a0 = count in t0
13     ret

```

Build and execute the program as follows:

```
$ cd ~/riscv/repo/08/parity/qemu64
$ gcc -g parity.S main.c
$ ./a.out
odd_parity( 0) (0x000000) returned 0
odd_parity( 1) (0x000001) returned 1
odd_parity( 2) (0x000002) returned 1
odd_parity( 3) (0x000003) returned 0
odd_parity( 5) (0x000005) returned 0
odd_parity( 7) (0x000007) returned 1
odd_parity( 9) (0x000009) returned 0
odd_parity(15) (0x00000F) returned 0
odd_parity(31) (0x00001F) returned 1
odd_parity(63) (0x00003F) returned 0
odd_parity(64) (0x000040) returned 1
odd_parity(127) (0x00007F) returned 1
odd_parity(1023) (0x0003FF) returned 0
odd_parity(1024) (0x000400) returned 1
odd_parity(2047) (0x0007FF) returned 1
odd_parity(2048) (0x000800) returned 1
odd_parity(65535) (0x00FFFF) returned 0
odd_parity(65536) (0x010000) returned 1
Done.
```

Listing 8.8: Assembly language routine, ~/riscv/repo/08/parity/qemu64/parity.S.

This parity function worked on 64 bits, just as it did with 32 for the ESP device. But there is a new efficiency concern. The loop iterations may run up to 64 times before returning to this platform. For XLEN=32-bit platforms, the maximum loop count was half of that. If you were computing parity on byte values that were loaded into an `int64_t` argument, the loop would be idle for 87.5% of the bits processed. Perhaps this can be improved upon.

Listing 8.9 shows a somewhat improved version of `odd_parity()` in file `parity2.S`. What is new is that we preload register `t2` with a mask value of 32 1-bits in the upper half of the register. Line 5 places all 1 bits into `t2`, while line 6 shifts it left logically 32 bits. Recall that the immediate constant is sign-extended when loaded. This mask value permits us to check if the upper 32-bits of `a0` are all zeros or not (lines 11 and 12). If they are zero, we branch to line 9 to simply shift the `a0` register left by 32 bits, to avoid iterating 32 more times to get to that point. Note that no parity adjustment is required for this.

```
1      .global odd_parity
2      .text
3  odd_parity:
4      li      t0,0          # t0 = 0
5      li      t2,-1        # t2 = 0xFFFFFFFFFFFFFFFF
```

```

6      slli   t2,t2,32      # t2 = 0xFFFFFFFF00000000
7      j      loop
8
9  bump32: slli   a0,a0,32
10     loop: beq   a0,zero,done # Branch if a0 == 0
11         and   a1,a0,t2      # a1 = upper 32 bits of a0
12         beqz  a1,bump32     # Branch if a1 is zeros
13         sltz  t1,a0         # t1 = a0 < 0 ? 1 : 0
14         xor   t0,t0,t1     # t0 = t0 ^ t1
15         slli  a0,a0,1      # a0 <<= 1
16         j      loop        # Repeat loop
17
18     done: mv    a0,t0       # a0 = count in t0
19         ret

```

Listing 8.9: *A, ~/riscv/repo/08/parity/qemu64/parity2.S.*

There is more that could be done to improve the efficiency, but each addition has its trade-offs. I'll leave that to you to experiment further.

Tip: To load a register with all 1 bits, take advantage of the sign extension of the immediate data. For example "li t2,-1" sign extends the 12-bit value of -1 (0xFFF) to the full XLEN width of the register.

This can also apply to other special mask values. For example, to create a mask for all but the low order byte, the instruction "li t2,-256" can be used. This sets all bits to 1 in t2, except for the low order byte because -256 (0xF00) is sign extended.

8.7. Position Independent Code

One advanced area that I've avoided talking about is position-independent code. Let's mention it here briefly so that those of you who jump into doing the difficult can be aware of the pitfalls.

Normally we write code to execute at one fixed address. But what happens if the code needs to be copied to a different address and executed there? Traditionally, this requires fixing up addresses of referenced routines that didn't move. Under Fedora Linux, for example, shared libraries need to be able to run in the shared memory segments, which are dynamically allocated. This creates two problems:

1. Labels of instructions that move with the code, need to be relative to the pc. In this way, moving the code does not upset branch references because they remain relative to the current address.
2. Fixed labels of routines or global data *that do not move*, require some special treatment in order to remain valid after the move.

Recall from the last chapter when we examined the far call to a function `foo()` at address `0x427F0000`:

```
#                call foo                # located at 0x427F0000
42005e16 007ea097  auipc  ra,0x7ea # ra = 0x42005e16 + 0x7ea000
42005e1a 1ea080e7  jalr   490(ra)  # ra = 0x42005e1a + 4, PC = 0x427f0000
```

That code relied on the fact that the "auipc" opcode does a calculation relative to the pc.

If we were to move this code to a new location and keep `foo` where it is, the value computed for the call to `foo()` would now be incorrect. This is due to the relative "auipc" calculation. To fix this as an absolute reference to `foo`, we can substitute the "auipc" opcode with "lui" (load upper immediate) instead. Instead of computing a relative result, the absolute value of `%hi(foo)` is placed in `ra` instead. Then when we call `lo%(foo)` plus `ra`, we arrive at the fixed address of `foo` (`0x427F0000`).

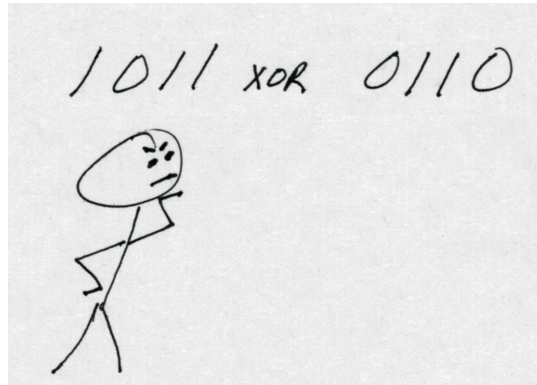
```
#                call foo                # located at 0x427F0000
1048e: 62c1      lui    ra,hi%(foo)
10490: 496282e7 jalr   lo%(foo)(ra)
```

For beginners, this kind of stuff might bother the brain. The good news is that, for the average routine, you can glibly use the "call" and otherwise ignore the issue. But those who write code that must be position independent will have to assume more responsibility.

8.8. Summary

This chapter covered unconditional jumps and conditional branches. You witnessed how RISC-V is able to function without condition flags. You've worked with the shift instructions and the compare and set operations. We compared an optimized C function against our own assembly language routine and found that it is not always trivial to beat the compiler. We even snuck in the use of the "xor" opcode. This and other opcodes are the subjects of the next chapter.

Chapter 9 • Basic Opcodes



Understanding the basics

Up to this point in this tutorial, things have been kept simple to avoid overwhelming the reader with details. But with the framework out of the way, you're at the point now where more functionality would be welcomed. This chapter introduces a number of additional opcodes, which are the essential building blocks for most RISC-V assembly language programs.

9.1. Arithmetic Opcodes

While we have used "add" before, let's list the arithmetic opcodes that RISC-V provides (all immediate values are signed):

```
add   rd,rs1,rs2      # rd = rs1 + rs2
addi  rd,rs1,imm      # rd = rs1 + imm
sub   rd,rs1,rs2      # rd = rs1 - rs2
lui   rd,imm          # rd = imm << 12
uipc  rd,imm          # rd = pc + (imm << 12)
```

9.1.1. add, addi and sub

You've already used "add" and "addi", which simply add two values. Notice that there is no "subi" (subtract immediate) opcode. The same effect can be accomplished by adding a negative immediate constant.

9.1.2. lui

The "lui" opcode provides a way to load 20 bits of unsigned data, into the high order 32 bits of a 32-bit register. We've discussed this one before. When registers are larger than 32 bits, the same applies except that bit 31 is sign-extended to the full XLEN bits for RV64 and larger. For example, the instruction:

```
lui a0,0xEEEEE
```

results in a0 receiving the value of 0xFFFFFFFFEEEE000 for RV64.

9.1.3. auipc

We've also seen this opcode before when it was used to do a relative address calculation. It adds the current location (register pc before it is incremented) to the unsigned immediate data (shifted up 20 bits). Like the "lui" opcode, bit 31 is sign-extended to the full register width for RV64 and larger platforms.

9.1.4. RV64 Arithmetic

In addition to the arithmetic opcodes just covered, RV64I includes the following additional opcodes:

```
addw   rd,rs1,rs2      # rd = rs1 + rs2
addiw  rd,rs1,imm      # rd = rs1 + imm
subw   rd,rs1,rs2      # rd = rs1 - rs2
```

These all behave as if the register result were only 32 bits in size. But once the result is computed, bit 31 is then sign-extended to the full width of the register. You might think of these as working on a word (32-bits) and then sign-extended.

9.2. Logical Opcodes

The logical opcodes permit the programmer to perform bit-wise logical operations. RISC-V provides the following operations (the C language equivalent expressions are shown in the comments):

```
and    rd,rs1,rs2      # rd = rs1 & rs2
andi   rd,rs1,imm      # rd = rs1 & signed(imm)
or     rd,rs1,rs2      # rd = rs1 | rs2
or     rd,rs1,imm      # rd = rs1 | signed(imm)
xor    rd,rs1,rs2      # rd = rs1 ^ rs2
xori   rd,rs1,imm      # rd = rs1 ^ signed(imm)
```

The operations of "and", "or" and "xor" should be familiar to the C/C++ language programmer. These boolean operations are summarized in Table 9.1. Remember that the immediate data constants are sign-extended to the full register width, even though the immediate constant itself is only 12-bits wide within the opcode.

Tip: To load a register with all 1-bits, load an immediate data value of -1, as in "li t0,-1". In this example, t0 is set to all 1-bits due to the sign extension of the constant -1.

Bit 1	Bit 2	Operation Result		
		And	Or	Xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 9.1: Basic logic operations.

9.3. ESP32-C3 Rotate Left

Time to exercise some code! Many microcomputer architectures support some kind of a rotate left or right instruction, which is noticeably absent in RISC-V. But the example program in Listing 9.1 demonstrates that this really is no hardship. Function `rotate_left()` rotates an `uint32_t` integer left one bit, placing the shifted out bit into bit position 0.

```

1      .global rotate_left
2      .text
3
4 rotate_left:
5      sltz    t0,a0          # t0=1 if bit 32=1
6      slli   a0,a0,1        # a0 <<= 1
7      or     a0,a0,t0        # Or in shifted out bit
8      ret

```

Listing 9.1: Rotate left assembler function, ~/riscv/repo/09/rol/rol.S.

The 32-bit unsigned value is passed in register `a0` (line 5). The "set less than zero" opcode then sets register `t0` to the value of 1 or 0 based upon the sign bit (bit 31) of `a0`. Line 6 then shifts that value in `a0` left one bit (this fills bit 0 with a zero). Finally, line 7 does a logical "or" of the bit in `t0`, which adds a 1-bit or leaves the register `a0` as it is, if `t0` happens to be zero. Finally, the value is returned in register `a0` (line 8).

The main driver program is shown in Listing 9.2. It simply iterates 32 times, rotating the value of `0xBEEF`, initialized in line 10. After iterating the full 32 times, the value should return to `0xBEEF`.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern uint32_t rotate_left(uint32_t bits); // Assembler routine
7
8  void
9  app_main(void) {
10     uint32_t bits = 0xBEEF, rol;
11
12     for ( unsigned ux=0; ux<32; ++ux, bits = rol ) {
13         rol = rotate_left(bits);
14         printf("rotate_left(0x%08X) returned 0x%08X\n",
15             bits,rol);
16     }
17     printf("Done.\n");
18 }

```

Listing 9.2: Main program from ~/riscv/repo/09/rol/main/main.c.

Build, flash and monitor the program as follows:

```
$ cd ~/riscv/repo/09/rol
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
rotate_left(0x0000BEEF) returned 0x00017DDE
rotate_left(0x00017DDE) returned 0x0002FBBC
rotate_left(0x0002FBBC) returned 0x0005F778
rotate_left(0x0005F778) returned 0x000BEEF0
rotate_left(0x000BEEF0) returned 0x0017DDE0
rotate_left(0x0017DDE0) returned 0x002FBBC0
rotate_left(0x002FBBC0) returned 0x005F7780
rotate_left(0x005F7780) returned 0x00BEEF00
rotate_left(0x00BEEF00) returned 0x017DDE00
rotate_left(0x017DDE00) returned 0x02FBBC00
rotate_left(0x02FBBC00) returned 0x05F77800
rotate_left(0x05F77800) returned 0x0BEEF000
rotate_left(0x0BEEF000) returned 0x17DDE000
rotate_left(0x17DDE000) returned 0x2FBBC000
rotate_left(0x2FBBC000) returned 0x5F778000
rotate_left(0x5F778000) returned 0xBEEF0000
rotate_left(0xBEEF0000) returned 0x7DDE0001
rotate_left(0x7DDE0001) returned 0xFBBC0002
rotate_left(0xFBBC0002) returned 0xF7780005
rotate_left(0xF7780005) returned 0xEEF0000B
rotate_left(0xEEF0000B) returned 0xDDE00017
rotate_left(0xDDE00017) returned 0xBBC0002F
rotate_left(0xBBC0002F) returned 0x7780005F
rotate_left(0x7780005F) returned 0xEF0000BE
rotate_left(0xEF0000BE) returned 0xDE00017D
rotate_left(0xDE00017D) returned 0xBC0002FB
rotate_left(0xBC0002FB) returned 0x780005F7
rotate_left(0x780005F7) returned 0xF0000BEE
rotate_left(0xF0000BEE) returned 0xE00017DD
rotate_left(0xE00017DD) returned 0xC0002FBB
rotate_left(0xC0002FBB) returned 0x80005F77
rotate_left(0x80005F77) returned 0x0000BEEF
Done.
```

At the end of the test run, the returned value is indeed 0xBEEF, which is the value that it started with. Success!

9.4. RV64 Rotate Left

Under RV64, we have 64-bit registers to work with and Listing 9.3 illustrates a version of the same for running under QEMU. The only difference is the comment in line 5 is modified to reflect 64 bits.

```

1      .global rotate_left
2      .text
3
4 rotate_left:
5      sltz    t0,a0          # t0=1 if bit 63=1
6      slli   a0,a0,1       # a0 <<= 1
7      or     a0,a0,t0      # Or in shifted out bit
8      ret

```

Listing 9.3: Program for rotate_left() function, ~/riscv/repo/09/rol/qemu64/rol.S.

The main program differs slightly in Listing 9.4. This time the test iterates 64 times, to arrive at the original value, due to the increase in register width.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern uint64_t rotate_left(uint64_t bits); // Assembler routine
7
8  int
9  main(int argc,char **argv) {
10     uint64_t bits = 0xBEEF, rol;
11
12     for ( unsigned ux=0; ux<64; ++ux, bits = rol ) {
13         rol = rotate_left(bits);
14         printf("rotate_left(0x%016llx) returned 0x%016llx\n",
15             bits,rol);
16     }
17     return 0;
18 }

```

Listing 9.4: The main program in ~/riscv/repo/09/rol/qemu64/main.c.

Compile and run this program as follows:

```

$ gcc -g rol.S main.c
$ ./a.out
rotate_left(0x000000000000BEEF) returned 0x000000000017DDE

```

```

rotate_left(0x0000000000017DDE) returned 0x00000000002FBBC
rotate_left(0x000000000002FBBC) returned 0x00000000005F778
rotate_left(0x000000000005F778) returned 0x0000000000BEEF0
rotate_left(0x00000000000BEEF0) returned 0x00000000017DDE0
rotate_left(0x000000000017DDE0) returned 0x0000000002FBBC0
rotate_left(0x00000000002FBBC0) returned 0x0000000005F7780
rotate_left(0x00000000005F7780) returned 0x000000000BEEF00
rotate_left(0x0000000000BEEF00) returned 0x0000000017DDE00
rotate_left(0x00000000017DDE00) returned 0x000000002FBBC00
rotate_left(0x0000000002FBBC00) returned 0x000000005F77800
rotate_left(0x0000000005F77800) returned 0x00000000BEEF000
rotate_left(0x000000000BEEF000) returned 0x000000017DDE000
rotate_left(0x0000000017DDE000) returned 0x00000002FBBC000
rotate_left(0x000000002FBBC000) returned 0x00000005F778000
rotate_left(0x000000005F778000) returned 0x0000000BEEF0000
rotate_left(0x00000000BEEF0000) returned 0x00000017DDE0000
rotate_left(0x000000017DDE0000) returned 0x0000002FBBC0000
rotate_left(0x00000002FBBC0000) returned 0x0000005F7780000
rotate_left(0x00000005F7780000) returned 0x000000BEEF00000
rotate_left(0x0000000BEEF00000) returned 0x0000017DDE00000
rotate_left(0x00000017DDE00000) returned 0x000002FBBC00000
rotate_left(0x0000002FBBC00000) returned 0x000005F77800000
rotate_left(0x0000005F77800000) returned 0x00000BEEF000000
rotate_left(0x000000BEEF000000) returned 0x000017DDE000000
rotate_left(0x0000017DDE000000) returned 0x00002FBBC000000
rotate_left(0x000002FBBC000000) returned 0x00005F778000000
rotate_left(0x000005F778000000) returned 0x0000BEEF0000000
rotate_left(0x00000BEEF0000000) returned 0x00017DDE0000000
rotate_left(0x000017DDE0000000) returned 0x0002FBBC0000000
rotate_left(0x00002FBBC0000000) returned 0x0005F7780000000
rotate_left(0x00005F7780000000) returned 0x000BEEF00000000
rotate_left(0x0000BEEF00000000) returned 0x0017DDE00000000
rotate_left(0x00017DDE00000000) returned 0x002FBBC00000000
rotate_left(0x0002FBBC00000000) returned 0x005F77800000000
rotate_left(0x0005F77800000000) returned 0x00BEEF000000000
rotate_left(0x000BEEF000000000) returned 0x017DDE000000000
rotate_left(0x0017DDE000000000) returned 0x02FBBC000000000
rotate_left(0x002FBBC000000000) returned 0x05F778000000000
rotate_left(0x005F778000000000) returned 0x0BEEF00000000000
rotate_left(0x00BEEF0000000000) returned 0x17DDE0000000000
rotate_left(0x017DDE0000000000) returned 0x2FBBC0000000000
rotate_left(0x02FBBC0000000000) returned 0x5F7780000000000

```

```

rotate_left(0x5F77800000000000) returned 0xBEEF000000000000
rotate_left(0xBEEF000000000000) returned 0x7DDE000000000001
rotate_left(0x7DDE000000000001) returned 0xFBBC000000000002
rotate_left(0xFBBC000000000002) returned 0xF778000000000005
rotate_left(0xF778000000000005) returned 0xEEF000000000000B
rotate_left(0xEEF000000000000B) returned 0xDDE0000000000017
rotate_left(0xDDE0000000000017) returned 0xBBC000000000002F
rotate_left(0xBBC000000000002F) returned 0x778000000000005F
rotate_left(0x778000000000005F) returned 0xEF000000000000BE
rotate_left(0xEF000000000000BE) returned 0xDE0000000000017D
rotate_left(0xDE0000000000017D) returned 0xBC000000000002FB
rotate_left(0xBC000000000002FB) returned 0x78000000000005F7
rotate_left(0x78000000000005F7) returned 0xF000000000000BEE
rotate_left(0xF000000000000BEE) returned 0xE0000000000017DD
rotate_left(0xE0000000000017DD) returned 0xC000000000002FBB
rotate_left(0xC000000000002FBB) returned 0x8000000000005F77
rotate_left(0x8000000000005F77) returned 0x000000000000BEEF
$

```

The result shows proof positive that the rotate left was correct because the original value returned after 64 iterations.

9.5. ESP32-C3 Rotate Right

Rotating the bits to the right requires an extra step, as shown in Listing 9.5. Line 5 performs an "and immediate" operation placing a 1 or 0 into register a1. This example also illustrates that you can work with unused argument registers as temporary registers. Line 6 shifts that result in a1 up to bit 31. Then the argument value in a0 is shifted right (logical) one bit in line 7. Finally, the bit saved in a1 is or-ed into register a0 (line 8) and then returned.

Tip: When using unused argument registers as temporaries, it is recommended that you start with a7 and work downward. The reason is that if more arguments are later needed, you won't have to modify your code to reassign the registers that were in use.

```

1      .global rotate_right
2      .text
3
4 rotate_right:
5      andi    a1,a0,1      # a1=1 if bit 0 of a0 is true
6      slli    a1,a1,31     # Set bit 31 of a1 if bit=1
7      srli    a0,a0,1     # a0 >>= 1 (logical)
8      or     a0,a0,a1     # or in sign bit
9      ret

```

Listing 9.5: Program for rotate_right() function, ~/riscv/repo/09/ror/main/ror.S.

The main program is illustrated in Listing 9.6 and is otherwise much the same as before.

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 #pragma GCC optimize("-O3")
5
6 extern uint32_t rotate_right(uint32_t bits); // Assembler routine
7
8 void
9 app_main(void) {
10     uint32_t bits = 0xBEEF, ror;
11
12     for ( unsigned ux=0; ux<32; ++ux, bits = ror ) {
13         ror = rotate_right(bits);
14         printf("rotate_right(0x%08X) returned 0x%08X\n",
15             bits,ror);
16     }
17     printf("Done.\n");
18 }

```

Listing 9.6: The main program to test rotate_right(), ~/riscv/repo/09/ror/main/main.c.

Build, flash and monitor the program as usual:

```

$ cd /riscv/repo/09/ror
$ idf.py build
...
$ idf.py -p $PORT flash monitor
...
I (257) cpu_start: Starting scheduler.
rotate_right(0x0000BEEF) returned 0x80005F77
rotate_right(0x80005F77) returned 0xC0002FBB
rotate_right(0xC0002FBB) returned 0xE00017DD
rotate_right(0xE00017DD) returned 0xF0000BEE
rotate_right(0xF0000BEE) returned 0x780005F7
rotate_right(0x780005F7) returned 0xBC0002FB
rotate_right(0xBC0002FB) returned 0xDE00017D
rotate_right(0xDE00017D) returned 0xEF0000BE
rotate_right(0xEF0000BE) returned 0x7780005F
rotate_right(0x7780005F) returned 0xBBC0002F
rotate_right(0xBBC0002F) returned 0xDDE00017
rotate_right(0xDDE00017) returned 0xEEF0000B
rotate_right(0xEEF0000B) returned 0xF7780005
rotate_right(0xF7780005) returned 0xFBBC0002
rotate_right(0xFBBC0002) returned 0x7DDE0001

```

```

rotate_right(0x7DDE0001) returned 0xBEEF0000
rotate_right(0xBEEF0000) returned 0x5F778000
rotate_right(0x5F778000) returned 0x2FBBC000
rotate_right(0x2FBBC000) returned 0x17DDE000
rotate_right(0x17DDE000) returned 0x0BEEF000
rotate_right(0x0BEEF000) returned 0x05F77800
rotate_right(0x05F77800) returned 0x02FBBC00
rotate_right(0x02FBBC00) returned 0x017DDE00
rotate_right(0x017DDE00) returned 0x00BEEF00
rotate_right(0x00BEEF00) returned 0x005F7780
rotate_right(0x005F7780) returned 0x002FBBC0
rotate_right(0x002FBBC0) returned 0x0017DDE0
rotate_right(0x0017DDE0) returned 0x000BEEF0
rotate_right(0x000BEEF0) returned 0x0005F778
rotate_right(0x0005F778) returned 0x0002FBBC
rotate_right(0x0002FBBC) returned 0x00017DDE
rotate_right(0x00017DDE) returned 0x0000BEEF
Done.

```

As an exercise, create a RV64 version of the same to be run under QEMU. There is one minor change needed for `ror.S`. Can you spot it?

9.6. Pseudo Opcodes

Some additional pseudo-opcodes are available that make writing your RISC-V assembly language code easier and at the same time, easier to read.

<code>nop</code>		<code># no operation (addi x0,x0,0)</code>
<code>not</code>	<code>rd,rs</code>	<code># one's complement (xori rd,rs,-1)</code>
<code>neg</code>	<code>rd,rs</code>	<code># two's complement (sub rd,x0,rs)</code>
<code>negw</code>	<code>rd,rs</code>	<code># two's complement, word (subw rd,x0,rs RV64)</code>

The "not" opcode is the same as the familiar one's complement (`~`) operator in C. The "neg" opcode produces a two's complement value by subtracting `rs` from zero. The "negw" is for the RV64 and larger platforms to treat a 64-bit value initially as a 32-bit value and then sign extend the result.

9.7. Unsigned Multi-precision Arithmetic

How is the data type `uint64_t` supported by the C/C++ compiler on the ESP32-C3, which is RV32-based? Since there are no 64-bit register operations available, it must be done 32 bits at a time. As you know from before, there are no flag register bits like Carry. So, we must explore the RISC-V solution to these types of problems.

For unsigned data types, this proves to be straightforward. Listing 9.7 demonstrates how it is done. The low order words are added together in line 11, with the result replacing `a0`, where the low order 32 bit result is returned. Line 12 then tests if the unsigned result in `a0` is less than `a2` (operand 2), and if so, places a 1 in `a7`, else zero. This value is essentially the

carry bit. In other words, if the unsigned sum of the low order words is less than either of the low order operands, then there is a carry of 1. The program continues in line 13 to add the high order 32-bit words, replacing a1 with the result. Finally, the carry (if any) is added to the high order word in line 14. The 64-bit result is then returned in a0 and a1 at line 15.

```

1      .global add64
2      .text
3
4      # ARGUMENTS:
5      #      a0, a1  uint64_t operand 1
6      #      a2, a3  uint64_t operand 2
7      #
8      # RETURNS:
9      #      a0, a1  uint64_t sum
10     #
11     add64: add    a0,a0,a2      # Add low order 32 bits
12         sltu   a7,a0,a2      # a7 = a0 < a2 ? 1 : 0
13         add    a1,a1,a3      # Add high order 32 bits
14         add    a1,a1,a7      # Add carry
15         ret

```

Listing 9.7: Multi-precision unsigned program ~/riscv/repo/09/multipu/main/multipu.S.

The main driver program for this test is shown in Listing 9.8. The variables x, y and z are marked volatile in this program to prevent the optimizing C compiler from pre-calculating the result.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  #pragma GCC optimize("-O3")
5
6  extern uint64_t add64(uint64_t op1,uint64_t op2);
7
8  void
9  app_main(void) {
10     volatile uint64_t x=0x7FFFFFFF, y=0x3000011115, z;
11
12     z = add64(x,y);
13     printf("0x%016llx + 0x%016llx = 0x%016llx\n",x,y,z);
14 }

```

Listing 9.8: Main driver program ~/riscv/repo/09/multipu/main.c.

Build, flash and monitor the program to see the results:

```
$ cd ~/riscv/repo/09/multipu
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
0x00000007FFFFFFFF + 0x0000003000011115 = 0x0000003800011114
```

If you plug those values into gdb, you can verify the result:

```
$ gdb
GNU gdb (GDB) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
...
(gdb) p /x 0x7FFFFFFFFfll + 0x3000011115ll
$1 = 0x3800011114
(gdb)
```

Unsigned Overflow

Testing for unsigned overflow is equally simple, since all that is required is to see if a carry occurred out of the high order word.

9.8. Signed Multi-precision Arithmetic

Signed arithmetic handles the carry precisely the same way. When adding the low order words, simply test if the result of the sum is less than either of the operand words. If so, then a carry is needed.

9.8.1. Signed Overflow

Testing for overflow of a signed value is a little more involved than a carry. Listing 9.9 illustrates an example function testing for overflow after addition. The general function `s32ovf` returns 1 if the addition of two `int32_t` values results in an overflow, else a zero.

```
1      .global s32ovf
2      .text
3
4 # ARGUMENTS:
5 #     a0     int32_t operand 1
6 #     a1     int32_t operand 2
7 #
8 # RETURNS:
9 #     a0     flag:
10 #           0 = no overflow
11 #           1 = overflow
12 #
```

```

13 s32ovf: slti    t1,a0,0      # t1 = a0 < 0 ? 1 : 0
14         slti    t2,a1,0      # t2 = a1 < 0 ? 1 : 0
15         bne     t1,t2,noovfl  # Signs differ: no overflow
16
17 #       Signs equal
18
19         add     t0,a0,a1      # t0 = a0 + a1 (sum)
20         slti    t0,t0,0      # t0 = t0 < 0 ? 1 : 0
21         bne     t0,t1,ovfl    # Branch if overflowed
22
23 noovfl: li     a0,0          # No overflow
24         ret
25
26 ovfl:   li     a0,1          # Overflow
27         ret

```

Listing 9.9: Program ~/riscv/repo/09/multips/main/overflow.S.

The overflow test procedure is summarized as follows:

1. The input values to be summed are passed into the function in registers a0 and a1 according to the GNU calling convention.
2. Line 13 sets t1 to 1 if the first operand is negative, else zero.
3. Line 14 sets t2 to 1 if the second operand is negative, else zero.
4. A branch to noovfl is taken in line 15 if the signs differ (no overflow is possible when the signs differ).
5. A trial sum is made of the two operands into t0 to determine the sign of the result (line 19).
6. Register t0 is set to 1 if the trial sum is negative, else zero (line 20).
7. If t0 from step 6 does not match t1 from step 2, then the result's sign has changed, indicating an overflow. In the overflow case, a branch is made to label ovfl (line 21).
8. When there is no overflow from step 7, fall through and return zero (line 23).

In this example, we only return the overflow status of the sum. An improved approach would be to return both the sum and the status together, but that would be getting ahead of ourselves.

The main driver program is provided in Listing 9.10. Four tests are performed by calling the static function report() (lines 8 to 15). Each test is invoked with different test values in lines 20 to 23. The assembly language function is invoked from line 12, and the results are reported in lines 13 and 14.

```

1 #include <stdio.h>
2 #include <stdint.h>
3

```

```

4 #pragma GCC optimize("-O3")
5
6 extern int32_t s32ovf(int32_t op1,int32_t op2);
7
8 static void
9 report(int32_t x,int32_t y) {
10     int z;
11
12     z = s32ovf(x,y);
13     printf("0x%08X (%d) + 0x%08X (%d) = 0x%08X (%d): %soverflow\n",
14           x,x,y,y,x+y,x+y,z?"":"no ");
15 }
16
17 void
18 app_main(void) {
19
20     report(0x7FFFFFFE,1);
21     report(0x7FFFFFFE,46);
22     report(-3,46);
23     report(-3,0x80000000);
24 }

```

Listing 9.10: Main driver program ~/riscv/repo/09/ovflow/main/main.c.

Build, flash and monitor the run as follows:

```

$ cd ~/riscv/repo/09/ovflow
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (258) cpu_start: Starting scheduler.
0x7FFFFFFE (2147483646) + 0x00000001 (1) = 0x7FFFFFFF (2147483647): no overflow
0x7FFFFFFE (2147483646) + 0x0000002E (46) = 0x8000002C (-2147483604): overflow
0xFFFFFFF (-3) + 0x0000002E (46) = 0x0000002B (43): no overflow
0xFFFFFFF (-3) + 0x80000000 (-2147483648) = 0xFFFFFFF (2147483645): overflow

```

In the output, the first example retains the original sign after the sum (result 0x7FFFFFFF). However, in the second sum, we see that the sign changed (result 0x8000002C) and the function `s32ovf()` correctly identifies it as an overflow. The third test had inputs of different signs and the result remains ok. The final test results in an overflow, which is correctly identified.

RV64 Signed Overflow

Let's improve our assembly language function `s32ovf()` as function `addi64()` for RV64 and return the sum *and* the overflow indication together. In addition, let's also provide function

`neg64()` since negating a value can also lead to overflow. For example, a 16-bit signed integer has a maximum range of -32768 to +32767. When the value -32768 is negated, the +32768 value cannot be represented. This counts as an overflow.

The function prototypes for `addi64()` and `neg64()` are as follows:

```
extern int64_t addi64(int64_t op1,int64_t op2,bool *pbool);
extern int64_t neg64(int64_t op1,bool *pbool);
```

In these versions of the functions, we return an overflow indication by use of the pointer argument `pbool`. When the operation for `addi64()` or `neg64()` results in an overflow, the `bool` value pointed to by the `pbool` pointer argument is set to `true`. When no overflow occurs, the value is set to `false`.

Let's examine the `addi64()` function first:

1. Register `t5` is initialized to zero in line 15. This register is assuming the role of the overflow indicator.
2. Register `t2` is set to 1 if the first argument (in `a0`) is negative (line 16).
3. Register `t3` is set to 1 if the second argument (in `a1`) is negative (line 17).
4. Register `a0` is set to the result of adding the arguments (`a0 + a1`, in line 18).
5. If the signs of the arguments differ in line 19, the branch to "doret" is taken, since no overflow is possible when the signs differ.
6. Otherwise, control continues to line 23 where `t3` is set to the sign of the result (in `a0`). Register `t3` assumes the value 1, if the result is negative else is zero.
7. If the sign of the result (`t3`) matches the sign of the first argument (originally in `a0`), then no overflow has occurred, and the branch to "doret" is taken.
8. Otherwise, control passes to line 28 where `t5` is set to 1 (`true`).
9. Control then passes to the label of "doret", where the current boolean value in register `t5` is then stored to the caller's `bool`, by means of the pointer in register `a2` (argument 3).
10. Finally, control returns to the caller in line 30.

An important feature of this calling convention is that it is thread-safe. The caller supplies its own `bool` variable to be updated, which is local to the calling thread. This allows several threads to be simultaneously calling `addi64()` and yet each caller receives its own private flag value for overflow.

```
1      .global addi64,neg64
2      .text
3
4 # ARGUMENTS:
5 #      a0      int64_t operand 1
6 #      a1      int64_t operand 2
7 #      a2      ptr to bool_t
8 # RETURNS:
```

```

 9 #      a0      int64_t sum
10 #      bool_t return values:
11 #          0      no overflow
12 #          1      overflow
13
14 addi64:
15     li      t5,0          # Flag = false
16     slti   t2,a0,0       # t2 = a0 < 0 ? 1 : 0 (sign 1)
17     slti   t3,a1,0       # t3 = a1 < 0 ? 1 : 0 (sign 2)
18     add    a0,a0,a1      # Sum opr 1 + opr 2
19     bne    t2,t3,doret   # No overflow possible: signs differ
20
21 #      Signs equal: test result sign
22
23     slti   t3,a0,0       # t3 = sum < 0 ? 1 : 0
24     beq    t3,t2,doret   # Signs equal: no overflow
25
26 #      Overflowed
27
28     li      t5,1          # flag = true
29 doret: sb    t5,0(a2)     # Update bool value by ptr
30     ret
31
32 # ARGUMENTS:
33 #      a0      int64_t operand 1
34 #      a1      ptr to bool_t
35 # RETURNS:
36 #      a0      int64_t sum
37 #      bool_t return values:
38 #          0      no overflow
39 #          1      overflow
40
41 neg64: li    t5,0          # Flag = false
42     li    t4,1          # t4 = 1
43     slli  t3,t4,63      # t3 = max negative value
44     bne  a0,t3,noprob   # Branch if arg is not maximally negative
45     sb   t4,0(a1)       # *ptr = true (overflow)
46     ret
47
48 noprob: neg  a0,a0       # Negate argument (a0)
49     sb   t5,0(a1)       # *ptr = false (no overflow)
50     ret

```

Listing 9.11: Assembler functions `addi64()` and `neg64()` in `~/riscv/repo/09/ovf64/qemu64/addi64.S`.

The `neg64()` function works on a similar principle:

1. Register `t5` assumes the role of the overflow flag and is initialized `false` (line 41).
2. Register `t4` is loaded with the immediate constant of `1` (line 42).
3. Register `t3` is then set to the most negative number possible by shifting the value of `1` (in `t4`) up by 63 bits.
4. The argument (`a0`) is then compared to `t3` and if not equal control transfers to "no-prob". Argument values other than the most negative value can be safely negated without an overflow (at step 6).
5. When the branch is not taken in step 4, the value of `t4` (still holding the value `1/True`) is then saved by pointer (in `a1`) to the caller's bool variable, and control returns to the caller (line 46).
6. Otherwise, at line 48 (label "noprob"), the argument is safely negated (in `a0`), and the value `0/False` (in `t5`) is stored by pointer argument (`a1`) in the caller's bool variable, prior to returning in line 50.

The main driver program is shown in Listing 9.12. Static functions `report()` and `negtest()` test out the assembly language routines `addi64()` and `neg64()` respectively. Lines 34 and 35 test out the `addi64()` routine, which uses the thread-safe calling format. The `addi64()` call is made in line 13, passing in the two operands and a pointer to the bool variable declared in line 10. It is best practice to initialize values like "overflow", but the function `addi64()` was designed to set the variable regardless of the overflow result. The overflow value is reported by `printf()` in lines 14 and 15.

Likewise, the `neg64()` function is invoked at line 23, returning the overflow status declared at line 20, by a pointer in the call. Lines 24 and 25 report the result of the call.

```

1  #include <stdio.h>
2  #include <stdint.h>
3  #include <stdbool.h>
4
5  extern int64_t addi64(int64_t op1,int64_t op2,bool *pbool);
6  extern int64_t neg64(int64_t op1,bool *pbool);
7
8  static void
9  report(int64_t op1,int64_t op2) {
10     bool overflow;
11     int64_t r;
12
13     r = addi64(op1,op2,&overflow);
14     printf("0x%016llx + 0x%016llx = 0x%016llx, overflow=%d\n",
15           op1,op2,r,overflow);
16 }
17
18 void
19 negtest(int64_t op) {

```

```

20     bool overflow;
21     int64_t r;
22
23     r = neg64(op,&overflow);
24     printf("- 0x%016llx = 0x%016llx, overflow=%d\n",
25           op,r,overflow);
26
27 }
28
29 int
30 main(int argc,char **argv) {
31     int64_t x, y;
32     int64_t r;
33
34     report(0x4EEEEEEE99999999ll,0x2AAAAAAAAABBBBBBll);
35     report(0x4EEEEEEE99999999ll,0x7AAAAAAAAABBBBBBll);
36
37     negtest(-45609);
38     negtest(45609);
39     negtest((1ll << 63) + 1);
40     negtest(1ll << 63);
41
42     return 0;
43 }

```

Listing 9.12: Main driver program ~/riscv/repo/09/ovf64/qemu64/main.c.

Compile and run the program in Fedora Linux as follows:

```

$ cd ~/riscv/repo/09/ovf64/qemu64
$ gcc -g addi64.S main.c
$ ./a.out
0x4EEEEEEE99999999 + 0x2AAAAAAAAABBBBBB = 0x7999999945555554, overflow=0
0x4EEEEEEE99999999 + 0x7AAAAAAAAABBBBBB = 0xC999999945555554, overflow=1
- 0xFFFFFFFFFFFF4DD7 = 0x000000000000B229, overflow=0
- 0x000000000000B229 = 0xFFFFFFFFFFFF4DD7, overflow=0
- 0x8000000000000001 = 0x7FFFFFFFFFFFFFFF, overflow=0
- 0x8000000000000000 = 0x8000000000000000, overflow=1
$

```

We see that the `addi64()` function performs the 64-bit addition and returns the correct overflow status for the two tests (first two lines of output). The last four lines of output are the result of testing `neg64()`. From the output shown, it is evident that the results of these tests are correct and that the one overflow case was identified.

Questions emerge when designing functions like `addi64()` and `neg64()`. When the overflow occurs for `neg64()` in this program, we chose to return the input argument unchanged. Some applications might prefer to return the value `0x7FFFFFFFFFFFFFFF` instead, which is only off by one. On the other hand, this is still not a correct result.

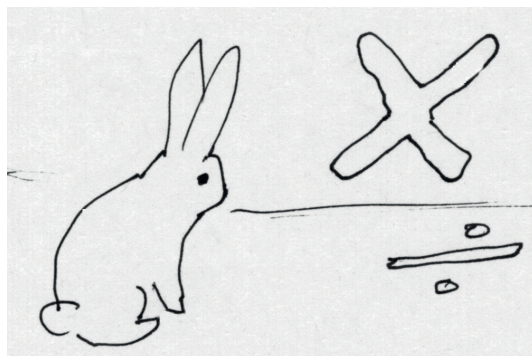
For `addi64()`, the incomplete sum was returned instead when an overflow occurred. If the application can tolerate the overflow (as wraparound), then perhaps this is acceptable. Here, it is assumed that the caller will take the appropriate action when the overflow indicator is set to true.

9.9. Summary

At this point in the book, you should feel good about your progress in the RISC-V assembly language. You have covered most of the RISC-V instruction set opcodes that you'll normally use. There are some others that are privileged or used in special situations, which will be left for the advanced students to study on their own.

The next chapter covers multiply and divide opcodes. These are available when extension M for RISC-V is provided. This M extension is included for the ESP32C3 device (RV32) and for QEMU (RV64I) used by our Fedora Linux. With the exception of floating point and multiply/divide our treatment of opcodes is complete. The beauty of RISC-V is not having to memorize volumes of opcodes!

Chapter 10 • Multiply / Divide



Multiply like a rabbit with extension M

Integer multiplication and division are essential mathematic operations. Yet RISC-V defines these as an extension M, which permits vendors to create CPUs without this capability to save on cost. These operations can, of course, be performed in software but at the expense of CPU time. Given the clear advantages of these hardware operations, devices like the ESP32-C3 do, however, provide for extension M. These added opcodes boost the overall performance of the CPU.

10.1. Multiplication Operations

When you multiply a 32-bit value (multiplier) by another 32-bit value (multiplicand), you obtain a result (product) that is represented by up to 32+32 bits in length. Because of this, some architectures provide one opcode for producing a product in a register pair. RISC-V takes a different approach. It defines opcodes to produce only the low order word, or another to return the high order product word. However, as will be revealed, RISC-V does permit an optimization that exploits the hardware to avoid having to perform the multiplication operation twice.

Signed Multiplication

Multiplication can be performed as unsigned or signed integer operations. Unsigned multiplication is often used for C language subscripting operations within an array or matrix, where the first element is at subscript 0. Other languages such as Ada permit subscripts to use negative numbers, and thus must use signed multiplication.

When multiplying signed numbers, the product's sign obeys the following relationships:

- a positive x positive → positive
- a positive x negative → negative
- a negative x positive → negative
- a negative x negative → positive

10.2. Division Operations

When dividing a 32-bit number (dividend) by another 32-bit number (divisor), it produces a result (quotient) up to 32 bits in length. Dividing by 1 will do, this for example. Additionally, when dividing integers, you may need the remainder. So, a division may be expected to produce both the quotient and remainder. Some architectures provide one opcode that produces the result in a pair of registers. Like multiplication, RISC-V takes a different approach, providing for an optimization that silicon can exploit.

In addition to the above characteristics, one has to be careful about the semantics of a divide by zero and the possibility of overflow.

Signed Division

When dividing signed numbers, the quotient's and remainder's sign obeys the following rules:

- a positive \div positive \rightarrow positive quotient, positive remainder
- a positive \div negative \rightarrow negative quotient, positive remainder
- a negative \div positive \rightarrow negative quotient, positive remainder
- a negative \div negative \rightarrow positive quotient, negative remainder

It should be noted that the C language compiler "/" and "%" operators follow these conventions.

10.3. Opcode mul/mulu

The RISC-V integer multiplication opcode is "mul" or "mulu" for signed and unsigned respectively:

```
mul    rd,rs1,rs2    # rd = lower(rs1 x rs2), signed
mulu   rd,rs1,rs2    # rd = lower(rs1 x rs2), unsigned
```

The destination register receives the lower XLEN bits of the product. To obtain the upper XLEN bits of the product, the "mulhs" (signed) or "mulhu" (unsigned) opcodes are available.

10.4. Opcode mulhs/mulhu

To obtain the upper XLEN bits of the product, use the "mulh" or "mulhu" opcodes for signed and unsigned respectively:

```
mulhs  rd,rs1,rs2    # rd = upper(rs1 x rs2), signed
mulhu  rd,rs1,rs2    # rd = upper(rs1 x rs2), unsigned
```

10.5. Optimized Multiply

Now let's reveal the RISC-V trick for using the optimized full multiply. To get the full XLEN+XLEN bit product, use the following back-to-back instructions for signed multiply, in this specific sequence:

```
mulhs    rdh,rs1,rs2    # rdh = high order product word
mul      rd1,rs1,rs2    # rd1 = low order product word
```

The destination register `rdh` receives the high order product word while `rd1` receives the low order product word. Register `rdh` cannot be the same as `rs1` or `rs2` for the optimized operation. For unsigned products use the following sequence instead:

```
mulhu    rdh,rs1,rs2    # rdh = high order product word
mulu     rd1,rs1,rs2    # rd1 = low order product word
```

For the multiply to function optimally, the following rules must be observed:

1. The `mulhs` or `mulhu` opcode must occur first.
2. The `mul` or `mulu` opcode must *immediately* follow.
3. Source registers `rs1` and `rs2` must be provided *in the same order* for both instructions.
4. And the destination register *rdh cannot be rs1 or rs2*.

Multiplication is a relatively expensive operation. When the silicon is designed for it, the first of the pair of opcodes can evaluate the XLEN-bit+XLEN-bit product internally but stores only the high order half of the product in the destination register `rdh`. When the second opcode of the pair is processed and the rules above are obeyed, then the CPU can store the lower half of the *precomputed* product in the current destination register. In this optimization, the only added overhead is the fetch and decode of the second instruction in the sequence. A failure to observe the above rules results in two completely separate multiplication operations being computed, along with added execution time. Figure 10.1 shows a creature that simply loves to multiply.

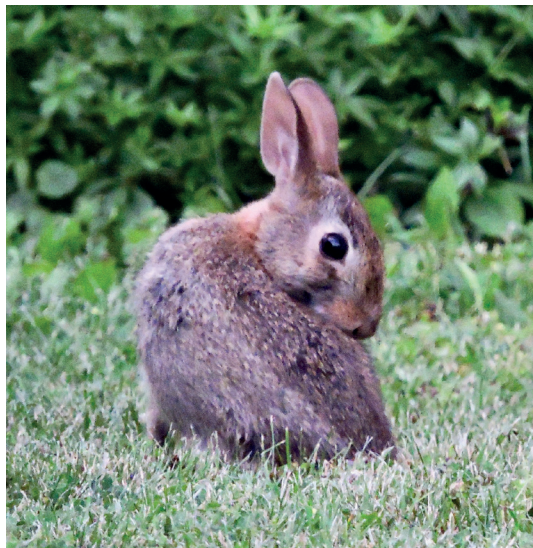


Figure 10.1 Canadian rabbits love to multiply.

10.6. Unsigned Factorial

As a fun little exercise for the ESP32-C3 device, let's compute an unsigned factorial value of $x!$ The `ufact32()` function computes the factorial without checking for overflow but is otherwise, designed to be as frugal as possible. The program is illustrated in Listing 10.1.

```

1      .global ufact32
2      .text
3
4  # ARGUMENTS:
5  #      a0      uint32_t x
6  #
7  # RETURNS:
8  #      a0      uint32_t factorial x!
9  #
10
11 ufact32:li    t1,1          # t1 = 1
12      mv      t0,a0
13
14 1:      addi   t0,t0,-1      # t0 = a0 - 1
15      ble    t0,t1,2f      # Branch if t0 <= 1
16      mul    a0,a0,t0      # a0 *= t0
17      j      1b           # Loop until t1 <= 1
18
19 2:      ret

```

Listing 10.1: Unsigned factorial in `~/riscv/repo/10/factorial/main/factorial.S`.

The function operates as follows:

1. Upon entry to the function, the constant 1 is loaded into temporary register 1, for use in comparisons (line 11).
2. The argument x is copied to temporary register `t0` (line 12).
3. At the top of the loop in line 14, the value in `t0` is decremented.
4. A branch is taken from line 15 if the value of `t0` is less than or equal to 1. The branch when taken, passes control to the return instruction in line 19.
5. The value that's currently in `a0` (initially x) is multiplied by register `t0` (initially $x-1$) at line 16. The result replaces `a0`.
6. The control then passes from line 17 to the top of the loop at line 14.

The main program to test this function is illustrated in Listing 10.2.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern uint32_t ufact32(uint32_t x);
5

```

```

6 void
7 app_main(void) {
8
9     for ( unsigned ux=0; ux<=12; ++ux ) {
10         uint32_t f = ufact32(ux);
11
12         printf("%2u ! => %u 0x%08X\n",ux,f,f);
13     }
14     printf("Done.\n");
15 }

```

Listing 10.2: Main program for ufact32() in file ~/riscv/repo/10/factorial/main/main.c.

Build and exercise the program as follows:

```

$ cd ~/riscv/repo/10/factorial
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
0 ! => 0 0x00000000
1 ! => 1 0x00000001
2 ! => 2 0x00000002
3 ! => 6 0x00000006
4 ! => 24 0x00000018
5 ! => 120 0x00000078
6 ! => 720 0x000002D0
7 ! => 5040 0x000013B0
8 ! => 40320 0x00009D80
9 ! => 362880 0x00058980
10 ! => 3628800 0x00375F00
11 ! => 39916800 0x02611500
12 ! => 479001600 0x1C8CFC00
Done.

```

The hexadecimal value was printed on the right so that you can visualize the limit of this calculation (eight hexadecimal digits). If you were to continue further, the result would overflow and be incorrect. Under Linux/Mac, you can check this using the Linux `bc` command:

```

$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.

```



```

39916800 * 12
479001600
479001600 * 13
6227020800

```

The 12! result is indeed 479001600, but 13! overflows 32-bits to the value 6227020800 (0x17328CC00).

10.7. Opcode `div/divu`

Division is even more costly to evaluate than multiplication. It is therefore beneficial to have it included in the M extension. The following opcodes provide an operation that divides an XLEN-bit dividend value by an XLEN-bit divisor. Two opcodes provide for signed and unsigned division:

```

div    rd,rs1,rs2    # rd = rs1 / rs2 (signed)
divu   rd,rs1,rs2    # rd = rs1 / rs2 (unsigned)

```

Notice that even though a product can be XLEN+XLEN bits in size, there is no provision to divide an integer of that size. Only XLEN-bits can be hardware divided by extension M.

Opcode `rem/remu`

In addition to division, a remainder is often required. The "rem" and "remu" opcodes provide the remainder after dividing an XLEN-bit dividend value by an XLEN-bit divisor, for signed and unsigned numbers respectively:

```

rem    rd,rs1,rs2    # rd = rs1 % rs2 (signed)
remu   rd,rs1,rs2    # rd = rs1 % rs2 (unsigned)

```

10.8. Optimized Divide

Algorithms frequently require both the quotient and the remainder in a calculation. So why expend extra time performing the operation twice? Like the optimized multiply operation, RISC-V permits an optimized divide if the programmer issues the instructions as a sequence in a specific order. The followings provide for signed division and remainder as one hardware operation:

```

div    rdq,rs1,rs2    # rdq = rs1 / rs2
rem    rdr,rs1,rs2    # rdr = rs1 % rs2

```

For unsigned use:

```

divu   rdq,rs1,rs2    # rdq = rs1 / rs2
remu   rdr,rs1,rs2    # rdr = rs1 % rs2

```

For the division operation to function optimally, the following rules must be observed:

1. The `div` or `divu` opcode must occur first.
2. The `rem` or `remu` opcode must immediately follow.
3. Source registers `rs1` and `rs2` must be provided in the same order for both instructions.
4. And the destination register `rdq` cannot be the same as `rs1` or `rs2`.

10.9 Division By Zero

Division by zero has that mathematical quirk that there is no defined answer. Some architectures raise an exception when this is attempted. The designers of RISC-V felt it best to simply specify the result of such an operation instead since it is easy to test for this condition. For signed division by zero, the following results apply:

- $x \div 0 \rightarrow -1$
- $x \% 0 \rightarrow 0$

For unsigned division, the following results apply:

- $x \div 0 \rightarrow 2^{XLEN} - 1$
- $x \% 0 \rightarrow x$

If there is a possibility of dividing by zero, it is best to test the divisor prior to performing the division or remainder.

10.10 Divide Overflow

Division of *signed* integer numbers also has the quirk that an overflow is possible in one specific case. Like the negate operation, the signed divide overflows when the most negative number of `XLEN` bits is divided by a negative one. RISC-V defines the following results for overflow:

- $2^{XLEN} / -1 \rightarrow 2^{XLEN} - 1$
- $2^{XLEN} - 1 \bmod -1 \rightarrow 0$

In other words, dividing the most negative number by a negative one produces a quotient that cannot be represented as a positive number, with a remainder of zero. If this is a possibility, an extra test should be included when dividing it by a negative one. Then if the dividend is the most negative number, you know that you have an overflow on your hands.

10.11 Safe Division

Now let's put in code what you've learned so far about division. Listing 10.3 illustrates the function `safediv()`, which returns both quotient and remainder, as well as flags indicating a divide-by-zero and overflow conditions.

```

1      .global safediv
2      .text
3
4  # ARGUMENTS:
5  #      a0      signed dividend
6  #      a1      signed divisor
7  #      a2      pointer to remainder
8  #      a3      pointer to bool (div by zero)
9  #      a4      pointer to bool (overflow)
10 #
11 # RETURNS:
12 #      a0      quotient
13 #      remainder by pointer
14 #      div by zero flag by pointer
15 #      overflow flag by pointer
16 #
17
18 safediv:
19     mv      t3,a0          # t3 = dividend
20     mv      t4,a1          # t4 = divisor
21
22     li      t5,0           # t5 = true when overflow
23     li      t6,0           # t6 = true if divisor zero
24     bnez    t4,nzero
25     li      t6,1           # t6 = true (div by zero)
26
27 nzero: div    a0,t3,t4      # a0 = dividend / divisor
28     rem    a1,t3,t4      # a1 = dividend % divisor
29     sw     a1,0(a2)      # Return remainder
30
31     li      t2,-1
32     bne    t4,t2,noovf    # Divisor != -1 => no overflow
33
34     slli   t2,t2,31       # t2 now maximally -ve
35     bne    t2,t3,noovf    # Branch if dividend not max -ve
36     li      t5,1         # Else set overflow t5 = true
37
38 noovf: sb     t5,0(a4)     # Return overflow flag
39     sb     t6,0(a3)     # Return div by zero flag
40     ret

```

Listing 10.3: The `safediv()` function in file `~/riscv/repo/10/safediv/main/safediv.S`.

The operation of this function breaks down as follows:

1. The dividend and divisor arguments are copied to temporary registers t3 and t4 respectively (lines 19 and 20).
2. The temporary register t5 is initialized false as the flag for overflow (line 22).
3. Temporary register t6 is initialized false as the flag for divide-by-zero (line 23).
4. The divisor is tested for zero in line 24. If it is *not* zero, a branch is taken to nzero (step 6).
5. Otherwise, the flag in t6 is set to true (line 25) to indicate divide-by-zero.
6. The division is performed in line 27, placing the result into register a0, which is the return value (line 27). The values divided are in temporaries t3 and t4.
7. The remainder, which is computed at step 6, is placed into register a1 at line 28. This is an optimized operation since it obeys the optimized operation rules.
8. The remainder is stored by pointer back to the caller's variable in line 29 (the pointer is in register a2).
9. Temporary register t2 is loaded with a value of -1 (line 31).
10. In line 32, if the divisor (in t4) is not equal to -1 (in t2), then a branch is made to label "noovf", since no overflow is possible.
11. Otherwise, in line 34, the value in t2 is shifted left logical for 31 bits, creating the most negative 32-bit value in t2.
12. If the dividend (in t3) is not equal to the most negative number (in t2), a branch is made to label "noovf" (line 35).
13. Otherwise, we set temporary register t5, holding the overflow flag to true (line 36).
14. At the label "noovf", we store both flags – register t5 to the caller's overflow flag (pointer in a4), and register t6 to the caller's divide-by-zero flag (pointer in a3), in lines 38 and 39.
15. Return to the caller in line 40.

The driver program main.c, is provided in Listing 10.4. The loop in lines 30 to 45 call the function safediv() with six test cases, while reporting the results.

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <stdbool.h>
4
5 extern int32_t safediv(
6     int32_t dividend,
7     int32_t divisor,
8     int32_t *remainder,
9     bool *divbyzero,
10    bool *overflow
11 );
12
13 struct s_div {
14     int32_t dividend;
15     int32_t divisor;
```

```

16  };
17
18  void
19  app_main(void) {
20      static struct s_div const tests[] = {
21          { 23, 3 },
22          { -23, 3 },
23          { 46, 0 },
24          { 0x80000000, -2 },
25          { 0x80000000, -1 },
26          { 0x80000000, 15 },
27          { 0, 0 }
28      };
29
30      for ( unsigned ux=0;
31          tests[ux].dividend || tests[ux].divisor;
32          ++ux ) {
33          int32_t dividend = tests[ux].dividend;
34          int32_t divisor = tests[ux].divisor;
35          int32_t quotient, remainder;
36          bool divbyzero, overflow;
37
38          quotient = safediv(dividend,divisor,
39                          &remainder,&divbyzero,&overflow);
40
41          printf("%d / %d => %d remainder %d; "
42                "divbyzero=%d, overflow=%d\n",
43                dividend, divisor, quotient, remainder,
44                divbyzero, overflow);
45      }
46      printf("Done.\n");
47  }

```

Listing 10.4: Main program ~/riscv/repo/10/safediv/main/main.c.

Build, flash and run the program on your ESP32-C3 as follows:

```

$ cd ~/riscv/repo/10/safediv
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
23 / 3 => 7 remainder 2; divbyzero=0, overflow=0
-23 / 3 => -7 remainder -2; divbyzero=0, overflow=0
46 / 0 => -1 remainder 46; divbyzero=1, overflow=0

```

```
-2147483648 / -2 => 1073741824 remainder 0; divbyzero=0, overflow=0
-2147483648 / -1 => -2147483648 remainder 0; divbyzero=0, overflow=1
-2147483648 / 15 => -143165576 remainder -8; divbyzero=0, overflow=0
Done.
```

In the resulting output, the first two lines report valid quotient and remainders with no divide-by-zero or overflow flags set. The third line does, however, report a divide-by-zero failure (and also confirms the quotient result of -1 and a remainder of 46, which was the dividend).

The second to last line reported an overflow. This is correct because a maximally negative 32-bit number (-2147483648) divided by -1, should be the positive value +2147483648, which cannot be represented by a signed 32-bit number.

10.12. Greatest Common Divisor

To further test our knowledge of division, let's implement a function named `gcd64()`, which will calculate the greatest common divisor between two integers. Listing 10.5 contains the function for use under Fedora Linux (QEMU).

The function is based upon the following algorithm:

1. If $a = 0$ in $\text{GCD}(a,b)$ then return b ($\text{GCD}(0,b) = b$).
2. If $b = 0$ in $\text{GCD}(a,b)$ then return a ($\text{GCD}(a,0) = a$).
3. If $b > a$ then swap a and b .
4. Otherwise, we compute $r = a \% b$, to satisfy the equation $a = b \times q + r$.
5. Set $a = b$ and $b = r$ to effectively call $\text{GCD}(b,r)$.
6. Repeat from step 1.

```
1      .global gcd64
2      .text
3
4      # ARGUMENTS:
5      #      a0      Number a
6      #      a1      Number b
7      #
8      # RETURNS:
9      #      a0      Returned GCD(a,b)
10
11     gcd64: bge      a0,a1,1f      # Branch if a >= b
12
13     #      swap a0 and a1
14
15     mv      t0,a0
16     mv      a0,a1
17     mv      a1,t0
18
```

```

19 1:      beqz   a0,retb      # If a == 0 return b
20      beqz   a1,reta      # If b == 0 return a
21
22 #      Compute r such that a = b x q + r
23
24      rem    t1,a0,a1      # t1 (r) = a % b
25
26 #      GCD(b,r)
27
28      mv     a0,a1         # a = b
29      mv     a1,t1         # b = r
30      j      gcd64
31
32 retb:   mv    a0,a1       # Return b
33 reta:   ret

```

Listing 10.5: The gcd64() function in ~/riscv/repo/10/gcd/qemu64/gcd64.S.

Now let's see how that mapped out to RISC-V in the program gcd64.S:

1. Make sure $a \geq b$ in line 11, branching to 1f when true.
2. Otherwise, swap arguments a and b on lines 15 to 17.
3. If a in a0 equals zero, branch to retb at line 19 (step 8).
4. if b in a1 equals zero, branch to rega at line 20 (step 9).
5. Compute the remainder by dividing a0 (a) by a1 (b), with the result going to temporary register t0 (line 24).
6. Set a0 (a) to the value b in register a1 (line 28) and then set a1 (b) to the value r in register t1 (line 29).
7. Repeat step 1 (line 30).
8. Label "retb" (line 32) returns the value of b (in register a0).
9. Label "reta" (line 33) returns the value of a (in register a0) when branched to from line 20. Otherwise, when execution continues from line 32, then the value of b is returned instead.

The main program for this test harness is provided in Listing 10.6.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern int64_t gcd64(int64_t a,int64_t b);
5
6  struct s_test {
7      int64_t a;
8      int64_t b;
9  };
10

```

```

11 int
12 main(int argc,char **argv) {
13     static struct s_test const tests[] = {
14         { 12, 10 },
15         { 51, 21 },
16         { 31 * 3, 31 * 7 },
17         { 211*5, 211 },
18         { 0, 0 }
19     };
20
21     for ( unsigned ux=0; tests[ux].a && tests[ux].b; ++ux ) {
22         int64_t g = gcd64(tests[ux].a,tests[ux].b);
23         printf("gcd(%d,%d) => %d\n",
24             tests[ux].a,tests[ux].b,g);
25     }
26     return 0;
27 }

```

Listing 10.6: Main driver program ~/riscv/repo/10/qemu64/main.c.

The loop on lines 21 to 25 uses the function with different test values. The result is printed on lines 23 and 24. Compile and run this program under Fedora Linux as follows:

```

$ cd ~/riscv/repo/10/gcd/qemu64
$ gcc -g gcd64.S main.c
$ ./a.out
gcd(12,10) => 2
gcd(51,21) => 3
gcd(93,217) => 31
gcd(1055,211) => 211

```

10.13. Combinations

To capitalize on the many things we've learned so far in this book, let's try our hand on an RV64 project that is a little more involved. Let's compute a combination $C(n,r)$ function such that out of n objects, determine how many unique samples of size r , can be obtained. This is calculated with the formula:

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

The listing for the assembly language `nCr()` function is provided in Listing 10.7.

```

1     .global nCr    # C(n,r) calculation
2     .text
3

```



```

4 # ARGUMENTS:
5 #     a0     uint64_t n of C(n,r)
6 #     a1     uint64_t r of C(n,r)
7 #
8 # RETURNS:
9 #     a0     uint64_t C(n,r) = n! / (r! * (n - r)!)
10
11 nCr:  mv     t5,a0     # Save t5 = n
12      mv     t4,a1     # Save t4 = r
13      sub    t6,t5,t4   # Save t6 = (n - r)
14
15      jal    t0,fact    # n! (in a0)
16      mv     t5,a0     # t5 = n!
17
18      mv     a0,t4     # a0 = r
19      jal    t0,fact    # r!
20      mv     t4,a0     # t4 = r!
21
22      mv     a0,t6     # a0 = (n - r)
23      jal    t0,fact    # (n - r)!
24      mv     t6,a0     # t6 = (n - r)!
25
26      mul    t3,t4,t6   # t3 = r! * (n-r)!
27      div    a0,t5,t3   # a0 = n! / t3
28      ret
29
30 #
31 #     Internal factorial routine
32 #
33 fact: li     t1,1     # t1 = 1
34      mv     t2,a0     # t2 = n
35
36 1:    addi   t2,t2,-1   # t2 = a0 - 1
37      ble    t2,t1,2f   # Branch if t2 <= 1
38      mul    a0,a0,t2   # a0 *= t2
39      j      1b         # Loop until t1 <= 1
40
41 2:    jr     t0         # Internal return via t0

```

Listing 10.7: The `nCr()` function found in file `~/riscv/repo/10/nCr/qemu64/ncr.S`.

From the formula, it is seen that the factorial is needed in three places. Rather than invoking the overhead of saving and restoring `t0` from the stack, the internal function `fact()`, starting in line 33, uses temporary register `t0` as the linkage register (the return occurs in line 41, through `t0`). Otherwise, this internal routine is much the same as the factorial function that we've seen before.

Let's break the rest of this procedure down:

1. Argument n , arriving in $a0$, is copied to temporary register $t5$ (line 11).
2. Argument r , arriving in $a1$, is copied to temporary register $t4$ (line 12).
3. The difference $(n - r)$ is computed into temporary register $t6$ to be used later (line 13).
4. Next, $n!$ is computed in lines 15 and 16 (n is still in register $a0$).
5. Lines 18 to 20 compute $r!$ in the register $t4$.
6. Lines 22 to 24 compute $(n - r)!$ in the register $t6$.
7. Register $t3$ receives the multiplication result of the denominator at line 26.
8. Finally, register $a0$ receives the quotient from dividing the numerator and the denominator in line (28), prior to returning to the caller.

It is possible to optimize this further, to reduce the number of registers needed. But, unless you can reduce the number of steps involved, it may not be worth it. I'll leave that exercise for the reader.

The main driver for the test is provided in Listing 10.8. It is like many test harness programs that we've seen before, testing the `nCr()` function with different values.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern uint64_t nCr(int64_t n,int64_t r);
5
6  struct s_test {
7      uint64_t      n;
8      uint64_t      r;
9  };
10
11 int
12 main(int argc,char **argv) {
13     static struct s_test const tests[] = {
14         { 3, 2 },      // 3
15         { 4, 3 },      // 4
16         { 9, 3 },      // 84
17         { 13, 7 },     // 1716
18         { 0, 0 }
19     };
20
21     for ( unsigned ux=0; tests[ux].n != 0; ++ux ) {
22         uint64_t ncr = nCr(tests[ux].n,tests[ux].r);
23         printf("C(n=%lu,r=%lu) => %lu\n",
24             tests[ux].n,
25             tests[ux].r,
26             ncr);
```

```
27     }
28
29     return 0;
30 }
```

Listing 10.8: The $nCr()$ function with different values.

Compile and run the test under Fedora Linux as follows:

```
$ gcc -g ncr.S main.c
$ ./a.out
C(n=3,r=2) => 3
C(n=4,r=3) => 4
C(n=9,r=3) => 84
C(n=13,r=7) => 1716
```

There are online combination calculators available where you can verify these results.[1] What was interesting about this assignment was that an internal function call was made through the temporary register `t0` to calculate the factorials. Since we didn't need to save any registers or use any stack-based variables, the entire calculation was register-based for the ultimate efficiency. With the use of the RISC-V M extension, the entire computation was performed in hardware. The number of registers available in a RISC CPU often permits optimal execution.

10.14. Summary

This chapter has shown the utility and use of the multiply and division operations provided by the RISC-V extension M. When this extension is not provided, execution time suffers considerably because these operations must be performed in software.

The next chapter will change gears somewhat and use what you've learned and apply it to addressing, indexing and array subscripting.

Bibliography

- [1] CalculatorSoup, L. L. C. (n.d.). *Combinations calculator (NCR)*. CalculatorSoup. Retrieved May 31, 2022, from <https://www.calculatorsoup.com/calculators/discretemathematics/combinations.php>

Chapter 11 • Addressing, Subscripting and Strings



There's no place like your home address

Many times, you'll need to access an array or matrix from within an assembler routine to achieve some performance goal. This chapter introduces some of the finer points of working with pointers and subscripts within the assembler function. Within C/C++, you can increment a pointer without giving it much thought. But in assembler language, there are some traps to watch out for. Additionally in this chapter, we'll examine some string-related examples.

11.1. Testing for Null Pointers

In the C language, you are often testing for the NULL pointer (or `nullptr` in C++). On most platforms today, this is a pointer with an address of zero. So, testing for a null pointer in assembly language is straightforward. If the pointer address is in register `t0`, then:

```
beqz t0,gotnull           # Branch if t0 holds a nullptr
```

Testing the address for zero will branch when the pointer is NULL/`nullptr`.

11.2. Sizeof Type for Pointers

In C/C++, each pointer has an associated data type and size for that type. On the ESP32-C3 for example, where the "int" type is 32 bits in size, a pointer to an int also has an associated data size of 4 bytes. So, when you increment an int pointer in C/C++, the address is actually incremented by 4. This is easy to forget when you are programming at the assembly language level.

11.3. Matrix Memory Layout

The C/C++ compilers use a particular matrix organization, which affects how you subscript to the correct element address. If subscripts `x` and `y` are used to access an element in the matrix `m`, then how is the matrix organized? Figure 11.1 illustrates how the subscripts map to rows and columns. This is known as row-major order, where each row element resides in sequential memory locations.



[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Figure 11.1: The subscript organization of a 3×3 C/C++ matrix.

Row-major order has the column number incrementing first, followed by the row number. A 3×3 matrix is implemented in memory storage as a linear array of 9 elements. The very first matrix element [0][0] is the first member of the array at index 0. The second row [1][0] starts at index 1*3 + 0, or index 3. The same matrix is shown in Figure 11.2 with linear array index numbers.



0	1	2
3	4	5
6	7	8

Figure 11.2: The mapping of a matrix to a 1-dimensional array.

There is one more important aspect to this matrix business to remember. Each element may be composed of multiple bytes, such as a matrix of "int" types. This must be taken into account when computing the byte address of a particular element.

11.3.1. Subscript Calculation

In memory, all matrices are linear arrays in the end. As Figure 11.2 illustrated, row-major order is used and from this, we can derive a formula for a linear array index. For any matrix *m* of *r* rows and *c* columns, the linear array index *x* for an element can be computed for subscripts *i* and *j* as:

$$x = i * c + j$$

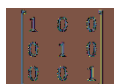
For example, if you have a 5×7 matrix, then the index to the *m*[3][2] element, is computed as:

$$23 = 3 * 7 + 2$$

Since a 5×7 matrix occupies the same storage as a 35-element linear array *a*, the index of *m*[3][2] is equivalent to accessing *a*[23].

11.4. Identity Matrix Example

To receive some hands-on experience in these matters but keeping things simple, let's examine an assembly language function *identm()* that efficiently initializes an integer identity matrix. An identity matrix contains all zero elements except for those on the diagonal. Figure 11.3 illustrates a 3×3 identity matrix.



1	0	0
0	1	0
0	0	1

Figure 11.3: A 3×3 identity matrix.

This time let's examine the main program first so that we're clear about the program elements being operated upon. Listing 11.1 illustrates the C main program that defines two

matrices and initializes each of them according to their size using our assembly language routine `identm()`.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern void identm(void *matrix,unsigned n);
5
6  void
7  app_main(void) {
8      {
9          unsigned const n=3;
10         int matrix[n][n];
11
12         identm(&matrix,n);
13
14         printf("%u x %u identity matrix:\n",n,n);
15         for ( unsigned ux=0; ux<n; ++ux ) {
16             putchar('[');
17             for ( unsigned uy=0; uy<n; ++uy ) {
18                 printf("%3d ",matrix[ux][uy]);
19             }
20             puts("]");
21         }
22     }
23     {
24         unsigned const n=6;
25         int matrix[n][n];
26
27         identm(&matrix,n);
28
29         printf("%u x %u identity matrix:\n",n,n);
30         for ( unsigned ux=0; ux<n; ++ux ) {
31             putchar('[');
32             for ( unsigned uy=0; uy<n; ++uy ) {
33                 printf("%3d ",matrix[ux][uy]);
34             }
35             puts("]");
36         }
37     }
38
39     puts("Done");
40     fflush(stdout);
41 }
```

Listing 11.1: Main program `~/riscv/repo/11/identm/main/main.c`.

This program defines a 3×3 matrix in line 10 and a 6×6 matrix in line 25. The matrix elements are of type "int", which are 32 bits for the ESP32-C3. As created these matrices are uninitialized. The calls to `identm()` in lines 12 and 27 will, however, initialize them according to their dimensions.

Line 4 declares the function prototype for our assembly language routine:

```
4 extern void identm(void *matrix,unsigned n);
```

The first argument is declared as a void pointer to avoid limitations of the C language. We might be tempted to write:

```
4 extern void identm(int matrix[][],unsigned n);
```

but the compiler will only accept "[]" for the *last* dimension of the matrix. To avoid this problem, we define that argument to be a byte address of the matrix, without any type checking by the compiler. The second parameter declares the dimension of the square matrix `n`.

The assembler language routine is illustrated in Listing 11.2. Let's break down its operation:

1. The pointer argument arrives in register `a0`, as a pointer to the first byte of the matrix.
2. The dimension `n` arrives in register `a1`.
3. Line 10 multiplies `n` times `n` to arrive at the total number of elements of the matrix. This is placed in temporary register `t1`.
4. Temporary register `t0` is loaded with the constant 4 (line 11). This corresponds to the `sizeof(int)`.
5. Finally, `t1` is multiplied by `t0` (with the `sizeof(int)`) so that `t1` becomes the total number of bytes that the matrix occupies (line 12).
6. The pointer to the matrix (in `a0`) is added to `t1` (total # of bytes) with the result placed into temporary register `t2` (line 13). This is the pointer to the last byte after the end of the matrix storage. This simplifies the end of the loop handling later.
7. Temporary register `t5` is loaded with the constant 1, to be stored on the diagonal elements (line 14).
8. At the top of the outer loop at label "put1", we store the value of 1 (in `t5`) at the current pointer in `a0` (line 16). Note that we are storing a "word" or 4 bytes.
9. Now the pointer in `a0` is incremented by 4 (`sizeof(int)`) in line 17.
10. The temporary register `t3` is loaded with `n` (in `a1`) at line 19. This will control the number of zeros that will be stored.
11. The loop in lines 20 to 24 will then store `n` zeros. There are `n` zeros between each 1 value on the diagonal.
12. A test is made at the top of the inner loop at line 20 to see if we have reached the end of the matrix. If so, the branch is taken to label "end" to return to the caller.
13. Otherwise, a zero is stored from line 21.
14. The pointer is incremented by `sizeof(int)` at line 22.

15. The counter in t3 is decremented (line 23) and branches back to "loop" (line 20) if the counter is non-zero (from line 24).
16. Otherwise, control falls through to line 25. As long as the pointer is still within the matrix we loop back to the outer loop label "put1" at line 16.
17. Eventually, line 20 will branch to "end" and return to the caller.

```

1      .global identm
2      .text
3
4      #      extern void identm(void *matrix,unsigned n)
5      #
6      # ARGUMENTS:
7      #      a0      Pointer to int matrix[n][n]
8      #      a1      unsigned n
9
10     identm: mul    t1,a1,a1      # t1 = total elements
11         li      t0,4          # sizeof(int) = 4
12         mul    t1,t1,t0      # t1 *= sizeof(int)
13         add    t2,a0,t1      # Ptr of end of matrix
14         li      t5,1          # t5 = 1
15
16     put1:  sw     t5,0(a0)      # *ptr = 1
17         addi   a0,a0,4        # ptr += 4
18
19         mv     t3,a1          # t3 = n
20     loop: bge    a0,t2,end     # At end of matrix?
21         sw     x0,0(a0)       # *matrix + ptr = 0
22         addi   a0,a0,4        # ptr += 4
23         addi   t3,t3,-1       # --t3
24         bnez   t3,loop        # Loop again if not at end
25         blt    a0,t2,put1     # Loop again if not at end
26
27     end:   ret

```

Listing 11.2: The `identm()` function in file `~/riscv/repo/11/identm/main/identm.S`.

The main lesson in this example is the need to be aware of the size of the matrix element. When I initially wrote this routine it failed because I was incrementing the pointer (in a0) by 1, rather than by the `sizeof(int)`, which is 4. This is a very easy mistake to make, so please take note.

Build, flash and monitor this program on the ESP32-C3 as follows:

```

$ cd ~/riscv/repo/11/identm
$ idf.py build
...

```



```

$ idf.py -p <<<yourport>>> flash monitor
...
I (258) cpu_start: Starting scheduler.
3 x 3 identity matrix:
[ 1  0  0 ]
[ 0  1  0 ]
[ 0  0  1 ]
6 x 6 identity matrix:
[ 1  0  0  0  0  0 ]
[ 0  1  0  0  0  0 ]
[ 0  0  1  0  0  0 ]
[ 0  0  0  1  0  0 ]
[ 0  0  0  0  1  0 ]
[ 0  0  0  0  0  1 ]
Done

```

From the displayed output, it is verified that the initialization was correctly done.

Tip: The observant reader will notice that the RISC-V opcodes for compare and set only includes "slt", "sltu" and the immediate constant forms. But what if you wanted to test for greater-than-or-equal-to for example? This can be done by reversing the operands of the comparison. If you want to test $a \geq b$, then use:

```
slt rd,b,a          # Test  $a \geq b$  by testing  $b < a$ 
```

11.5. String Functions

One area where assembly language may be of great service is in the special handling of string data. So, let's examine some common functions as well as some string conversions.

11.5.1. Function strlen()

The strlen() function is pretty basic to the C language support, and there is no need to replace it. But it is instructive to write one to see how much code it would require. Listing 11.3 illustrates our assembly language version of strlen32(). It is pretty basic:

1. The pointer to the first byte of the string is passed as an argument in register a0. This pointer is copied to temporary register t0 at line 13, since we need a0 to return the result.
2. The register a0 is zeroed at line 14. This will be the byte count.
3. The loop starts at line 16, loading into register t1 the byte at the pointer (in t0). Be aware that this is a signed character load (with sign extended) but that doesn't affect this algorithm.
4. A branch is taken from line 17 to label "end" (step 7), if the byte we just loaded from memory was zero (i.e. a null byte).
5. If control continues to line 18, we then increment a0, which holds the current string length.

6. The pointer value in `t0` is incremented in line 19, before looping back to line 16.
7. The register `a0` already has the accumulated string length to return, so the control returns to the caller in line 22.

```
1      .global strlen32
2      .text
3
4      #      extern int strlen(char const *s)
5      #
6      # ARGUMENTS:
7      #      a0      Pointer to string
8      #
9      # RETURNS:
10     #      a0 (int) string length
11
12     strlen32:
13         mv      t0,a0          # t0 = char const *ptr
14         li      a0,0          # Zero strlen
15
16     loop:  lb      t1,0(t0)     # t1 = *ptr
17         beqz   t1,end         # Branch if Null byte
18         addi   a0,a0,1        # Increment strlen
19         addi   t0,t0,1        # ++ptr
20         j      loop
21
22     end:    ret
```

Listing 11.3: Function `strlen32()` file `~/riscv/repo/11/strlen/main/strlen.S`.

The main program is shown in Listing 11.4, which calls `strlen32()` with a few different strings and prints the results.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern int strlen32(char const *s);
5
6  void
7  app_main(void) {
8      static char const *tests[] = {
9          "One",
10         "Three",
11         "Ten four",
12         "",
13         "This is the end!",
14         NULL
```

```

15     };
16
17     for ( unsigned ux=0; tests[ux] != NULL; ++ux ) {
18         printf("strlen32('%s') => %d\n",
19             tests[ux],strlen32(tests[ux]));
20     }
21
22     puts("Done");
23     fflush(stdout);
24 }

```

*Listing 11.4: The main program for testing strlen32() in file
~/riscv/repo/11/strlen/main/main.c.*

Build, flash and monitor the program as follows:

```

$ cd ~/riscv/repo/11/strlen
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
I (257) cpu_start: Starting scheduler.
strlen32('One') => 3
strlen32('Three') => 5
strlen32('Ten four') => 8
strlen32('') => 0
strlen32('This is the end!') => 16
Done

```

From these results, we can see that the function() produced the correct results.

11.5.2. Function strncpy32()

One standard function that has always bugged me is the function strncpy(). Its function prototype is as follows:

```
char *strncpy(char *dest, const char *src, size_t n);
```

This function copies characters from pointer src to pointer dest, until a null byte is encountered in the source string *or* n characters have been copied, whichever occurs first. When the src string is less than n characters, null bytes are appended until n characters have been placed into the destination buffer. Note that if the src string is greater or equal to n in length, there is no null byte placed into the destination buffer. This is often the source of many a C language program bug!

My complaint about this function is about efficiency. If you have a large receiving buffer, say 1024 bytes in length, and you happen to have a short string to copy into it, then the call is wasteful. For example, assume the following:

```
char buf[1024];
...
strncpy(buf,"abc",sizeof buf);
```

The `strncpy()` function will copy the three characters "abc" and then fill the remaining 1021 bytes of `buf` with null bytes. This results in a nice and clean `buf` array but is wasteful of CPU cycles. Cost-wise, it would be best if `strncpy()` copied the three characters "abc" and only *one* null byte. So let's implement our own `strncpy32()` function to do exactly that.

Listing 11.5 illustrates our more efficient `strncpy()` function, named `strncpy32()` to avoid affecting other functions that may depend upon the original behaviour. Let's explain its operation:

1. Registers `a0`, `a1` and `a2` receive the calling arguments. Note that we return the argument `dest`, so it remains left alone in register `a0` during this call.
2. Register `t6` has the `dest` pointer copied into it in line 15. We will be incrementing this pointer as the execution proceeds.
3. Register `t5` receives the calculated end of the buffer pointer in line 16. This is the original `dest` argument plus the value `n`.
4. The top of the loop begins in line 18. The branch to label "end" is taken if our working `dest` pointer in `t6` has gone past the end of the buffer.
5. Otherwise, we do an unsigned byte load in line 19, from the `src` pointer.
6. The `src` pointer is incremented by 1 (line 20).
7. If the byte loaded into `t4` at line 19 is a null byte, then branch to label "nul" (line 21).
8. Otherwise, store the byte in `t4` at the destination buffer using the working pointer in `t6` (line 22).
9. The `dest` pointer is incremented in line 23, and the loop repeats starting again at line 18.
10. If a null byte is encountered at line 21, we arrive at line 26. Here one null byte is stuffed into the `dest` buffer using working pointer `t6`.
11. Arriving at label "end" from line 26 or line 18, causes execution to return to the caller. The unmodified `dest` pointer in `a0` is returned.

```
1      .global strncpy32
2      .text
3
4      #      extern char *strncpy32(char *dest,char const *src,size_t n);
5      #
6      # ARGUMENTS:
7      #      a0      char *dest (also returned)
8      #      a1      char const *src
9      #      a2      size_t n
10     #
11     # RETURNS:
12     #      a0      char *dest
```

```

13
14 strncpy32:
15     mv     t6,a0           # t6 = dest ptr
16     add   t5,a0,a2       # Points past end of dest buf
17
18 loop:  bge   t6,t5,end    # Branch if we passed dest buf end
19     lbu   t4,0(a1)       # Load byte from source
20     addi  a1,a1,1        # ++src
21     beqz  t4,nul         # Branch if null byte
22     sb    t4,0(t6)       # Copy byte to dest
23     addi  t6,t6,1        # ++dest
24     j     loop
25
26 nul:   sb    x0,0(t6)    # Store null byte
27 end:   ret

```

*Listing 11.5: The strncpy32() function in file
~/riscv/repo/11/strncpy32/main/strncpy32.S.*

The main program to test our strncpy32() function is provided in Listing 11.6. The dest buffer is declared in line 16 to hold 8 characters. Various tests from the for loop in line 19. To validate our test, we fill the array buf with 8 character 'X' bytes in line 20. Then we call our assembler routine strncpy32() at line 21, taking note of the returned pointer in variable rp. This is checked in the assertion at line 23. Lines 24 to 32 then report the results of what the array buf contains.

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <assert.h>
4
5  extern char *strncpy32(char *dest,char const *src,size_t n);
6
7  void
8  app_main(void) {
9      static char const *tests[] = {
10         "abc",
11         "Main",
12         "",
13         "1234567890",
14         NULL
15     };
16     char buf[8], *rp;
17     char const *src;
18
19     for ( unsigned ux=0; (src = tests[ux]) != NULL; ++ux ) {
20         memset(buf,'X',sizeof buf);

```

```

21         rp = strncpy32(buf,src,sizeof buf);
22
23         assert(rp == buf);
24         printf("src='%s', n=%u, buf[] = ",src,sizeof buf);
25         for ( unsigned u=0; u<sizeof buf; ++u ) {
26             if ( buf[u] )
27                 printf("%c%c",
28                     buf[u],
29                     u+1<sizeof buf?','':'\n');
30             else    printf("NUL%c",
31                     u+1<sizeof buf?','':'\n');
32         }
33     }
34
35     puts("Done");
36     fflush(stdout);
37 }

```

*Listing 11.6: Main program to test strncpy32() in file
~/riscv/repo/11/strncpy32/main/main.c.*

Build, flash and execute the project as follows:

```

$ cd ~/riscv/repo/11/strncpy32
$ idf.py build
...
idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
src='abc', n=8, buf[] = 'a','b','c',NUL,'X','X','X','X'
src='Main', n=8, buf[] = 'M','a','i','n',NUL,'X','X','X'
src='', n=8, buf[] = NUL,'X','X','X','X','X','X','X'
src='1234567890', n=8, buf[] = '1','2','3','4','5','6','7','8'
Done

```

How did we do? When the short string "abc" was copied, the destination buffer (buf) received only the letters 'a', 'b', 'c' and the one null byte at the end, as expected. The library strncpy() routine would have filled the entire buffer with null bytes. In the third test, an empty source string was tested, but passed with just one null byte. In the last case, we supplied a longer source string than the destination buffer could hold. This too passed, but be careful in this case, since there is no terminating null byte.

11.5.3. String to Integer Conversion

Data arrives by different means but often in the form of character text. In order to perform a computation, that text must be converted into a numeric data type. Let's write a simple assembly language function to accept an unsigned number in text form and produce an

unsigned int value. This function will also return a boolean indicating whether or not the conversion was successful.

Listing 11.7 illustrates the function `struint()`. This function will skip all space characters but will fail if any other non-digit character is encountered.

```

1      .global struint
2      .text
3
4      #      extern unsigned struint(char const *text,bool *ok)
5      #
6      # ARGUMENTS:
7      #      a0      char const *text (text to convert)
8      #      a1      pointer to bool
9      #
10     # RETURNS:
11     #      a0      unsigned value (when ok is true)
12     #      ok:
13     #              true, conversion successful
14     #              false, conversion failed
15
16     struint:
17         mv      t6,a0          # t6 = ptr to test
18         li      a0,0          # Accumulator for uint
19         li      t2,0          # Digit count
20         li      t4,'0'
21         li      t3,'9'
22         li      t1,' '
23         li      t0,10
24
25     loop:  lbu      t5,0(t6)      # Load text char
26             beqz   t5,nulbyt     # Branch if null byte
27             beq    t5,t1,skip    # Skip white space
28             bgt    t5,t3,fail    # char > '9'?
29             blt    t5,t4,fail    # char < '0'?
30             andi   t5,t5,0x0F    # Mask out 0x00 to 0x09
31             mul   a0,a0,t0      # a0 *= 10
32             add   a0,a0,t5      # a0 += t5
33             addi  t2,t2,1       # Bump digit count
34     skip:  addi   t6,t6,1       # ++text ptr
35             j     loop
36
37     nulbyt: beqz   t2,fail       # Fail if no digits
38             li    t0,1
39             sb    t0,0(a1)      # ok = true
40             ret

```

```

41
42 fail:  li      t0,0
43 exit:  sb      t0,0(a1)      # ok = false
44        ret

```

Listing 11.7: Routine `struint()` in file `~/riscv/repo/11/struint/main/struint.S`.

The breakdown of `struint()` is as follows:

1. The pointer to the input text arrives in register `a0`, and `a1` is a pointer to the bool that will receive a pass (1) or fail (0) status.
2. Register `t6` receives a working copy of the input text address from `a0` (line 17).
3. Register `a0` is the value to be returned. It is initialized to zero in line 18.
4. The digit count in `t2` is initialized to zero (line 19).
5. Registers `t4` and `t3` are loaded with the ASCII characters '0' and '9' respectively, to be used for comparison purposes (lines 20, 21).
6. Register `t1` loads a space character, for comparison purposes (line 22).
7. Temporary register `t0` loads the value 10, to be used for multiplication (line 23).
8. The loop begins at line 25 by loading a text character into `t5`.
9. A branch is taken to "nulbyt" if the loaded character is 0x00 (line 26).
10. A branch to "skip" is taken if the character loaded was a space (line 27).
11. Lines 28 and 29 test if the character is a digit. If not, a branch is made to "fail".
12. The last 4 bits of the digit are masked out in line 30.
13. Line 31 multiplies the accumulated unsigned value in register `a0`, by 10.
14. Then the isolated digit from step 12 is added to `a0` (line 32).
15. Line 33 increments the digit count (for validating results).
16. Line 34 increments the text byte pointer, before returning to the top of the loop (at step 8).
17. If execution lands at "nulbyt", the digit count in `t2` is checked. If there are no digits, the branch to "fail" is taken (line 37).
18. Otherwise, `ok` is set to true, and control returns to the caller.
19. If execution lands at "fail", the value of "ok" is set to false.

There is considerable room for improvement in the error checking in this routine, but it was kept simple to highlight the important concepts. The main driver program is illustrated in Listing 11.8, calling `struint()` with various strings and reporting the results.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  extern unsigned struint(char const *text, bool *ok);
5
6  void
7  app_main(void) {
8      static char const *tests[] = {
9          " 123",

```



```

10         "0234",
11         "0",
12         "905",
13         "9_05",
14         " 907, ",
15         NULL
16     };
17     bool ok;
18     unsigned v;
19
20     for ( unsigned ux=0; tests[ux] != NULL; ++ux ) {
21         ok = 0;
22         v = struint(tests[ux],&ok);
23
24         printf("struint('%s') => %u, ok=%d\n",
25             tests[ux],v,ok);
26     }
27
28     puts("Done");
29     fflush(stdout);
30 }

```

Listing 11.8: Main program for struint() in file ~/riscv/repo/11/struint/main/struint.S.

Build, flash and monitor the program as follows:

```

$ cd ~/riscv/repo/11/struint
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
struint(' 123') => 123, ok=1
struint('0234') => 234, ok=1
struint('0') => 0, ok=1
struint('905') => 905, ok=1
struint('9_05') => 9, ok=0
struint(' 907, ') => 907, ok=0
Done

```

From the test run output, all of the input texts converted ok except for the last two. The failures were due to the underscore (`_`) and comma (`,`) characters that were encountered in the input string.

11.5.4. What if there is no Multiply?

While we live in relative luxury on the ESP32-C3, what do you do on *embedded* platforms that do not provide for hardware multiplication? It turns out that the multiplication can be done inline when we only have to multiply by ten.

Replace line 31 in `struint.S` (where the multiply is performed) with the following:

```
sll    a6,a0,3           # a6 = a0 * 8
sll    a0,a0,1          # a0 *= 2
add    a0,a0,a6         # a0 += a6
```

And retry the build, flash and run. If you made the change correctly, the execution results will be identical. These three instructions perform the multiplication of `a0` by 10 as follows:

1. Shift `a0` left 3 bits, which effectively multiplies the value by 8 and save this in `a6`.
2. Shift `a0` left 1 bit, which effectively multiplies the value by 2.
3. Add these two and the result is the original value multiplied by 10.

Is this faster than using the hardware multiply? Without timing information or doing a benchmark on the ESP32-C3, it is difficult to know.

Tip: When processing with a character pointer, it is often convenient to look at the next or previous character without changing the pointer.

If the pointer variable in C is `cp`, then `*cp` or `cp[0]` returns the current character while `cp[1]` and `cp[-1]` return the next and prior characters respectively. In RISC-V assembly, if the pointer is held in register `t3`, then `"0(t3)"` references the current character, while expressions `"1(t3)"` and `"-1(t3)"` access the next and prior characters respectively.

11.5.5. Integer to String Conversion

We've examined the conversion from an ASCII string, so now let's perform the reverse. Converting an unsigned integer into printable text, the assembly language program for function `uintstr()` is shown in Listing 11.9.

```
1      .global uintstr
2      .text
3
4      #      extern char *uintstr(unsigned u,char const *buf,unsigned buflen)
5      #
6      # ARGUMENTS:
7      #      a0      unsigned value to convert to text
8      #      a1      pointer to buf
9      #      a2      max length for buf
10     #
11     # RETURNS:
12     #      a0      pointer to buf
13
14     uintstr:
```

```

15      add    t5,a1,a2      # ptr past end of buffer
16      beqz   a2,smlbuf    # Branch if zero length buffer
17      sb     x0,-1(t5)    # Put nul byte
18      addi   t5,t5,-1     # --ep
19      li     t1,10        # t1 = 10
20
21 loop: div    t4,a0,t1     # t4 = a0 / 10
22      rem    t3,a0,t1     # t3 = a0 % 10
23      mv     a0,t4        # now a0 /= 10
24      addi   t3,t3,'0'    # Make ascii digit
25      ble    t5,a1,smlbuf # Branch if past start of buf
26      addi   t5,t5,-1     # --ep
27      sb     t3,0(t5)     # *ep = char
28      bnez   a0,loop
29
30 smlbuf: mv    a0,t5       # Return addr of first char
31      ret

```

Listing 11.9: The `uintstr()` function from file `~/riscv/repo/11/uintstr/main/uintstr.S`.

The `uintstr()` function uses the quotient and remainder from a division, to convert an unsigned value into a text string, one digit at a time. When the function is called register `a0` contains the value to be converted, `a1` points to the receiving character buffer, and `a1` is the maximum length for that buffer. The steps for conversion are as follows:

1. Register `t5` is set to the pointer of the buffer + its maximum length. This points one byte past the end of the passed buffer (line 15). This is done because we must fill the buffer in reverse order. This is because each division will determine the low order decimal digits first.
2. Line 16 tests for a zero-length buffer, and if so, just returns at line 30, returning the buffer pointer.
3. A null byte is stored at the very end of the caller's buffer in line 17.
4. Then the pointer in `t5` is decremented towards the start of the buffer by one (line 18).
5. The unsigned integer 10 is loaded into `t1` (line 19). This will be the divisor used in the loop.
6. The loop begins in line 21. Here the value of `a0` (the unsigned integer) is divided by 10 and the result is placed in `t4`.
7. Line 22 also fetches the remainder into register `t3` (Line 24).
8. The ASCII value for '0' is added to the remainder in `t3`, to form a digit character, and this replaces `t3` (Line 24).
9. The pointer in `t5` is tested against the one in `a1`, to see if we have gone past the start of the buffer. If so, the branch is taken to "smlbuf" to exit safely (line 25).
10. Otherwise, pointer register `t5` is decremented once more (line 26) and then the character is stored in the buffer at line 27.
11. If the division did not end in a quotient of zero, we loop back to step 6.

The main driver program for this test is provided in Listing 11.10. Of particular note, notice that the pointer returned by `uintstr()` is saved in pointer variable `cp` in line 16. This is done because the start of the converted number is not necessarily going to be at the start of the array `buf` when function `uintstr()` returns. This is due to the fact that function works from the tail end of the buffer and works backwards.

```
1  #include <stdio.h>
2
3  extern char *uintstr(unsigned v,char *buf,unsigned buflen);
4
5  void
6  app_main(void) {
7      static unsigned const tests[] = {
8          1023, 32, 96001, 10045, 90999, 1770771,
9          0, 0xFFFFFFFF
10     };
11     char buf[7];
12     char const *cp;
13     unsigned v;
14
15     for ( unsigned ux=0; (v = tests[ux]) != 0xFFFFFFFF; ++ux ) {
16         cp = uintstr(v,buf,sizeof buf);
17
18         printf("uintstr(%u,buf,%u) => '%s' %s\n",
19             v,(unsigned)sizeof buf,cp,
20             cp <= buf ? "!!" : "");
21     }
22
23     puts("Done");
24     fflush(stdout);
25 }
```

*Listing 11.10: Main driver program to test `uintstr()` in file
~/riscv/repo/11/uintstr/main/uintstr.S.*

Build, flash and monitor the program as follows:

```
$ cd ~/riscv/repo/11/uintstr
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
uintstr(1023,buf,7) => '1023'
uintstr(32,buf,7) => '32'
uintstr(96001,buf,7) => '96001'
```

```

uintstr(10045,buf,7) => '10045'
uintstr(90999,buf,7) => '90999'
uintstr(1770771,buf,7) => '770771' !!
uintstr(0,buf,7) => '0'
Done

```

All conversions worked except for the one identified with "!!". The buffer was not large enough to support writing the text to buf, to contain the value 1770771. This was intentionally done to test the safety of the function.

11.6. Indexed Branching

Rather than have a long list of if-then-branch statements in C or assembler language, it is sometimes more efficient to have a "computed goto", when it can be arranged. In other words, use an index to select a jump or call based upon an index value. Let's first examine the main driver program in Listing 11.11.

Lines 3 to 10 define an enumerated value for each function we want to execute (the exception is func_bad, which is used to test the handling of a bad index). The values start at zero for func_add, with a value of 1 for func_sub, etc.

Our assembly language function cgoto() will execute an arithmetic function based upon the first argument "fun" (line 12). Line 16 defines an array for six returned results that will be reported once the testing is completed. Lines 18 through 23 test the function with different arguments and requested functions.

```

1  #include <stdio.h>
2
3  typedef enum {
4      func_add=0,
5      func_sub,
6      func_mul,
7      func_div,
8      func_rem,
9      func_bad
10 } func_t;
11
12 extern int cgoto(func_t fun,int a,int b);
13
14 void
15 app_main(void) {
16     int r[6];
17
18     r[0] = cgoto(func_add,1,2);
19     r[1] = cgoto(func_sub,5,2);
20     r[2] = cgoto(func_mul,6,5);
21     r[3] = cgoto(func_div,35,6);

```

```

22     r[4] = cgoto(func_rem,13,5);
23     r[5] = cgoto(func_bad,9,99);
24
25     for ( unsigned ux=0; ux<6; ++ux ) {
26         printf("r[%u] = %d\n",ux,r[ux]);
27     }
28     puts("Done");
29     fflush(stdout);
30 }

```

*Listing 11.11: Main driver function for testing cgoto() in file
~/riscv/repo/11/cgoto/main/main.c.*

Now let's examine Listing 11.12, which illustrates the assembly language code. When `cgoto()` is called, register `a0` contains the function number, and registers `a1` and `a2` have the integer operands to compute with.

1. Lines 15 and 16 do a range check on the function requested. If the value is out of range, then control passes to label "null" (at line 34), which then returns the result -1 in protest.
2. Line 17 multiplies the function index by 4 (by shifting left 2), to change the function index into a byte offset (we need to address the table entries (line 37) by byte address. Since our table need not change, it is kept in `.text` and remains read-only.
3. The address of "table" is loaded into temporary register `t5` (line 18).
4. The byte offset is added to the address in `t5`, to address the required word in "table". The result of that is placed in register `t4` (line 19).
5. The address of the starting code required replaces `t4` by loading the word from the table (line 20).
6. In line 21, we finally jump to the required code using register `t4`. Depending upon the calculation, the code will jump to "add", "sub", "mul", "div" or "rem".

```

1     .global cgoto
2     .text
3
4 #     Computed goto example:
5 #     extern int cgoto(unsigned func,int a,int b);
6 #
7 # ARGUMENTS:
8 #     a0     function
9 #     a1     int a
10 #     a2     int b
11 #
12 # RETURNS:
13 #     a0     result
14
15 cgoto: li     t6,4

```

```

16      bgt    a0,t6,null    # Branch and return if > 4
17      sll    a0,a0,2      # Turn into a byte address
18      la     t5,table     # Establish t5 as table address
19      add    t4,a0,t5     # Address table entry
20      lw     t4,0(t4)     # Load routine address
21      jr     t4           # Jump to code
22
23 add:   add    a0,a1,a2
24       ret
25 sub:   sub    a0,a1,a2
26       ret
27 mul:   mul    a0,a1,a2
28       ret
29 div:   div    a0,a1,a2
30       ret
31 rem:   rem    a0,a1,a2
32       ret
33
34 null:  li     a0,-1      # Bad function
35       ret
36
37 table: .word  add
38       .word  sub
39       .word  mul
40       .word  div
41       .word  rem

```

Listing 11.12: The `cgoto()` function in file `~/riscv/repo/11/cgoto/main/cgoto.S`.

Build, flash and run the code as follows:

```

$ cd ~/riscv/repo/11/cgoto
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
r[0] = 3
r[1] = 3
r[2] = 30
r[3] = 5
r[4] = 3
r[5] = -1
Done

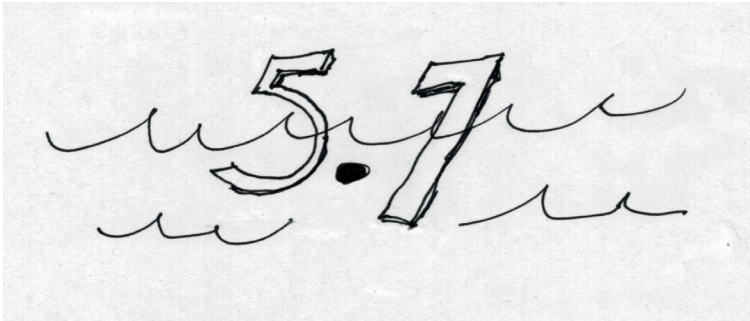
```

Checking with the main program, the reported results are correct. The sixth result of -1 is due to requesting an out-of-range function index, which the routine anticipated.

11.7. Summary

This chapter began with a focus on matrix and array subscripting. From there, a matrix identity function and some string functions were developed. The last example demonstrated how to compute a branch based upon an index value for efficiently selecting the desired code to run. At this point, you should be well equipped for programming RISC-V for integer-based data. In the next chapter, we will explore the hardware floating point.

Chapter 12 • Floating Point



A Floating Point

In chapter 6, *Load and Store*, we managed to convert a temperature in Fahrenheit to degrees Celsius using only integers. Managing more complicated computations in integers is inconvenient and error-prone. Even when there is no hardware floating-point capability, it is often preferred to use software floating-point routines to make the calculations easier. The floating-point data format provides for fractional values and a greater range by use of exponents.

In this chapter, we'll explore the hardware support of floating-point for RISC-V. The ESP32-C3 does not have this facility so it manages by using compiler-supplied software libraries. The QEMU emulated Fedora Linux, however, does emulate hardware floating-point, giving us the opportunity to exercise the floating-point opcodes.

There is considerable theory surrounding floating-point data formats. Since this is a tutorial book, I will assume that you are familiar with that material or will research it when necessary. So, without further delay, let's examine the RISC-V hardware registers and instructions available.

12.1. Floating Point Registers

There are 32 floating point registers of FLEN bits, named f0 to f31 (the embedded E variant will only have 16 registers). The value of FLEN is 32 for the RISC-V standard extension "F" for Single-Precision Floating-Point values. RISC-V extension "D" adds the capability for Double-Precision Floating-Point values, where FLEN is 64. There are also extensions "Q" and "V" that we will not be covering here.

When extension "D" is supported, the floating-point registers are capable of holding 32-bit single-precision or 64-bit double-precision values. Most of the floating-point instructions work upon the floating-point register file. There are, however, some that transfer values to or from an integer register. Finally, there are also opcodes, which load or store floating-point values from or to memory.

Unlike the integer register x0, the floating-point register f0 has no special function and may be used in general calculations.

12.2. GNU Calling Convention

Like the integer registers, the GNU calling convention assigns ABI Names, uses and responsibilities for saving them. Using the ABI register names helps maintain the correct GNU register conventions. Table 12.1 lists the hardware floating-point register names, the ABI name and responsibilities (saver).

Register	ABI Name	Description	Saver
f0-f7	ft0-ft7	Floating-point temporaries	Caller
f8-f9	fs0-fs1	Floating-point saved registers	Callee
f10-f11	fa0-fa1	Floating-point arguments/return values	Caller
f12-f17	fa2-fa7	Floating-point arguments	Caller
f18-f27	fs2-fs11	Floating-point saved registers	Callee
f28-f31	ft8-ft11	Floating-point temporaries	Caller

Table 12.1: Floating-point registers and their ABI names and savers.

12.3. Floating-Point Control and Status Register (fcsr)

In addition to the floating-point register file, there is a 32-bit status register named fcsr. This register contains a mode and accrued exception flags (fflags). We have not yet examined the Control and Status Register (CSR), shown in Figure 12.1.

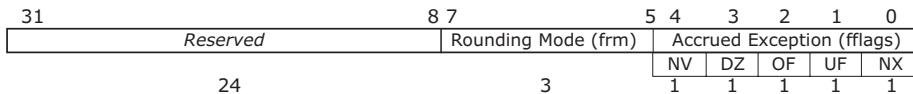


Figure 12.1 The Floating-Point Control and Status Register.

The fcsr.frm field establishes a default rounding mode to be used or when the opcodes specify "dyn". The accrued exceptions field fcsr.fflags, remain set until they are cleared. In this manner, the programmer has the option of testing these flags after each opcode or at the end of a calculation.

The values for fcsr can be loaded and modified using the following pseudo-ops:

```
frcsr    rd          # rd = fcsrr
fscsr    rd,rs1     # rd = original fcsr, fcsr = rs1
```

The "frcsr" simply reads fcsr into the destination register. The "fscsr" replaces the fcsr with the value in integer register rs1, after loading the destination register with the original fcsr value.

The reserved field is for use with other standard extensions. For example, extension L provides for decimal floating-point. The RISC-V standards document has this to say about the reserved field:[1]

If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

12.3.1. Rounding Modes `fcsr.frm`

The floating-point rounding mode can be controlled by the `fcsr.frm` field or from the instruction opcode itself. The enumerated list of rounding modes is provided in Table 12.2.

Rounding Mode	Mnemonic	Meaning
000	<code>rne</code>	Round to Nearest, ties to Even
001	<code>rtz</code>	Round towards Zero
010	<code>rdn</code>	Round Down (towards $-\infty$)
011	<code>rup</code>	Round Up (towards $+\infty$)
100	<code>rmm</code>	Round to Nearest, ties to Max Magnitude
101		Reserved for future use
110		Reserved for future use
111	<code>dyn</code>	Dynamic rounding mode: use instruction's <code>rm</code> field to select rounding mode

Table 12.2: Floating-point rounding mode encoding.

Note: The GNU assembler recognizes the mnemonics of Table 12.2 for the optional rounding mode parameter of an opcode. However, if you want to load a rounding mode as immediate data in `fsrmi` for example, then you must either specify the immediate data as a numeric constant or declare a symbol with the correct value. For example, you might declare a symbol `.equ rmm,4`, and then use `fsrmi x0,rmm`.

The following pseudo-opcodes are available for working with the `fcsr.frm` field directly.

```
frrm    rd                # rd = fcsr.frm
fsrcm   rd,rs1           # rd = fcsr.frm, fcsr.frm=rs1
fsrcmi  rd,imm          # rd = fcsr.frm, fcsr.frm=imm
```

The opcode "frrm" simply copies the `fcsr.frm` into the destination integer register (bit positions left of the loaded value are set to zero). The "fsrcm" opcode likewise loads the destination integer register with the original `fcsr.frm` and sets `fcsr.frm` from `rs1`. The "fsrcmi" opcode is similar, except that `fcsr.frm` is set from the immediate data.

The following is an instruction that loads register `a2` with the current copy of `fcsr.frm`, while setting the `fcsr.frm` from the current value of `t3`:

```
fsrcm   a2,t3            # a2 = fcsr.frm, fcsr.frm=t3
```

12.3.2. Accrued Exception Flags `fcsr.fflags`

The meanings of the accrued exception flags are listed in Table 12.3. The following pseudo-opcodes are available for your convenience:

```
frrflags rd           # rd = fcsr.fflags
fsflags rd,rs1       # rd = fcsr.fflags, fcsr.fflags = rs1
fsflagsi rd,imm      # rd = fcsr.fflags, fcsr.fflags = imm
```

The "frrflags" pseudo-opcode conveniently loads the flags into the destination integer register (bits left of the flags are all set to zero). Opcode "fsflags" likewise loads the flags into the destination register, but also sets the flags from integer register rs1. Finally, "fsflagsi" performs the same except that the fcsr.fflags are set from immediate data instead.

Flag Mnemonic	Flag Meaning
NV	Invalid operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Table 12.3: Floating-point accrued exception flag encoding.

12.4. NaN Generation and Propagation

Floating-point handling is somewhat complex and messy. A number of mathematical operations results in NaN (Not a Number) and infinite values (positive and negative). For more information about the rules pertaining to this, see both the RISC-V standards documents and the IEEE Floating-Point Formats. For now, it is enough just to be aware of these special values.

12.5. Opcodes and Data Formats

Floating-point opcodes work with different floating data formats. These formats are listed in Table 12.4. In this chapter, we will focus on the S and D formats.

fmt Field	Mnemonic	Meaning	Bits	Extension
00	S	Single-precision	32	F
01	D	Double-precision	64	D
10	H	Half-precision	16	V
11	Q	Quad-precision, requires RV64IFD	128	Q

Table 12.4: Floating-Point Formats.

The general format for many of the floating-point opcodes is as follows:

```
fopcode.{S|D|H|Q} rd,rs1,rs2[,rm]
fopcode.{S|D|H|Q} rd,rs1[,rm]
```

The braces show a choice for format (Table 12.4), while the square brackets show an optional rounding mode (Table 12.2). For example:

```
fadd.s  fa0,ft1,ft2,rmm      # add single-precision, round nearest
fsqrt.d ft1,fa0,rup          # sqrt double-precision, round up
```

When the rounding mode is unspecified or given as "dyn", the rounding mode used is determined by `fcsr.frm`.

12.6. Load and Store

In order to load floating-point values directly from memory, or to store the same, the following opcodes are used. X must be one of W, D or Q from Table 12.4:

```
fLX    rd,imm(rs1)          # rd = load imm(rs1)
fsX    rs1,imm(rs2)         # store imm(rs2) = rs1
```

For example, if register `a1` contains the pointer to a double-precision value, then register `fa2` can be loaded as follows:

```
fld    fa2,0(a1)            # fa2 = load @ a1
```

12.7. Floating Computation

In the following basic floating-point opcodes, the format "F" is chosen to be one of the values S, D, H or Q (Table 12.4). For these opcodes, an optional rounding mode can be added as the last parameter (Table 12.2):

```
fadd.F  rd,rs1,rs2          # rd = rs1 + rs2
fsub.F  rd,rs1,rs2          # rd = rs1 - rs2
fmul.F  rd,rs1,rs2          # rd = rs1 * rs2
fdiv.F  rd,rs1,rs2          # rd = rs1 / rs2
fmin.F  rd,rs1,rs2          # rd = min(rs1,rs2)
fmax.F  rd,rs1,rs2          # rd = max(rs1,rs2)
fsqrt.F rd,rs1               # rd = square root of rs1
```

For example, the following divides `fa0` by `ft0` using rounding mode `rmm`, placing the result into `fa2`:

```
fdiv.d  fa2,fa0,ft0,rmm
```

In addition to these, RISC-V provides "fused multiply-add" operations, which require a third operand `rs3` (an optional rounding mode may be added from Table 12.2):

```
fmuladd.F rd,rs1,rs2,rs3    # rd = rs1 * rs2 + rs3
fmulsub.F rd,rs1,rs2,rs3    # rd = rs1 * rs2 - rs3
fnmulsub.F rd,rs1,rs2,rs3   # rd = -rs1 * rs2 + rs3
```

```
fmuladd.F rd,rs1,rs2,rs3 # rd = -rs1 * rs2 - rs3
```

For example, rounding towards zero:

```
fmuladd.s ft2,fa0,fa1,ft3,rtz # ft2 = fa0 * fa1 + ft3
```

12.8. Conversion Operations

Opcodes for hardware conversions to and from floating-point values are also provided. The integer format must be one of W, WU, L or LU, as listed in Table 12.5.

Mnemonic	Meaning
W	32-bit signed word
WU	32-bit unsigned word
L	64-bit signed word
LU	64-bit unsigned word

Table 12.5: Integer Formats.

The general format of the "fcvt" opcode is as follows, where F is one of H, S, D, or Q (Table 12.4), and "int" is from Table 12.5. An optional rounding mode may also be added from Table 12.2:

```
fcvt.int.F rd,rs1[,rm] # Convert from integer -> float
fcvt.F.int rd,rs1[,rm] # Convert from float -> integer
```

For example, convert from unsigned integer register a0 to a single-precision floating-point value in ft0:

```
fcvt.wu.s ft0,a0 # ft0 = float(a0), single-precision from 32-bit a0
```

Another example converting from a single-precision floating-point value in ft4 to a 32-bit signed word in integer a0:

```
fcvt.s.w a0,ft4 # a0 = int32(ft4), from single-precision ft4
```

12.8.1. Floating-Point Zero

Unlike the integer register x0, there is no dedicated zero register for floating point. To create a zero in a floating-point register, simply use: one of the following:

```
fcvt.s.w rd,x0 # Set single-precision fp register rd to 0.0
fcvt.d.l rd,x0 # Set double-precision fp register rd to 0.0
```

These two examples of producing a floating-point zero will never raise an exception.

12.8.2. Conversion Failures

Conversions from floating-point to integer formats are prone to problems because of the data type's limitations. If after rounding, the value cannot be represented in the destination's format, it is clipped to the nearest value and the NV flag ("invalid" from Table 12.3) is set. This affects opcodes like the following example:

```
fcvt.lu.d  t2,fa3      # Convert from float fa3 -> integer t2
```

Table 12.6 lists the possible results for a failed conversion.

Description	fcvt.w.s	fcvt.wu.s	fcvt.l.s	fcvt..lu.s
Minimum valid input after rounding	-2^{31}	0	-2^{63}	0
Maximum valid input after rounding	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
Output for out-of-range negative input	-2^{31}	0	-2^{63}	0
Output for negative infinity	-2^{31}	0	-2^{63}	0
Output for out-of-range positive input	$2^{32}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
Output for positive infinity or NaN	$2^{32}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$

Table 12.6: Behaviour for invalid inputs to float-to-integer conversions.

12.9. Floating-Point Signs

To ease the programmer's efforts in working with floating-point signs, the following sign injection opcodes are available, where F is one of S, D or Q (Table 12.4):

```
fsgnj.F   rd,rs1,rs2   # rd = |rs1| with sign(rs2)
fsgjn.F   rd,rs1,rs2   # rd = |rs1| with opposite_sign(rs2)
fsgnjx.F  rd,rs1,rs2   # rd = |rs1| with sign(rs1) xor sign(rs2)
```

The following is an example that loads fa0 with the double-precision value of |ft0| but using the sign of ft1:

```
fsgnj.d   fa0,ft0,ft1
```

There are no exception flags raised by these opcodes.

The pair of pseudo-opcodes "fneg" and "fabs" take advantage of the sign injection opcodes, where F is one of S, D or Q (Table 12.4):

```
fneg.F    rx,ry        # equivalent: fsgjn.F rx,ry,ry
fabs.F    rx,ry        # equivalent: fsgnjx.F rx,ry,ry
```

12.10. Floating-Point Move

If the data value is already in IEEE 754-2008 floating-point format it can be copied as is from an integer register to a floating-point register or vice versa. To move these values from a floating-point register (*rs1*) to an integer register (*rd*), use one of these opcodes:

```
mv.x.w rd,rs1      # rd = rs1 single-precision
fmv.x.d rd,rs1     # rd = rs1 double-precision (RV64)
```

To move floating-point *representation* data from an integer register (*rs1*) to a floating-point register (*rd*), use one of these:

```
fmv.w.x rd,rs1     # rd = rs1 single-precision
fmv.d.x rd,rs1     # rd = rs1 double-precision (RV64)
```

Note that the "d" opcode versions are supported by extension RV64 (or larger). These require an integer register width of 64 bits.

Note: The RISC-V specification notes that: "The FMV.W.X and FMV.X.W instructions were previously called FMV.S.X and FMV.X.S. The use of W is more consistent with their semantics as an instruction that moves 32 bits without interpreting them. This became clearer after defining NaN-boxing. To avoid disturbing existing code, both the W and S versions will be supported by tools."^[1]

12.11. Floating-Point Compare

Provision was made for comparing floating-point values by placing the boolean result of the comparison in the destination *integer* register. In the following, F must be S, D, or Q (Table 12.4):

```
feq.F rd,rs1,rs2   # rd = rs1 == rs2
flt.F rd,rs1,rs2   # rd = rs1 < rs2
fle.F rd,rs1,rs2   # rd = rs1 <= rs2
```

In the "flt" and "fle" opcodes, be aware that Invalid Operation (NV) is raised when either input is NaN (since no proper comparison can be made). For "feq", only the *signaling* NaN (sNaN) causes an Invalid Operation (NV) to be raised. All three opcodes return boolean false (zero) when either operand is a NaN value.

Note: The purpose of a Signaling NaN (sNaN) is to cause an exception for debugging (perhaps because of an uninitialized value). A *sNaN is never produced as the result of arithmetic*. Arithmetic may, however, produce a *quiet NaN* value.

12.12. Classify Operation

The floating-point classify operation provides a quick and easy way to classify a floating value in one operation. Replace "F" below with the precision specifier S, D or Q (Table 12.4):

```
fclass.F rd,rs1    # rd = classify(rs1)
```


The destination register `rd` is an *integer* register, which receives the 8 bits illustrated in Table 12.7.

Note: A subnormal number is a non-zero number that is smaller than the smallest number that can be represented in the given precision.

rd Bit	Meaning
0	rs1 is $-\infty$
1	rs1 is a negative normal number
2	rs1 is a negative subnormal number
3	rs1 is -0
4	rs1 is +0
5	rs1 is a positive subnormal number
6	rs1 is a positive normal number
7	rs1 is $+\infty$
8	rs1 is a signaling NaN (sNaN)
9	rs1 is a quiet NaN

Table 12.7: *fclass* result format, by bit number.

12.13. Fahrenheit to Celsius Revisited

Whew! That was a lot of material to cover. Let's now apply what we've learned in converting the original integer-based program to use a floating-point for calculating degrees Celsius from Fahrenheit. By way of review, the formula for the conversion is:

$$^{\circ}\text{C} = (^{\circ}\text{F} - 32) \times \frac{5}{9}$$

Instead of multiplying by 5 and then dividing by 9, we'll just divide by the constant 1.8 instead.

This will be a Fedora Linux project using QEMU where we have hardware floating-point support in extensions F and D. The listing for the assembler portion is provided in Listing 12.1.

```

1 #      The floating-point version of conftemp (fconvtemp)
2      .global fconvtemp
3      .text
4
5      .equ   rtz,0x1           # Round to zero
6      .equ   rmm,0x4         # Round to Nearest
7      .equ   dyn,0x7        # Dynamic rounding mode
8 #
9 #      extern double fconvtemp(double fahrenheit,unsigned *pflags)
10 #
```

```

11 # ARGUMENTS:
12 #     fa0     temperature in Fahrenheit
13 #     a0     pointer to int to return flags
14 #
15 # RETURNS:
16 #     fa0     temperature in Celsius
17 #     flags through ptr in a0
18
19 fconvtemp:
20     frcsr    t2                # t2 = original fcsr
21     fsrmi    x0,rmm           # Set rnd mode to RMM
22     fsflagsi x0,0             # Clear exceptions
23     la      t4,f18
24     fld     ft0,0(t4)         # ft0 = 1.8
25     addi    t0,x0,32          # t0 = 32
26     fcvt.d.lu ft1,t0,rtz      # ft1 = 32.0
27
28 conv:    fsub.d  fa0,fa0,ft1,rtz  # fa0 -= 32.0
29         fdiv.d  fa0,fa0,ft0,rmm  # fa0 /= 1.8
30
31         frflags t0                # t0 = fcsr.flags
32         sw     t0,0(a0)           # Store fcsr.flags
33
34         fscsr  x0,t2              # Restore fcsr
35         ret
36
37         .section .rodata
38 f18:    .double 1.8

```

*Listing 12.1: Floating-point program fconvtemp in
~/riscv/repo/12/celsius/qemu64/celsius.S.*

Let's now examine the breakdown of this code:

1. The function takes two arguments:
 - A. A double value containing the temperature in degrees Fahrenheit. This will arrive in hardware register fa0.
 - B. A pointer to an unsigned int, which will receive the returned fcsr.flags after the computation is completed.
2. Line 20 loads the current value of fcsr into integer register t2. We'll use this value to restore the fcsr when the function returns later.

3. Just in case no rounding mode is specified on the instruction, we set the rounding mode in line 21. We don't care about its prior value so x0 is the destination. We set the default rounding mode to "round to nearest".
4. We are interested in the exception flags, so these are all cleared in line 22 by setting the exceptions register to zero. Again, we don't care about the prior value so the register x0 is used for the destination.
5. We need to access a double constant from line 38, so we establish an address in t4 using the "load address" pseudo-op (line 23).
6. Line 24 loads the double-precision value 1.8 into the floating-point register ft0, for use later on.
7. We establish the integer constant of 32 in temporary register t0 (line 25).
8. Then convert that integer (32) to floating-point in line 26, using "round to zero", just for fun.
9. Line 28 finally gets started on the calculation. It subtracts 32.0 in ft1 from the value passed in fa0, rounding towards zero. The result is returned to fa0.
10. Then fa0 is divided by 1.8 in ft0 to arrive at the temperature in Celsius, rounding to the nearest. The result is returned to fa0, which will be the function's return value.
11. Line 31 then loads the fcsr.flags into temporary register t0. This value is returned to the caller through the pointer passed in integer a0 (recall that the first argument went into fa0 instead).
12. The flags in t0 are passed back to the caller in line 32 by storing a word through the given pointer.
13. The fcsr register is restored to the way we found it, in case the caller (and C/C++) needs it that way in line 34.
14. Finally, the function returns, with the return value passed back in floating-point register fa0 at line 35.

For a simple calculation, that procedure may seem a little tedious. But this is an advantage to consider. If this were a complex and serious scientific calculation, then we've guided the rounding at every step of the way. In C/C++, a rounding mode is selected (likely by default) and used throughout.

Let's now examine the main program in Listing 12.2.

```

1  #include <stdio.h>
2
3  extern double fconvtemp(double f,unsigned *pflags);
4
5  int
6  main(int argc,char **argv) {
7      static double const tests[] = {
8          32.0, 0.0, -40.0, 18.5
9      };
10
11     for ( int ux=0; ux < 4; ++ux ) {
12         unsigned flags;
```

```
13
14         double celsius = fconvtemp(tests[ux],&flags);
15
16         printf("%.1lf F -> %.1lf C (flags = 0x%04X)\n",
17               tests[ux], celsius, flags);
18     }
19     return 0;
20 }
```

Listing 12.2: Main driver program for the Fahrenheit to Celsius conversion in `~/riscv/repo/12/celsius/qemu64/main.c`.

The main program loops through trying every test Fahrenheit temperature in lines 11 to 18. The result of the conversion is reported along with the returned flags. Compile and run it as follows:

```
$ cd ~/riscv/repo/12/celsius/qemu64
$ gcc -g celsius.S main.c
$ ./a.out
32.0 F -> 0.0 C (flags = 0x0000)
0.0 F -> -17.8 C (flags = 0x0001)
-40.0 F -> -40.0 C (flags = 0x0001)
18.5 F -> -7.5 C (flags = 0x0001)
```

Yes, -40°F is the same as -40°C (it's one of a few favourite things I memorized). Note that the flag's value for the first conversion was zero, indicating that there were no exceptions raised. The flag's value of `0x0001` for the remaining calculations indicates that the NX exception flag was set. This simply means that the computed result was inexact. This is not unusual in floating-point calculations. These exception flags can be very helpful for checking difficult calculations.

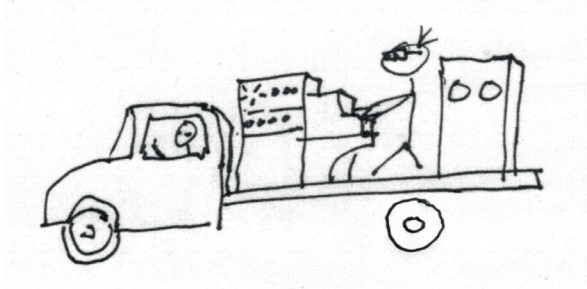
12.14. Summary

Precise floating-point calculations can be tedious to get right. In C/C++, they are often written in a careless fashion without any regard for rounding. In the RISC-V assembler language, you are more likely to pay close attention to each step. This can be helpful for correctly rounding critical formulas. Finally, the hardware operation of these floating calculations is much faster than performing those in software.

Bibliography

- [1] Asanovi, K., & SiFive Inc. (n.d.). In A. Waterman (Ed.), *The RISC-V Instruction Set Manual Volume I: User-Level ISA* (Version 2.2, Vol. I).

Chapter 13 • Portability



Portability Pioneers

Programs, particularly assembler language code can be tedious to write and test for correctness. Portable C/C++ code is relatively straightforward with the use of the CPP (C Pre-Processor) macro capabilities. Sometimes, it is equally desirable to do the same for assembly language code so that only one source module needs to be maintained. In this chapter, let's see what is available to you using the GNU Compiler Collection (GCC).

13.1. C/C++ Pre-Processor

You might recall that the reason we use the capital ".S" suffix for our assembly language source code in this book is to take advantage of the CPP macro capabilities. The alternative is the lowercase ".s" file suffix but then you give up the preprocessing.

The first question that follows is "what macros are available?" Because of the large number of them, you'll find yourself referring to the source code rather than a specific document. You might say that the macros are self-documenting. But these are inconveniently defined in several files, so it is more convenient to ask the compiler to list them instead. On POSIX systems like Linux/MacOS/*BSD, you can ask for this list from GCC directly as follows:

```
$ gcc -dM -E - </dev/null
```

This dumps all predefined GCC macros. For example, the following is a dump of the first few:

```
$ gcc -dM -E - </dev/null | head
#define __riscv 1
#define __DBL_MIN_EXP__ (-1021)
#define __FLT32X_MAX_EXP__ 1024
#define __UINT_LEAST16_MAX__ 0xffff
#define __ATOMIC_ACQUIRE 2
#define __FLT128_MAX_10_EXP__ 4932
#define __FLT_MIN__ 1.17549435082228750796873653722224568e-38F
#define __GCC_IEC_559_COMPLEX 0
#define __UINT_LEAST8_TYPE__ unsigned char
#define __INTMAX_C(c) c ## LL
```

The compiler options and arguments used here are described in Table 13.1. The definitions are directed to standard output, which is convenient for piping to `grep` or other filter commands.

Option/Argument	Description
<code>-dM</code>	Dump preprocessing macro definitions
<code>-E</code>	Stop at the end of the preprocessing (do not compile)
<code>-</code>	Input will come from standard input (stdin)
<code></dev/null</code>	Standard input is redirected from <code>/dev/null</code>

Table 13.1: GCC Compiler Options for dumping out macro definitions.

Macros relevant to RISC-V are listed in Table 13.2. For example, when the macro `__riscv` is defined, then you know that the compile is targeted to the RISC-V instruction set.

Macro Name	Description
<code>__riscv</code>	Defined for any RISC-V target.
<code>__riscv_xlen</code>	Defined as 32 for RV32, 64 for RV64 etc.
<code>__riscv_float_abi_soft</code> , <code>__riscv_float_abi_single</code> , <code>__riscv_float_abi_double</code>	One of these three will be defined, depending on target ABI.
<code>__riscv_cmodel_medlow</code> , <code>__riscv_cmodel_medany</code>	One of these two will be defined, depending on the target code model.
<code>__riscv_mul</code>	Defined when 'M' ISA extension is the target.
<code>__riscv_muldiv</code>	Defined when targeting the 'M' ISA extension and option <code>mno-div</code> has not been used.
<code>__riscv_div</code>	Defined when targeting the 'M' ISA extension and <code>mno-div</code> has not been used.
<code>__riscv_atomic</code>	Defined when targeting the 'A' ISA extension (atomics).
<code>__riscv_flen</code>	Defined as 32 when targeting the 'F' ISA extension (but not 'D'), or 64 when targeting 'FD' instead.
<code>__riscv_fdiv</code>	Defined when targeting the 'F' or 'D' ISA extensions and <code>mno-fdiv</code> has not been used.
<code>__riscv_fsqrt</code>	Defined when targeting the 'F' or 'D' ISA extensions and <code>mno-fdiv</code> has not been used.
<code>__riscv_compressed</code>	Defined when targeting the 'C' ISA extension.

Table 13.2: RISC-V relevant CPP macros.

Some of the macros depend upon the compiler options used. The two options that are referenced in Table 13.2 are described in Table 13.3.

Option	Meaning
<code>-mdiv</code> or <code>-mno-div</code>	Determines if the hardware instructions for division should be used/not-used. The default is to use them if they are defined for that architecture. This requires the RISC-V ISA 'M' extension.
<code>-mfdiv</code> or <code>-mno-fdiv</code>	Determines if the hardware floating-point divide or square-root instructions should be used/not-used. These require the RISC-V 'F' or 'D' ISA extensions. The default is to use them if the architecture supports them.

Table 13.3: Compiler options affecting RISC-V macro definitions.

13.2. Testing for RISC-V Architecture

Probably the most basic of these macros is whether we are assembling/compiling for the RISC-V architecture at all, or some other platform. If you want to force an error in assembling your RISC-V assembly program, you could do something like the example in Listing 13.1.

```

1      .global      foo
2      .text
3
4      #ifndef __RISCV
5      #error This is not a RISCV architecture!
6      #endif
7      foo:      ret

```

Listing 13.1: Forcing an assembly error for non-RISC-V platforms.

In Listing 13.1 the CPP `#ifndef` is used with the macro `__RISCV`. When the macro value for `__RISCV` is *undefined*, the `#error` message "This is not a RISC-V architecture!" will be issued and the build will stop.

13.3. Testing For Integer Multiplication

Say you wanted to write a universal RISC-V version of our earlier `struint()` function from Chapter 11 Addressing and Subscripting, but you wanted to use the hardware integer multiply opcode when it was available else fall back to the three-step software procedure instead. Listing 13.2 demonstrates how this can be accomplished:

```

1      .global struint
2      .text
3
4      #      extern unsigned struint(char const *text, bool *ok)
5      #
6      # ARGUMENTS:
7      #      a0      char const *text (text to convert)
8      #      a1      pointer to bool
9      #

```

```
10 # RETURNS:
11 #     a0     unsigned value (when ok is true)
12 #     ok:
13 #           true, conversion successful
14 #           false, conversion failed
15
16 struint:
17     mv     t6,a0         # t6 = ptr to test
18     li     a0,0         # Accumulator for uint
19     li     t2,0         # Digit count
20     li     t4,'0'
21     li     t3,'9'
22     li     t1,' '
23     li     t0,10
24
25 loop:  lbu     t5,0(t6)   # Load text char
26         beqz   t5,nulbyt # Branch if null byte
27         beq    t5,t1,skip # Skip white space
28         bgt    t5,t3,fail # char > '9'?
29         blt    t5,t4,fail # char < '0'?
30         andi   t5,t5,0x0F # Mask out 0x00 to 0x09
31 #ifdef __riscv_mul
32     mul   a0,a0,t0     # a0 *= 10
33 #else
34     sll   a6,a0,3      # a6 = a0 * 8
35     sll   a0,a0,1      # a0 *= 2
36     add   a0,a0,a6     # a0 += a6
37 #endif
38     add   a0,a0,t5     # a0 += t5
39     addi  t2,t2,1      # Bump digit count
40 skip:  addi  t6,t6,1    # ++text ptr
41     j     loop
42
43 nulbyt: beqz   t2,fail   # Fail if no digits
44     li   t0,1
45     sb   t0,0(a1)     # ok = true
46     ret
47
48 fail:  li   t0,0
49 exit:  sb   t0,0(a1)   # ok = false
50     ret
```

Listing 13.2: The universal RISC-V version of struint() function.

Line 31 uses the C Pre-Processor to test the macro name `__riscv_mul`. If the macro is defined, then the source at line 32 invokes the hardware multiply opcode. Otherwise, the assembly falls back to lines 38 to 36 to perform that multiplication by ten in three steps instead. We can test this change by producing a compiler listing.

```
$ cd ~/riscv/repo/13/cpp
$ ~/riscv/repo/listesp struint.S           # For ESP32-C3 environment
$ ~/riscv/repo/list struint.S            # For QEMU in Fedora Linux
$ C:\riscv\repo\listesp.bat struint.S    # For Windows ESP32-C3
```

Once the listing is produced, look at lines 25 to 41. Notice how there is assembled opcode data for line 32 specifying the `mul` instruction, but no code for lines 33 to 36.

```
25 0014 03CF0F00    loop:  lbu     t5,0(t6)      # Load text char
26 0018 63000F02          beqz    t5,nulbyt      # Branch if null byte
27 001c 630C6F00          beq     t5,t1,skip     # Skip white space
28 0020 6342EE03          bgt     t5,t3,fail     # char > '9'?
29 0024 6340DF03          blt     t5,t4,fail     # char < '0'?
30 0028 137FFF00          andi    t5,t5,0x0F     # Mask out 0x00 to 0x09
31                                     #ifdef __riscv_mul
32 002c 33055502          mul     a0,a0,t0       # a0 *= 10
33                                     #else
34                                     sll     a6,a0,3        # a6 = a0 * 8
35                                     sll     a0,a0,1        # a0 *= 2
36                                     add     a0,a0,a6       # a0 += a6
37                                     #endif
38 0030 7A95            add     a0,a0,t5       # a0 += t5
39 0032 8503            addi    t2,t2,1        # Bump digit count
40 0034 850F            skip:  addi    t6,t6,1    # ++text ptr
41 0036 F9BF            j       loop
```

Listing 13.3: Listing output of the main loop for `struint.S`.

Now try removing extension support 'M'. Under Fedora, specify `-march=rv64iac` and for the ESP32-C3 environment, use `-march=rv32iac` instead. This compiles *without* the 'M' extension. Listing 13.4 illustrates what the ESP32-C3 listing would look like after the assembly:

```
$ cd ~/riscv/repo/13/cpp
$ ~/riscv/repo/listesp -march=rv32iac struint.S           # For ESP32-C3 environment
```

```
25 0014 03CF0F00    loop:  lbu     t5,0(t6)      # Load text char
26 0018 63020F02          beqz    t5,nulbyt      # Branch if null byte
27 001c 630E6F00          beq     t5,t1,skip     # Skip white space
28 0020 6344EE03          bgt     t5,t3,fail     # char > '9'?
29 0024 6342DF03          blt     t5,t4,fail     # char < '0'?
30 0028 137FFF00          andi    t5,t5,0x0F     # Mask out 0x00 to 0x09
```

```

31             #ifdef __riscv_mul
32             mul    a0,a0,t0        # a0 *= 10
33             #else
34 002c 13183500    sll    a6,a0,3      # a6 = a0 * 8
35 0030 0605      sll    a0,a0,1      # a0 *= 2
36 0032 4295      add    a0,a0,a6     # a0 += a6
37             #endif
38 0034 7A95      add    a0,a0,t5     # a0 += t5
39 0036 8503      addi   t2,t2,1      # Bump digit count
40 0038 850F      skip: addi   t6,t6,1    # ++text ptr
41 003a E9BF      j      loop

```

Listing 13.4 Compiling with `-march=rv32iac` for `struint.S` for ESP32-C3 environment.

Notice in Listing 13.4 how there is no code assembled for line 32, but there is for lines 34 to 36. When we take away the extension 'M', the assembler substitutes the software solution for us.

13.4. RV32 vs RV64

There may be times when you want to make an assembler routine portable to both the 32-bit and 64-bit environments. In many cases, you can just leverage the fact that the sign-extend feature of many opcodes to the full width of the 64-bit registers in RV64 allows you to code it the same way. For example, the first integer argument will arrive in register a0, whether RV32 or RV64. A second integer argument likewise will arrive in a1. If the goal was to multiply the two arguments and return an int, the code can remain the same for both environments:

```
mul    a0,a0,a1        # return a0 * a1
```

But when you need to return a 64-bit result in the RV32 environment, you have to split the returned results into a0 and a1, with a0 holding the low order word.

```
mulh   t2,a0,a1      # high order a0 * a1 into t2
mul    a0,a0,a1      # low order a0 * a1
mv     a1,t2         # a1 = high order a0 * a1
```

But for RV64, the product can be fully returned in a0 because the register holds 64 bits:

```
mul    a0,a0,a1        # return a0 * a1
```

To work around this, preprocessor statements can help:

```

#if __riscv_xlen == 32
mulh   t2,a0,a1      # high order a0 * a1 into t2
mul    a0,a0,a1      # low order a0 * a1
mv     a1,t2         # a1 = high order a0 * a1

```

```

#elif __riscv_xlen == 64
    mul  a0,a0,a1          # return a0 * a1
#else
#error The budgie died: wrong RISC-V environment
#endif

```

Let's examine the listings for RV32 and RV64, to see if the assembler did the correct thing. Listing 13.5 is the ESP32-C3 listing and Listing 13.6 is the QEMU Fedora Linux assembler listing for the RV64 environment.

```

1          # 1 "port3264.S"
1          .global      foo
0
0
2          .text
3
4          foo:
5          #if __riscv_xlen == 32
6 0000 B313B502      mulh  t2,a0,a1 # high order a0 * a1 into t2
7 0004 3305B502      mul   a0,a0,a1 # low order a0 * a1
8 0008 9E85          mv    a1,t2    # a1 = high order a0 * a1
9          #elif __riscv_xlen == 64
10         mul  a0,a0,a1  # return a0 * a1
11         #else
12         #error The budgie died: wrong RISC-V environment
13         #endif
14 000a 8280          ret

```

Listing 13.5: The assembler listing for ~/riscv/repo/13/cpp/port3264.S for RV32.

```

1          # 1 "port3264.S"
1          .global      foo
0
0
1          /* Copyright (C) 1991-2020 Free Software Foundation, Inc.
2          .text
3
4          foo:
5          #if __riscv_xlen == 32
6          mulh  t2,a0,a1 # high order a0 * a1 into t2
7          mul   a0,a0,a1 # low order a0 * a1
8          mv    a1,t2    # a1 = high order a0 * a1
9          #elif __riscv_xlen == 64
10 0000 3305B502      mul   a0,a0,a1 # return a0 * a1
11         #else
12         #error The budgie died: wrong RISC-V environment

```

```

13             #endif
14 0004 8280             ret

```

Listing 13.6: The assembler listing for `~/riscv/repo/13/cpp/port3264.S` for RV64.

Comparing the two listings, you can see that for the RV32 environment, lines 6 through 8 are assembled, but not line 10. In the RV64 listing, lines 6 through 8 are not assembled, yet line 10 is. The preprocessor allowed the code to adapt to its environment.

13.5. Assembler Macros

The GNU assembler possesses a macro processor that can be helpful, especially if you need to code something repetitively or with some variation in parameters. The basic format of a macro is as follows:

```

.macro      macname parm1=default1 parm2=default2...
opcode     \parm1,\parm2             # Statement(s)
etc...     # etc..
.endm      # End of macro

```

where:

- `macname` is the required unique name of the macro
- `parm1` is the first optional parameter
- `default1` is the default for `parm1` (when specified)
- `parm2` is the second optional parameter
- etc.

Let's take the last assignment from chapter 12 and define and use a macro named `fpinit`, to save the current floating-point state and to initialize the environment for a new calculation. The definition of the macro has been extracted for ease of reference:

```

8 #
9 #     Macro to save current fcsr status to register 'save',
10 #     and set the default rounding mode to 'round' (round to
11 #     nearest, by default), and clear exceptions:
12 #
13     .macro  fpinit save=t2 round=0x4
14 #if __riscv_flen > 0
15     frcsr  \save             # save = original fcsr
16     fsrmi  x0,\round        # Set rnd mode to RMM
17     fsflagsi x0,0          # Clear exceptions
18 #else
19 #error No RISC-V hardware floating point support
20 #endif
21     .endm

```

Line 13 declares the start of the macro body, declaring its name to be `fpinit` and to take two optional parameters `save` and `round`. Each of these parameters has default values of `t2` for `save` and `0x4` for `round` (which is the value for `round` to the nearest). Specifying defaults for each parameter is optional.

Line 14 checks that the hardware-floating point exists, and if not, the error at line 19 is highlighted. Otherwise, lines 15 to 17 are expanded, with the text for `"\save"` and `"\round"` substituted according to the values supplied in the macro parameters.

Listing 13.7 illustrates the entire program for `celsius.S` where the macro is defined and used. The definition of the macro must appear before its use.

```

1  #      The floating-point version of conftemp (fconvtemp)
2      .global fconvtemp
3      .text
4
5      .equ   rtz,0x1           # Round to zero
6      .equ   rmm,0x4          # Round to Nearest
7      .equ   dyn,0x7          # Dynamic rounding mode
8  #
9  #      Macro to save current fcsr status to register 'save',
10 #      and set the default rounding mode to 'round' (round to
11 #      nearest, by default), and clear exceptions:
12 #
13      .macro fpinit save=t2 round=0x4
14 #if __riscv_flen > 0
15          frcsr  \save          # save = original fcsr
16          fsrmi  x0,\round      # Set rnd mode to RMM
17          fsflagsi x0,0        # Clear exceptions
18 #else
19 #error No RISC-V hardware floating point support
20 #endif
21      .endm
22
23 #
24 #      extern double fconvtemp(double ahrenheit,unsigned *pflags)
25 #
26 # ARGUMENTS:
27 #      fa0      temperature in Fahrenheit
28 #      a0      pointer to int to return flags
29 #
30 # RETURNS:
31 #      fa0      temperature in Celsius
32 #      flags through ptr in a0
33
34 fconvtemp:

```

```

35      fpinit  save=a7 round=rmm
36      la      t4,f18
37      fld     ft0,0(t4)           # ft0 = 1.8
38      addi    t0,x0,32           # t0 = 32
39      fcvt.d.lu ft1,t0,rtz       # ft1 = 32.0
40
41 conv: fsub.d  fa0,fa0,ft1,rtz   # fa0 -= 32.0
42      fdiv.d  fa0,fa0,ft0,rmm   # fa0 /= 1.8
43
44      frflags t0                 # t0 = fcsr.flags
45      sw      t0,0(a0)          # Store fcsr.flags
46
47      fscsr   x0,a7             # Restore fcsr
48      ret
49
50      .section .rodata
51 f18:  .double 1.8

```

Listing 13.7: Program using macro `fpinit`, `~/riscv/repo/13/cpp/qemu64/celsius.S`.

We see that the macro is declared at the top of the source file on lines 13 to 21. The macro is invoked in line 35, with the same parameter set to `a7`, and the `round` parameter set to `rmm`. The excerpted assembler output listing of the main section of code is illustrated in Listing 13.8.

```

34      fconvtemp:
35 0000 F3283000          fpinit  save=a7 round=rmm
35      73502200
35      73501000
36 000c 970E0000          la      t4,f18
36      938E0E00
37 0014 07B00E00          fld     ft0,0(t4)           # ft0 = 1.8
38 0018 93020002          addi    t0,x0,32           # t0 = 32
39 001c D39032D2          fcvt.d.lu ft1,t0,rtz       # ft1 = 32.0
40
41 0020 5315150A      conv: fsub.d  fa0,fa0,ft1,rtz   # fa0 -= 32.0
42 0024 5345051A          fdiv.d  fa0,fa0,ft0,rmm   # fa0 /= 1.8
43
44 0028 F3221000          frflags t0                 # t0 = fcsr.flags
45 002c 23205500          sw      t0,0(a0)          # Store fcsr.
flags
46
47 0030 73903800          fscsr   x0,a7             # Restore fcsr
48 0034 8280              ret

```

Listing 13.8: Excerpt of assembler listing for `~/riscv/repo/13/cpp/qemu64/celsius.S`.

The macro invocation occurs in line 35, from which you can see the assembled code at the left in lines labeled as 35 (all three of them). Had we set the parameter `save=t2`, the code shown would have been identical to the program in chapter 12, Floating-Point. Since the macro permits us to use a different register, we chose unused register `a7` this time. Note that this change requires us to change the register referenced in line 47.

Line 35 supplied the value of `round=rmm` to the macro. The default for this parameter was the value `0x4`, which is the bit pattern for round to the nearest. Since we supplied "rmm" to this parameter and it is used as immediate data in line 16, the value "rmm" must be a defined symbol. In this program, that value comes from line 6 of the source program:

```
.equ    rmm,0x4                # Round to Nearest
```

These are little details that sometimes mess us up. If we were lazy and didn't want to define `rmm`, we could have invoked the macro in either of the following ways:

```
fpinit save=a7 round=0x4      # Specify numeric value for rounding
fpinit save=a7                # Depend upon default for round=0x4
```

While this is a fairly simple macro with limited utility in this case, the idea was to introduce to you the ability of the GNU assembler to apply macros. For completeness, Listing 13.9 illustrates the main driver program for this particular project.

```
1  #include <stdio.h>
2
3  extern double fconvtemp(double f,unsigned *pflags);
4
5  int
6  main(int argc,char **argv) {
7      static double const tests[] = {
8          32.0, 0.0, -40.0, 18.5
9      };
10
11     for ( int ux=0; ux < 4; ++ux ) {
12         unsigned flags;
13
14         double celsius = fconvtemp(tests[ux],&flags);
15
16         printf("%.1lf F -> %.1lf C (flags = 0x%04X)\n",
17             tests[ux], celsius, flags);
18     }
19     return 0;
20 }
```

Listing 13.9: Main driver program ~/riscv/repo/13/cpp/qemu64/main.c.

Let's make sure that this change to use the `fpinit` macro produces the same results. Start up Fedora Linux in QEMU, and perform the following:

```
$ cd ~/riscv/repo/13/cpp/qemu64
$ gcc -g celsius.S main.c
$ ./a.out
32.0 F -> 0.0 C (flags = 0x0000)
0.0 F -> -17.8 C (flags = 0x0001)
-40.0 F -> -40.0 C (flags = 0x0001)
18.5 F -> -7.5 C (flags = 0x0001)
```

The results look good!

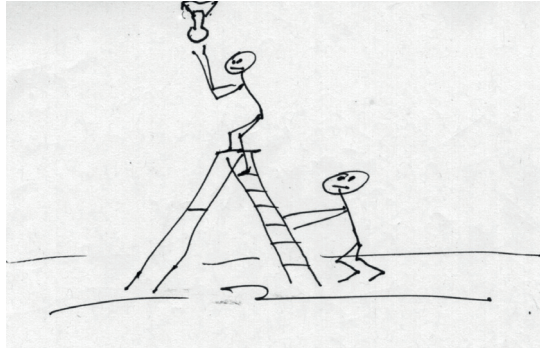
13.6. Summary

Using the C preprocessor macros in assembler work the same as they do for C/C++ when the source code is provided in a ".S" suffixed file. This permits the programmer to perform clever work-arounds to adjust for environmental differences. Whether you use this feature or not will depend upon the complexity involved. Sometimes it may be simpler to use separate source files for different environments because a source file with too many preprocessor directives can be difficult to read.

When developing a more involved function, it may be profitable to develop separately for RV32 and RV64 at first. Once both are debugged and tested, you might try to merge the two by careful use of preprocessing. If that still proves difficult the other option is to `#if` out large unique sections of the code for each environment and share the other sections in common. Do keep the source code pleasant for the reader.

Additionally, the GNU assembler provides macro capabilities that are especially helpful for repetitive or error-prone code. The GCC preprocessor and macro processor offer another pair of tools for the RISC-V developer to exploit.

Chapter 14 • Determining Support



Trusted Support

In the last chapter, C preprocessor macros helped shape the assembly of RISC-V code by use of macros to direct it. For example, if it was known at build time that multiply support was provided then the multiply opcode would be assembled. But what do you do when you must assemble a binary that will run on platforms that may or may not have multiply support? If you can't determine the support level at compile time, then how do you determine it at runtime? That is one exploration that lies before us in this chapter.

Another area that will be touched upon is the CPU's counters and timers. RISC-V defines some standard ones that should exist. There may also be custom timers and counters provided, which are obviously defined by the vendor.

14.1. Privilege Levels

Before we get started, let's introduce RISC-V privilege levels. You will discover that on the ESP32-C3 that you can run almost any defined opcode. Yet while running under QEMU emulating Fedora Linux, that is not possible. Some opcodes require a particular *operating privilege mode* and will cause an exception if this is violated. Table 14.1 lists the defined RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	0b00	User/Application	U
1	0b01	Supervisor	S
2	0b10	<i>Reserved</i>	-
3	0b11	Machine	M

Table 14.1: RISC-V privilege levels.

For a given hardware platform, the vendor may support a subset of these privilege levels. The one mandatory level for all platforms is, however, the Machine Level. Level 2 in Table 14.1 may be defined as Hypervisor in some early documents. But some of these aspects of RISC-V are still undergoing development.

14.1.1. Machine Level

Machine Level (M) is the highest privilege level and is the mode that must be supported by all RISC-V hardware platforms. Code executing at Machine Level is inherently trusted and has access to all low-level aspects of "the machine".

14.1.2. Supervisor Level

The Supervisor Level (S) is used by conventional operating systems like Linux for operating system-level operations, which must be protected from the application-level code.

14.1.3. User Level

Applications in a conventional operating system such as Linux, run in User Level (U) mode so that problems in the application are prevented from causing issues for other applications on the same system. When special operations are requested, the application will enter Supervisor Level (S) through a standard interface.

14.2. Control and Status Registers

Now let's examine an important register: The Control and Status Register (CSR). The RISC-V ISA has assigned up to 4,096 CSRs with the use of a 12-bit encoded address. There are several addresses available for each of the privilege levels supported. Some registers are read-only while others may be read/write. In this chapter, we will initially examine the Machine ISA Register (misa), which is available at address 0x301.

14.2.1. Machine ISA Register

The misa is a read-only register of XLEN bits in length, which reports the ISA and extensions supported by this CPU. The specification indicates that:

"This register must be readable in any implementation, but a value of zero can be returned to indicate the misa register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism." [1]

According to the RISC-V standard, the misa register is classed as "Write Any Values, Read Legal Values" (WARL) type of register. This means that no exceptions are raised if you write values in unsupported bits of the register. When reading the register, only the legal supported bit values are returned. The layout of the misa CSR is illustrated in Figure 14.1.

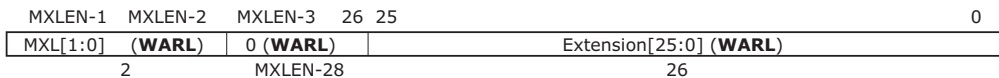


Figure 14.1: The MISA Control Status Register (address 0x301).

Notice that the leftmost 2 bits encode the MXL field, while extensions are the rightmost 26 bits. This has an impact when working with 32-bit or 64-bit machines. The XLEN value is encoded in the left 2 most significant bits according to Table 14.2.

MXL	XLEN
1	32
2	64
3	128

Table 14.2: Encoding of the MXL field within MISA.

The bit encodings for the instruction sets and extensions are shown in Table 14.3. Some of these may change in meaning as the standards are ratified over time. Most notable is bit 8 which indicates one of the RV32I, RV64I or RV128I base ISA, which combined with the MXL value identifies your base platform. If bit 8 is not set, then check bit 4 for the special RV32E base ISA, reserved for very small-embedded platforms. In addition, extension bits will appear when they apply. For example, the ESP32-C3 will also indicate bit 2 (C) for compressed instruction set support and bit 12 (M) for multiply support.

Bit	Character	Extension
0	A	Atomic
1	B	Bit-Manipulation
2	C	Compressed
3	D	Double-precision floating-point
4	E	RV32E base ISA
5	F	Single-precision floating-point
6	G	Reserved
7	H	Hypervisor
8	I	RV32I/RV64I/RV128I base ISA
9	J	Tentatively reserved for Dynamically Translated Languages extension
10	K	Reserved
11	L	Reserved
12	M	Integer Multiply
13	N	Tentatively reserved for User Level Interrupts
14	O	Reserved
15	P	Tentatively reserved for Packed-SIMD
16	Q	Quad-precision floating-point
17	R	Reserved
18	S	Supervisor mode implemented
19	T	Reserved
20	U	User mode implemented
21	V	Vector
22	W	Reserved

Bit	Character	Extension
23	X	Non-standard extensions present
24	Y	Reserved
25	Z	Reserved

Table 14.3: Encoding for the Extensions field in MISA.

14.3. Opcodes

There are three basic opcodes to know about before we look at the pseudo-opcodes. Where "CSR" is specified, supply the Control Status Register address that you want to work with. For one example the address `misa` from Figure 14.1 can be supplied. The generalized opcodes are:

```
csrrs rd, csr, rs1    # rd<-csr, csr<-rs1 Atomic Read and Set Bit in CSR
csrrw rd, csr, rs1    # rd<-csr, csr<-rs1 Atomic Read/Write CSR
csrrwi rd, csr, imm   # rd<-csr, csr<-imm Atomic Read/Write CSR immediate
```

The `csrrw` and `csrrwi` atomically place the CSR contents into `rd`, while replacing the CSR with the value in `rs1` or the immediate data. The `csrrs` opcode differs by only setting bits in the CSR for each bit that is a 1-bit in `rs1`, while zero bits have no effect.

For programmer convenience, there are three pseudo-instructions offered, with the equivalent opcode shown at right in the comment field:

```
csrr  rd, csr        # csrrs rd, csr, x0 (read by do not change)
csrw  csr, rs1       # csrrw x0, csr, rs1 (write by do not read)
csrwi csr, imm       # csrrwi x0, csr, imm (write imm by do not read)
```

To read the `misa` register into register `t3` without changing the value of the `misa` register, you would code:

```
csrr  t3, misa      # t3 <- misa
```

14.4. ESP32-C3

The ESP32-C3 device as programmed by the Espressif ESP-IDF operates in the Machine Level mode, allowing us to explore privileged opcodes like `csrr`. The ESP-IDF incorporates a number of libraries, including FreeRTOS, so that preemptive multi-threading is possible. But all this runs at the "machine level" (M).

14.5. Reporting MISA

To demonstrate reading the `misa` register, we'll run the main program shown in Listing 14.1, which calls upon the assembler function named `extensions()`. This function will return the extensions supported in the return value, and the `XLEN` value by the pointer in the third argument. The buffer and its maximum length are passed in arguments one and two (line 11) to be populated with some text. Upon return, the `printf()` statement will further format and display the result.

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  extern unsigned extensions(char *buf,unsigned bufsiz,unsigned *bits);
5
6  void
7  app_main(void) {
8      char buf[32];
9      unsigned exten=0, bits=0;
10
11     exten = extensions(buf,sizeof buf,&bits);
12
13     printf("exten = 0x%06X, %u bits, RV%u%s\n",
14           exten, bits, bits, buf);
15 }

```

Listing 14.1: Main program ~/riscv/repo/14/extend/main/main.c to report MISA.

The assembler function is illustrated in Listing 14.2. It might look a little complicated, but it is merely a bunch of little steps used to return an improved format to the caller.

```

1      .global extensions
2      .text
3
4  # extern unsigned extensions(
5  #     char *buf,
6  #     unsigned bufsiz,
7  #     unsigned *bits);
8  #
9  # ARGUMENTS:
10 #     a0     char const *buf (text to return)
11 #     a1     unsigned buf size (bytes)
12 #     a2     pointer to unsigned int 'bits'
13 #
14 # RETURNS:
15 #     a0     unsigned value (extension bits)
16
17 extensions:
18     mv     t6,a0         # t6 = buf ptr
19     add   a6,t6,a1      # a6 = buf + buf_size
20     csrr  t3,misa       # t3 = misa register
21     li    t2,1         # t2 = 1
22     sll   t2,t2,8       # t2 <= 8 (Mask for 'I')
23     and   t4,t3,t2      # t4 = t3 & t2
24     beq   x0,t4,3f      # Branch if not 'I'
25     li    t1,'I'       # t1 = ascii 'I'

```

```

26     jal    t0,putch    # Stuff 'I' into buf
27     xor    t3,t3,t4    # Clear 'I' bit
28     #
29     #     Return 32/64/128 bits
30     #
31 3:    li    t4,32      # t4 = 32
32     blt   t3,x0,7f    # Branch if t3 negative
33     j     8f
34 7:    li    t4,64      # t4 = 64
35     slli   t2,t3,1     # t2 = t3 << 1
36     blt   t2,x0,9f    #
37     j     8f
38 9:    li    t4,128     # t4 = 128
39 8:    sw    t4,0(a2)   # Return bits = xlen
40     #
41     #     Now mask out extensions only (exclude 'I')
42     #
43     addi   a4,t4,-26   # a4 = xlen - 26
44     li    a0,-1       # a0 = mask all 1's
45     sll   a0,t3,a4     # a0 = t3 << (xlen - 26)
46     srl   a0,a0,a4     # a0 >>= (xlen - 26)
47     #
48     #     Populate buf with extensions
49     #
50     li    t1,'A'
51     mv    t2,a0        # t2 = a0
52 loop: andi   t5,t2,1   # Bit set?
53     beq   t5,x0,4f    # Branch if zero bit
54     jal   t0,putch    # Put character t1
55 4:    addi   t1,t1,1   # ++t1 (ascii char)
56     srli   t2,t2,1    # t2 >>= 1
57     bne   t2,x0,loop  # Loop until all bits
58     li    t1,0        # Load nul byte
59     jal   t0,putch    # Put nul byte
60 xit:  ret
61     #
62     #     Internal routine: returns via t0
63     #     Put char in t1 to buffer pointed by t6
64     #
65 putch: bgeu   t6,a6,5f # If at end of buf...
66     sb    t1,0(t6)    # else put byte
67     addi   t6,t6,1    # ++ptr in t6
68 5:    jr    t0

```

Listing 14.2: Program to read MISA, `~/riscv/repo/14/extend/main/extend.S`.

Let's first examine the inner function named `putch()` defined in lines 65 to 68.

1. The function is called using `t0` as the "link register". The value in `t0` will be used for returning to the caller.
2. Register `a6` has been initialized as the address past the end of the caller's buffer, which is established in line 19. Register `t6` is used as the working pointer into the caller's buffer.
3. Upon entry to `putch()` in line 67, we first test if the buffer pointer (`t6`) is greater than or equal to the end of the buffer. If it is, control passes to label 5 in line 68, where we promptly return.
4. Otherwise, the byte held in temporary `t1` is stored at the pointer `0(t6)` placing that character into the caller's buffer (line 66).
5. The buffer pointer is then incremented by 1 in line 67.
6. The function returns by the link register `t0` in this case.

This use of the little internal function allows us to blindly call it, knowing that if the caller ever supplies a buffer that is too small, no harm will be done. The bounds of the buffer will always be checked.

The entry of the function is line 18, where the buffer pointer in `a0` is copied to `t6`, and the end pointer is computed in `a6` (an unused argument register). Now let's trace the remaining steps:

1. The `misa` CSR is read in line 20, with the value placed into `t3`. The `misa` is left unchanged.
2. Register `t2` is then loaded with 1, then shifted left 8 bits in lines 21 and 22.
3. That mask value is used in line 23 to set `t4` (line 23).
4. If `t4` is zero (in other words, the 'I' bit in `misa` is not set) the code branches to line 31 (label 3).
5. Otherwise, the `misa` 'I' bit was set, and then the ASCII value for 'I' is loaded into `t1` and then stored in the caller's buffer in lines 25 and 26.
6. To suppress the 'I' from being reported later, we mask out that bit in line 27, by using the xor operation to flip the 1-bit to a zero.
7. Line 31 then sets `t4` to the value 32 as a trial value for `XLEN`.
8. If the sign bit of `t3` is set (negative), then branch to line 34 (label 7).
9. Otherwise, we jump to line 39 from line 33 (`XLEN` is 32).
10. If control passes to line 34 (from line 32), we then set `t4` to 64 as the next trial `XLEN` value.
11. Line 35 shifts `t4` left one bit and tests the sign bit again, branching from line 36 if the bit was set. If the branch to line 38 is taken, the `XLEN` is determined to be 128 (label 9).
12. At line 39, the value for `XLEN` in `t4` is saved to the caller's unsigned int argument, returning the `XLEN` value to the caller.
13. Lines 43 to 46 then mask out the lower 26 bits of the `misa` register, since we want to eliminate all but the lower 26 bits. These bits are left in register `a0` to be the return value containing all extension bits (except for 'I', which we turned off).

14. Line 50 initializes t1 with the ASCII value for 'A' and copies a0 to t2, for the extension bits to be tested.
15. The loop begins at line 52, where we test the low-order bit of t2, placing that bit in t5. If that bit was zero, then the code skips ahead to line 55 (label 4).
16. If the bit tested is a 1 bit, then the ASCII character in t1 is saved to the caller at line 54. The putchar() function will also increment the buffer pointer in t6.
17. Line 55 increments the ASCII character to the next in sequence. The first time around the 'A' will become a 'B'.
18. The extension bits in t2 are shifted right one bit in line 56, and if that shift results in a non-zero value, then we loop back to line 52, from 57.
19. Otherwise, we fall through to line 58, where we load a null byte into t1.
20. Internal function putchar() is called one more time from line 59 to place a null byte at the end of the string.
21. The extensions() function returns at line 60.

Build, flash and monitor the ESP32-C3 as follows:

```
$ cd ~/riscv/repo/14/extend
$ idf.py build
...
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
exten = 0x101004, 32 bits, RV32ICMU
```

From the last line of output (from our main program), we see that it reports an XLEN of 32 bits, and that it is an RV32I platform with CMU extensions. These are all encoded in the "exten = 0x101004" bits that were returned and reported. Note that the 'I' bit was stripped out of this string in the code. With the information returned, the caller can determine at runtime, whether multiply is supported for example.

14.6. RV64 Platform

What happens if we run that same code on an RV64 platform? Start up your Fedora Linux under QEMU, and perform the following?

```
$ ~/riscv/repo/14/extend/qemu64
$ gcc -g -Wall extend.S main.c
$ ./a.out
Illegal instruction (core dumped)
$
```

It appears that the program builds ok, but aborts when run under Fedora Linux. To see why, let's enlist the help of gdb:


```

$ gdb ./a.out
GNU gdb (GDB) Fedora 9.0.50.20191119-2.0.riscv64.fc32
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "riscv64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...
(gdb) r
Starting program: /home/riscv/riscv/repo/14/extend/qemu64/a.out
  glibc-2.30.9000-29.fc32.riscv64
Missing separate debuginfos, use: dnf debuginfo-install
Program received signal SIGILL, Illegal instruction.
extensions () at extend.S:20
20          csrr    t3,misa          # t3 = misa register

```

Normally we'd just analyze a "core file" in gdb after an abort but in recent years it seems that every distribution is doing their best to do away with core files (this drives developers nuts!) That problem is fixable, but each platform has its own rules for configuring core file support. Because of this annoyance, I've cut to the chase and just run the executable a.out directly from gdb instead.

After considerable verbiage, the "(gdb)" prompt appears:

1. At the prompt type "r" + CR to tell gdb to "run" the program.
2. The gdb command then starts the program, and produces a few more messages before reporting the message "Program received signal SIGILL, Illegal instruction. extensions () at extend.S:20". This tells us why ("signal SIGILL, Illegal instruction"), the function ("extensions()") and the module ("extend.S") and the line number where the problem occurred (line 20). Note that to get all of this information, we must have built the project with the debug (-g) compile option.
3. Additionally, gdb reports the source line itself, and it is evident that the Linux kernel did not like us issuing the csrr opcode.

Note: If gdb does not report the source line of the failure, it is because the executable was not compiled and linked with debugging support (GCC option -g).

The last lines shown by gdb are:

```
Program received signal SIGILL, Illegal instruction.
extensions () at extend.S:20
20          csrr    t3,misa          # t3 = misa register
```

There is nothing wrong with the way the program (or the opcode) is written. What caused the above signal is that Fedora Linux is running our programs in "User/Application mode" (U). This mode of execution does not permit privileged instructions like `csrr` to be executed. If you wanted to, you could write a kernel device module, and issue the opcode there. But that is beyond the scope of this book.

To exit gdb, type "q" + CR to quit.

Fortunately, Linux provides another way to query the capabilities:

```
$ cat /proc/cpuinfo
processor      : 0
hart         : 1
isa          : rv64imafdcsu
mmu         : sv48

processor      : 1
hart         : 0
isa          : rv64imafdcsu
mmu         : sv48
```

From this display, we see that there are two configured cores (processors). Each RV64I core with MAFDCSU extensions supported. Table 14.4 lists the support available in our Fedora Linux RISC-V system. A user program can open and parse the file `/proc/cpuinfo` to determine the level of support available.

Bit	Character	Extension
0	A	Atomic
2	C	Compressed
3	D	Double-precision floating-point
5	F	Single-precision floating-point
8	I	RV32I/RV64I/RV128I base ISA
12	M	Integer Multiply
18	S	Supervisor mode implemented
20	U	User mode implemented

Table 14.4: Encoding for the Extensions field in MISA reported by QEMU.

14.7. Counters

The RISC-V standards define performance counters and timers, which are available to the unprivileged mode of operation. These counters are 64 bits in width and are CSR read-only values. For RV32, the upper half of the counter is read with an "h" version of the opcode. For example, the upper word is read with `rdcycleh` instead of `rdcycle`.

The RISC-V standard for reading the cycles counter is the `rdcycle` pseudo-opcode:

```
rdcycle rd          # equivalent: csrr rd,cycle
```

For RV32I, there is the risk of the counter overflowing between the reading of the low order 32 bits and the high order bits. To work around that, use the following sequence:

```
loop: rdcycleh t2
      rdcycle  t1
      rdcycleh t3
      bne     t2,t3,loop
```

This sequence checks to see if the upper 32-bit word changed between the start and end of the sequence. If it did, the sequence is repeated one more time.

There are also two more timers and counters that often interest the programmer:

```
rdtime  rd          # equivalent: csrr rd,time
rdinstret rd        # equivalent: csrr rd,instret
```

For RV32I, there are also `rdtimeh` and `rdinstreth` opcodes. The `rdtime` pseudo-opcode reads the time CSR value into the destination register. This counter tracks the wall-clock real time that has passed from some arbitrary start point (the units used may be implementation specific). The CSR `instret` counts the number of instructions "retired" by this hardware thread.

14.7.1 Project `rdcycle`

To get a feel for the `rdcycle` pseudo-opcode, let's run Fedora Linux under QEMU, to obtain our RV64 environment. In this project, we're going to attempt to measure the difference between using the multiply instruction and using the multiply by ten without the multiply opcode. The main driver program is shown in Listing 14.3.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 extern uint64_t measure(int mul);
5
6 int
7 main(int argc,char **argv) {
8     uint64_t cycles;
```

```

9
10     for ( int x=0; x<10; ++x ) {
11         cycles = measure(1);
12         printf("multiply cycles = %lu\n",cycles);
13         cycles = measure(0);
14         printf("mul10 cycles   = %lu\n",cycles);
15     }
16     return 0;
17 }

```

Listing 14.3: Main program ~/riscv/repo/14/rdcycle/qemu64/main.c.

When the assembler routine `measure()` is called with a true argument, it will measure the cycle time of the multiply opcode. When called with zero (false), the multiply by 10 without the multiply opcode is measured instead. The test is repeated ten times for the pair of calls by the main program (lines 11 and 13).

The source program for the assembler `measure()` function is provided in Listing 14.4.

```

1     .global measure
2     .text
3
4     # extern unsigned measure(int mul)
5     #
6     # ARGUMENTS:
7     #     a0     When true, measure the mul instruction,
8     #           otherwise measure two shifts and one add.
9     # RETURNS:
10    #     a0     unsigned count of cycles
11
12    measure:
13        beqz    a0,1f          # Branch if measuring by 10
14        li     t5,10          # t5 = 10
15        rdcycle t1
16        mul    a2,a0,t5       # a2 = a0 * 10
17        rdcycle t2
18        sub    a0,t2,t1       # Difference in cycles
19        ret
20
21    1:    rdcycle t1
22        sll    a2,a0,3         # a2 = a0 * 8
23        sll    a1,a0,1         # a1 = a0 * 2
24        add    a2,a2,a1       # a2 = a0 * 10
25        rdcycle t2
26        sub    a0,t2,t1       # Difference in cycles
27        ret

```

Listing 14.4: Program ~/riscv/repo/14/rdcycle/qemu64/measure.S.

Let's break down the steps used:

1. The 1 or 0 value is provided in the call through the register a0 and is tested in line 13. The argument was non-zero (true), then the execution falls through to line 14.
2. The constant 10 is loaded into temporary register t5 (line 14).
3. Line 15 takes a cycle counter snapshot into register t1.
4. Then the actual multiplication by the mul opcode occurs in line 16.
5. Followed by that is the second snapshot of the cycle counter into t2 at line 17.
6. The difference between the two counts is computed and returned in a0 (line 18).
7. Control returns to the caller in line 19.
8. When the argument is false, control passes to label "1" at line 21 from line 13. At this point, a cycle snapshot is copied into register t1.
9. Lines 22 through 24 compute a multiply by ten using shifts and add.
10. The second cycle snapshot is captured into t2 at line 25.
11. Finally, the cycle difference is computed in line 26 and returned in register a0.

There is no provision for handling a counter rollover in the presented code. If the cycle counter overflows after capturing the first snapshot, then the difference computed will be huge. This is unlikely to happen depending on how long you have had QEMU running. If it does happen, repeat the test.

First, compile and then run the test, as follows:

```
$ gcc -g measure.S main.c
$ ./a.out
multiply cycles = 18724
mul10 cycles   = 25428
multiply cycles = 19714
mul10 cycles   = 846
multiply cycles = 594
mul10 cycles   = 526
multiply cycles = 604
mul10 cycles   = 504
multiply cycles = 410
mul10 cycles   = 512
multiply cycles = 450
mul10 cycles   = 458
multiply cycles = 496
mul10 cycles   = 678
multiply cycles = 432
mul10 cycles   = 494
multiply cycles = 810
mul10 cycles   = 460
multiply cycles = 432
mul10 cycles   = 454
```

Your results will likely differ, perhaps considerably so. The first three lines indicate a lot of work being done by Fedora Linux, perhaps the result of loading and linking with shared libraries. There is also the overhead of mapping the executable file into memory. So I would throw away at least the first four results.

The remaining results are inconsistent. This is because the QEMU emulation is not a true reflection of what hardware would have returned. So, the best we can do with the emulation run is say "we did it" and leave it at that. The essential lesson from this exercise is how we used the `rdcycle` opcode to measure cycle time.

```
15      rdcycle t1
16      mul      a2,a0,t5      # a2 = a0 * 10
17      rdcycle t2
```

Benchmarking is tricky. This exercise glosses over other issues like the fact that there is no guarantee that the instructions in lines 15 to 17 run without interruption. So, keep these factors in mind when designing benchmarks.

If you have access to real RV64 hardware, you should be able to run this exercise and obtain good results. New affordable hardware announcements for RISC-V are frequently being made these days.

14.7.2. ESP32-C3 `rdcycle` Support

Given that it is difficult to get a consistent experience under QEMU, can we do better with the ESP32C3? At the time of writing, it turns out that the ESP32-C3 does not support the `rdcycle` command. It will "fault" if you try to invoke it. Despite that, the Espressif folks have provided an alternative custom counter that we can experiment with to give us nearly the same experience. They provide the MPCCR counter, which is addressed at `0x7E2`. In our assembler code, we can define the value symbolically as follows:

```
.equ    mpccr,0x7E2
```

The main program, which is similar to the QEMU version is presented in Listing 14.5.

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  extern uint32_t measure(int mul);
5
6  void
7  app_main(void) {
8      uint32_t cycles;
9
10     for ( int x=0; x<10; ++x ) {
11         cycles = measure(1);
12         printf("multiply cycles = %u\n",cycles);
```

```

13         cycles = measure(0);
14         printf("shift   cycles = %u\n",cycles);
15     }
16 }

```

Listing 14.5: Main program ~/riscv/repo/14/rdcycle/main/main.c.

This program runs the `measure()` test multiple times, for reasons that will be explained shortly. The assembler routine for the ESP32-C3 version is shown in Listing 14.6.

```

1     .global measure
2     .text
3
4     .equ    mpccr,0x7E2
5
6     # extern unsigned measure(bool mul)
7     #
8     # ARGUMENTS:
9     #     a0     true = use mul else shift
10    #
11    # RETURNS:
12    #     a0     unsigned count of cycles
13
14    measure:
15        li    a1,99          # Some number
16        beqz  a0,1f          # If mul is false, jump to 1
17        li    a2,10         # ten
18    #
19    #     Multiply by ten with mul opcode
20    #
21        csrr  t1,mpccr
22        mul   t0,a1,a2       # 99 * 10 -> 136 cycles
23        csrr  t3,mpccr
24        j     xit
25    #
26    #     Multiply by ten without mul opcode
27    #
28    1:    csrr  t1,mpccr
29        sll   a2,a1,3        # a2 = a1 * 8
30        sll   a1,a1,1        # a1 *= 2
31        add  a0,a1,a1        # a0 = a1 * 10 => 200 cycles
32        csrr  t3,mpccr
33
34    xit:  sub   a0,t3,t1
35        ret

```

Listing 14.6: Program ~/riscv/repo/14/rdcycle/main/measure.S.

Let's now examine the assembler language listing for `measure()`:

1. Line 4 defines a symbol `mpccr` with the Espressif custom address of `0x7E2`. This permits us to refer to it symbolically in lines 21 and 23 for example.
2. The function begins by defining a value for `a1` in line 15, though this is not really required (we can multiply any number).
3. Line 16 tests the calling argument to see if it is zero or non-zero. If non-zero, the execution falls through to line 17.
4. The value of 10 is loaded into `a2`, to keep the code comparison fair, so that we are multiplying by ten in line 22 (line 17).
5. The first `mpccr` value is captured into `t1` (line 21).
6. The multiply opcode is used in line 22.
7. The second `mpccr` value is captured into `t3` (line 23).
8. Then the code branches from line 24 to line 34, where the `mpccr` difference is computed and returned in register `a0`.
9. If the argument was zero upon calling this routine, control would resume at line 28 where label "1" is defined. At this point, the first `mpccr` counter is captured into `t1`.
10. Then the multiply by ten the hard way is performed in lines 29 through 31.
11. The second `mpccr` value is captured into `t3` at line 32.
12. Finally, the difference in `mpccr` values is computed and returned in register `a0` (line 34) before returning to the caller (line 35).

Once again, note that this code does not handle counters that roll over after overflow. Because your demonstration runs shortly after the CPU reset, you are not likely to see a problem with that.

Let's now build, flash and monitor the program run:

```
$ idf.py build
$ idf.py -p <<<yourport>>> flash monitor
...
I (258) cpu_start: Starting scheduler.
multiply cycles = 64
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
multiply cycles = 2
shift   cycles = 4
```



```
multiply cycles = 2
shift    cycles = 4
multiply cycles = 2
shift    cycles = 4
multiply cycles = 2
shift    cycles = 4
```

Except for the initial surprise, we have better results this time. It is easy to forget that the ESP32C3 code resides in flash memory and must be loaded into the RAM cache before it can execute. This is the reason for the initial high count.

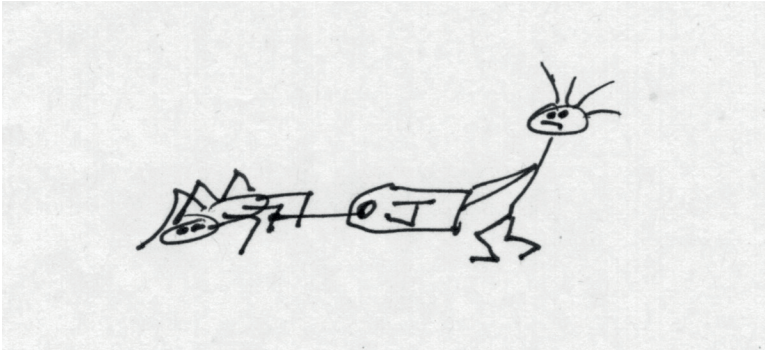
Once the code has entered the cache, we see that the multiply instruction measures consistently 2, while the shift and add return a count of 4. If we assume that this counter represents instructions performed, then we can say that the multiply should be a count of 1, while the shift and add should be 3. Keep in mind an added instruction was needed to perform the second counter capture within `measure()`. The Espressif documentation in the "ESP32C3 Technical Reference Manual" simply describes the MPCCR counter as "Machine Performance Counter Value". Espressif's counter address values are defined in the address space reserved by the RISC-V standard for customization.

14.8. Summary

There is considerably more that could be said about the counters, timers, Control and Status Registers and privilege levels than space would allow. This chapter serves the reader, however, as an introduction to these concepts.

[1] Waterman, A., Krste Asanović, & Hauser, J. (Eds.). (n.d.). *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. Retrieved June 23, 2022.

Chapter 15 • JTAG Debugging



A bug attempting to escape JTAG debugging

Debugging can be difficult at the best of times. Programs today are often long and complicated. When an MCU program suddenly faults, how do you determine the exact cause? Knowing exactly where the fault occurs is a help. But knowing the state of the variables it was working with can make the solution to the problem obvious.

When programming in assembly language the need for a debugger is often more urgent. You can look at the code hundreds of times and still be convinced that it should work. But the evidence demonstrates that it clearly doesn't. If only you were able to step through that same code one instruction at a time looking at the registers and associated memory along the way. That is when you experience that "aha!" moment. Seeing is believing.

JTAG is a standard that was developed to help with testing circuits without the use of the traditional bed-of-nails approach. This required defining new electrical connections and a protocol to drive it. In addition to chip-level testing, the JTAG protocol can be used to program flash memory and debug your device. In this chapter, we will examine the Espressif ESP32-C3 debugging capabilities made available over the USB link.

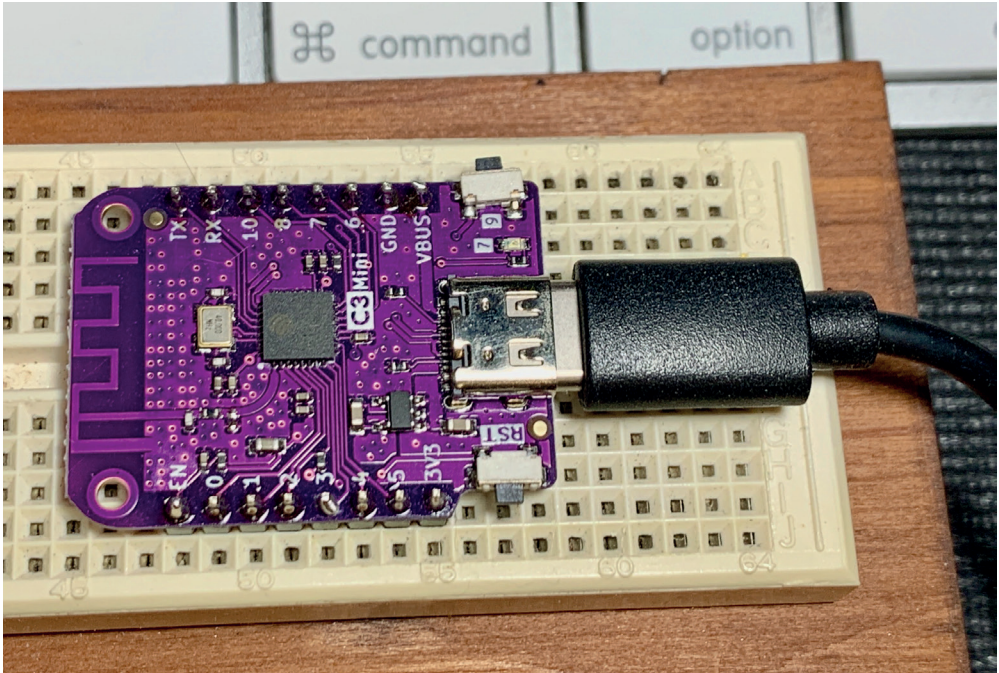
15.1. Espressif JTAG

The ESP32-C3 device can be JTAG debugged using a USB cable connected to the D+/D-USB pins of the ESP32-C3 device. This is especially convenient because it requires no JTAG adapter and requires no extra wiring. According to Espressif's documentation, it is also possible to use a JTAG adapter if you prefer. But this requires a number of things including operation at the 3.3V level and support from the OpenOCD software.[1] Because there are many adapters available and the use of USB is so easy, this chapter will focus on the USB JTAG support, which is already supported by OpenOCD.

15.2. Device Requirements

Unfortunately, not all ESP32-C3 devices will support JTAG. The chip must be revision 3 or newer. Furthermore, the device must be wired directly to the USB D+/D- USB pins, unlike the early devkit versions that used a USB to serial interface chip instead. When shopping for your device, if the PCB lacks the USB to serial chip, then there is a good chance that the

device is revision 3 or later and will support JTAG. Figure 15.1 illustrates one example of a JTAG capable ESP32-C3 dev board.



*Figure 15.1: A JTAG capable ESP32-C3 on a bread board.
Notice the lack of a USB to Serial interface chip.*

15.3. Software Components

Since we'll be using USB to perform JTAG debugging, we can summarize the major software components as follows:

1. The Espressif version of OpenOCD
2. ESP32-C3 (RISC-V) version of gdb
3. Espressif software on the ESP32-C3 device

Figure 15.1 illustrates the relationships between the major components. For debugging, you will have the OpenOCD software running and communicating with the RISC-V aware gdb process, both on your desktop PC. The gdb process will be operated through a PC window. OpenOCD in turn communicates with the ESP32-C3 device over USB, working through a layer of device JTAG support. In this manner, the user on the PC can direct the execution of code running on the ESP32-C3 device.

Optionally, you can have simultaneous USB CDC (Communication Device Class) support so that your ESP32C3 code can send and receive serial data to another window on your desktop. This allows the user to see what the code has "printed" for example. This support is optional and requires some configuration to use.

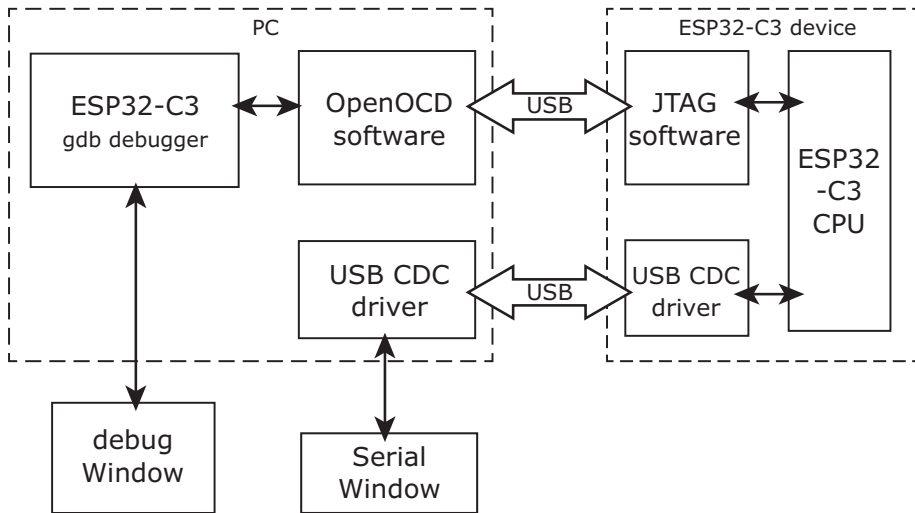


Figure 15.2: Major components of ESP32-C3 JTAG debugging.

If you don't need to see your program output, you can dispense with the second optional (serial) window session and interact only with the gdb debugger.

While Figure 15.2 does not show a window attached to the OpenOCD software, there can be a window used there also. The procedure that I am going to show you will have it run from a terminal session window. If you get fancy and use an IDE like Eclipse or Visual Studio, you can have OpenOCD run automatically for you in the background and in that case won't need a terminal session. I'm going to stick with manual command line methods here, since it leaves us in full control and makes it easier to identify and resolve problems.

15.4. JTAG With No Serial Window

As your first foray into JTAG, I recommend that you start this way and leave the simultaneous serial terminal window for later. This simplifies getting things started and tests your software and hardware setup with the minimum of dependencies.

Using this procedure, the following general steps are:

1. Plug in your ESP32-C3 device USB cable to the PC.
2. Start OpenOCD.
3. Start gdb.

The following subsections will expand on the details for OpenOCD and gdb.

15.4.1. Starting OpenOCD

1. Start a terminal window to run OpenOCD from. Be sure to set up your ESP32-C3 session environment if you're using a Linux/macOS type of session (review section "4 Setup Environment Variables" in chapter 2 if necessary). Windows users can simply click on the Espressif provided CMD icon. The current directory is not critical in this case.
2. Your JTAG-capable ESP32-C3 device is assumed to be plugged into the PC with the USB cable at this point.
3. Enter the OpenOCD command shown in Figure 15.3. You should see messages very similar to the example.

```
$ openocd -f board/esp32c3-builtin.cfg
Open On-Chip Debugger v0.11.0-esp32-20211220 (2021-12-20-15:45)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
Info : esp_usb_jtag: VID set to 0x303a and PID to 0x1001
Info : esp_usb_jtag: capabilities descriptor set to 0x2000
Warn : Transport "jtag" was already selected
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : esp_usb_jtag: Device found. Base speed 40000KHz, div range 1 to 255
Info : clock speed 40000 kHz
Info : JTAG tap: esp32c3.cpu tap/device found: 0x00005c25 (mfg: 0x612 (Espressif
Systems), part: 0x0005, ver: 0x0)
Info : Examined RISC-V core; found 1 harts
Info : hart 0: XLEN=32, misa=0x40101104
Info : starting gdb server for esp32c3 on 3333
Info : Listening on port 3333 for gdb connections
```

Figure 15.3: Typical messages from OpenOCD when it starts up.

Note that you don't need to specify a port because OpenOCD locates the device using USB identifiers. If the device is not plugged in or not recognized, you will see the message:

```
Error: esp_usb_jtag: could not find or open device!
```

What you want to see is the message:

```
Info : Listening on port 3333 for gdb connections
```

since this tells you that OpenOCD is ready to work with a gdb session.

15.4.2. Problems with OpenOCD

Sometimes OpenOCD can get confusing signals from the ESP32-C3 device or encounter USB errors. The best option for recovery is to kill the OpenOCD session and unplug the USB cable. On Linux/macOS systems, you may also need to kill background processes that may still be running:

```
$ ps -ef | grep openocd
 501 76202 1690  0 2:48pm ttys005  0:02.16 openocd -f board/esp32c3-builtin.cfg
$ kill -9 76202
```

Normally it is bad practice to use `kill -9`, but when OpenOCD goes astray, I found it necessary under macOS. Once all instances of OpenOCD have been terminated, plug in the USB cable again and restart OpenOCD.

Note: It is possible for the ESP32-C3 to get into a state that OpenOCD cannot work with because the flashed code has done something bad. If you cannot get OpenOCD to start successfully, try flashing the device with a different project. Test if the OpenOCD and `gdb` can work with *that* project. If so, go back and recheck your code of the problem project. Often adding a 10-second delay at the start of your `app_main()` will give time for OpenOCD to connect. Use the FreeRTOS function `vTaskDelay()`.

15.4.3. Terminating OpenOCD

At some point, you'll want to terminate your OpenOCD access to the device. I recommend that you kill (Control-C) the OpenOCD command before unplugging the USB cable. The current version of the software seems to get upset on macOS when you unplug the USB first. If you already unplugged the USB cable, look for any OpenOCD processes running in the background and issue `kill -9` on any that remain running.

15.4.4. Start `gdb`

With OpenOCD at the ready, you can now start `gdb` if you've already built the project. For this example, we'll use the project at directory `~/riscv/repo/14/rdcycle`. If you've not built and flashed the project, then do so now:

```
$ cd ~/riscv/repo/14/rdcycle
$ idf.py build
$ idf.py -p <<<yourport>>> flash
```

This creates an executable in file `./build/rdcycle.elf`, that will be loaded by `gdb` to match the code in the device flash. To start `gdb`, use the Espressif aid, `idf.py` as follows (it will know where to find your `*.elf` file). You must be in the same directory that you used to build the project:

```
$ idf.py gdb
```

The messages should resemble Figure 14.2. If you see error messages instead, it may be because OpenOCD was not started, or that the process ran into problems. Recheck the OpenOCD startup and try again.

```

$ idf.py gdb
Executing action: gdb
GNU gdb (crosstool-NG esp-2021r2-patch3) 9.2.90.20200913-git
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-host_apple-darwin12
--target=riscv32-esp-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
/Users/ve3wwg/.gdbinit:1: Error in sourced command file:
No symbol table is loaded. Use the "file" command.
0x40000000 in ?? ()
JTAG tap: esp32c3.cpu tap/device found: 0x00005c25 (mfg: 0x612 (Espressif
Systems), part: 0x0005, ver: 0x0)
Hardware assisted breakpoint 1 at 0x42004c38: file /Users/ve3wwg/riscv/repo/14/
rdcycle/main/main.c, line 7.
[New Thread 1070133268]
[New Thread 1070128416]
[Switching to Thread 1070132924]

Thread 1 hit Temporary breakpoint 1, app_main () at /Users/ve3wwg/riscv/repo/14/
rdcycle/main/main.c:7
7   app_main(void) {
(gdb)

```

Figure 14.2: gdb startup messages connecting to OpenOCD.

Notice that gdb has automatically set a breakpoint at `app_main()`, where the ESP32-C3 program starts. To quit debugging, you can type "quit" followed by return.

```
(gdb) quit
A debugging session is active.

    Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /Users/ve3wgg/riscv/repo/14/rdcycle/build/rdcycle.elf,
Remote target
Ending remote debugging.
[Inferior 1 (Remote target) detached]
$
```

If you got that far, then consider it a success! We'll dig into gdb in the next section, so that you can do a whole lot more.

15.5. Operating gdb

The gdb tool is command line driven. But don't fear the command line! GNU has made it easy to use and painless. Like operating the bash/zsh shell in Linux/macOS, gdb supports command line editing. Because of this feature, it is rarely necessary to retype an entire command. GNU provides documentation here [2], but many users may prefer a quick-start read instead.[3] Even if you have never learned any editing control sequences, you can use the arrow-up key to locate a previously entered command. Then using the left and right arrow keys, you can reposition and edit the command before you hit enter.

In addition to command line editing and history, the gdb debugger repeats the last executed command if you just press enter (return). This is extremely useful for some commands like stepping through code one instruction (or statement) at a time.

15.5.1. Abbreviations

The gdb command goes further than editing to make entering commands easy. It allows you to abbreviate commands as short as you want, as long as your abbreviation is unique. For example, the "info" command can be shortened to "inf", "in" or even just "i" at the moment. If GNU eventually adds another "i" command, then the abbreviation "i" will no longer work.

The abbreviations also apply to command arguments. For example:

```
(gdb) info registers
```

can be (at the moment) be shortened to just:

```
(gdb) i r
```

The "r" is accepted as an abbreviation for "registers".

While [4] is an older resource, you might consider it as a gentle introduction to using gdb. In this chapter I am just going to show some highlights to gdb usage, to whet your appetite and get you started. There is a hefty manual for gdb that can be downloaded and printed. See [5] under the section "GDB User Manual (PDF)".

15.5.2 GDB Walkthrough

Rather than listing a lot of documentation for you to digest, let's walk through our rdcycles project example to learn some basics. Let's continue with the rdcycle project from chapter 14. Start OpenOCD in another terminal window as described in the section "Starting OpenOCD". In a new window, after setting the environment, if necessary, start gdb as follows:

```
$ cd ~/riscv/repo/14/rdcycle
$ $ idf.py gdb
...
Thread 1 hit Temporary breakpoint 1, app_main () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/main.c:7
7     app_main(void) {
(gdb)
```

Note: Square brackets as in "[info]" will be used in this chapter to indicate valid abbreviations. The square brackets are not typed and are not part of the command. A frequently used command is seldom typed out in full.

Note: The sessions that are shown in this chapter have pathnames abbreviated to a path that begins with a tilda (~). The actual path shown will include a full pathname when you perform the same operations. For example, userid jackie on MacOS will display without the tilda as in:
/Users/jackie/riscv/repo/14/rdcycle/main/main.c

Espressif automatically breakpoints at the start of `app_main()`, where the ESP32-C3 application begins. One helpful command is the "`list`" command, since it shows us source lines near the current breakpoint without having to refer to the program listing:

```
Thread 1 hit Temporary breakpoint 1, app_main () at ~/riscv/repo/14/rdcycle/main/main.c:7
7     app_main(void) {
(gdb) list
2     #include <stdint.h>
3
4     extern uint32_t measure(int mul);
5
6     void
7     app_main(void) {
8         uint32_t cycles;
9
10        for ( int x=0; x<10; ++x ) {
```

```
11             cycles = measure(1);
(gdb) <CR>
12             printf("multiply cycles = %u\n",cycles);
13             cycles = measure(0);
14             printf("shift  cycles = %u\n",cycles);
15         }
16         fflush(stdout);
17     }
(gdb) <CR>
Line number 18 out of range; ~/riscv/repo/14/rdcycle/main/main.c has 17 lines.
(gdb)
```

Notice how the "list" command was repeated twice by just pressing return (shown as "<CR>") in the sample session. After listing several lines of code, you might lose track of where the program was stopped. Use the "f[rame]" command to display and refresh your memory:

```
(gdb) frame
#0  app_main () at ~/riscv/repo/14/rdcycle/main/main.c:7
7   app_main(void) {
(gdb)
```

This reminds us that we are stopped at line 7 of main.c.

Since there is a stack involved, we can also do a "ba[cktrace]" (or "bt"). This is one command that is also abbreviated as "bt" since it is often used and is more mnemonic than "ba":

```
#0  app_main () at ~/riscv/repo/14/rdcycle/main/main.c:7
#1  0x4201080c in main_task (args=<optimized out>) at ~/esp32c3/esp-idf/
components/freertos/port/port_common.c:129
#2  0x40385b32 in vPortSetInterruptMask () at ~/esp32c3/esp-idf/components/
freertos/port/riscv/port.c:306
Backtrace stopped: frame did not save the PC
(gdb)
```

The "backtrace" command is extremely helpful in identifying where a problem occurred, or exactly where we have stopped. It indicates which function called which, especially when a fault or abort occurs. In this example, we see that there are two stack frames shown prior to calling app_main() at frame number 0. Keep the "bt" in your back pocket.

In C/C++ code, a person often wants to step through the code one statement at a time. This is done with the "n[ext]" command. Let's perform one "next" now:

```
(gdb) n
10         for ( int x=0; x<10; ++x ) {
(gdb)
```

This takes us to the first executable statement in `main.c:10` (the convention used by `gdb` is to report a file name followed by a colon and then a line number for ease of reference). Let's step one more time:

```
(gdb) n
11             cycles = measure(1);
(gdb)
```

At this point, the "for" statement has begun, and we are now ready to call the assembler routine `measure()`. What if you wanted to know what variable `x` was at this point? You "p[rint]" it of course:

```
(gdb) p x
$2 = 0
(gdb)
```

This is what we expected the first time into the loop. Sometimes you may want to see all local variables at once for convenience. This can be done with the "`i[nfo] lo[cal]s`" command:

```
(gdb) i lo
x = 0
cycles = <optimized out>
(gdb)
```

Unfortunately, the value for `cycles` has been "optimized out" by the compiler. This can be a nuisance when debugging. In this particular example, the value is not technically defined anyway, since it was not initialized. Anyway, that is another `gdb` command to keep in your back pocket.

Let's assume that for this particular time, we don't actually want to examine all the assembler steps involved in the `measure()` function. We can invoke the function and have it return its result by use of the "`n[ext]`" command again:

```
(gdb) n
Note: automatically using hardware breakpoints for read-only addresses.
12             printf("multiply cycles = %u\n",cycles);
(gdb)
```

The JTAG system reports a message about using hardware breakpoints, and then we're placed at the statement following the call to `measure()`. The value `cycles` should now be defined with the returned value. Fortunately, we can report it this time (sometimes even this is optimized out):

```
(gdb) i lo
x = 0
cycles = 2
(gdb)
```

From this, we can see that variable `x` is still zero, and that the variable `cycles` now have the value 2. Let's continue with the next statement:

```
(gdb) n
13             cycles = measure(0);
(gdb)
```

Notice in this example, that no printed output is shown. This is because we don't have a window open for the serial output. In this example, we don't need it since we've already seen the first value of `cycles` returned.

Let's assume that we now want to see the assembler function's operation in greater detail. To step *into* the function `measure()`, use the "`s[tep]`" command:

```
(gdb) s
measure () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/measure.S:15
15             li      a1,99             # Some number
(gdb)
```

This time, the execution has stopped at `measure.S:15` at the beginning of the assembled function `measure()`. Just for fun, issue the "`ba[cktrace]`" (or "`bt`") to report on the stack:

```
(gdb) bt
#0  measure () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/measure.S:15
#1  0x42004c5c in app_main () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/main.c:13
#2  0x4201080c in main_task (args=<optimized out>) at /Users/ve3wwg/esp32c3/esp-idf/components/freertos/port/port_common.c:129
#3  0x40385b32 in vPortSetInterruptMask () at /Users/ve3wwg/esp32c3/esp-idf/components/freertos/port/riscv/port.c:306
Backtrace stopped: frame did not save the PC
(gdb)
```

From this, you can see that `app_main()` at `main.c:13` has called `measure()` at `measure.S:15`. Notice the frame numbers have changed, with `app_main()` at frame 1, and `measure()` is now frame 0.

We know that our argument arrives in register a0. We can display that as follows:

```
(gdb) p $a0
$a4 = 0
(gdb)
```

From this, we can report any register by name by prepending a dollar (\$) to the register name. We can also report *all* registers if you need to with the "i[nfo] r[egisters]" command:

```
(gdb) i r
ra          0x42004c5c      0x42004c5c <app_main+36>
sp          0x3fc8edf0      0x3fc8edf0
gp          0x3fc8a600      0x3fc8a600
tp          0x3fc8891c      0x3fc8891c
t0          0x3de   990
t1          0x3fc8ea4c      1070131788
t2          0x0    0
fp          0x0    0x0
s1          0x0    0
a0          0x0    0
a1          0x3fc8ea28      1070131752
a2          0x0    0
a3          0x1    1
a4          0x3fc8c000      1070120960
a5          0x0    0
a6          0x42001dc8      1107303880
a7          0x0    0
s2          0x0    0
s3          0x0    0
s4          0x0    0
s5          0x0    0
s6          0x0    0
s7          0x0    0
s8          0x0    0
s9          0x0    0
s10         0x0    0
s11         0x0    0
t3          0x6b197044      1796829252
t4          0x0    0
t5          0x0    0
t6          0x0    0
pc          0x42004c84      0x42004c84 <measure>
(gdb)
```

Let's now step further into the `measure()` function:

```
(gdb) fr
#0  measure () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/measure.S:15
15      li      a1,99          # Some number
(gdb) s
16      beqz   a0,1f          # If mul is false, jump to 1
(gdb) p $a1
$5 = 99
(gdb)
```

Here I repeated "f[rame]" to remind me where we were, followed by an "s[tep]" to execute that statement (instruction in this case). Gdb reports that the next instruction is line 16. Then I printed the value of register `a1`, and it is reported as 99 as line 15 said it should.

We previously reported register (argument 1) as containing a zero, so it is no surprise that the branch is now taken as a result of the `beqz` instruction:

```
(gdb) s
28      1:     csrr   t1,mpccr
(gdb)
```

Stepping again:

```
(gdb)
29      sll    a2,a1,3        # a2 = a1 * 8
(gdb) <CR>
(gdb) p $t1
$6 = 324356966
(gdb) p /x $t1
$7 = 0x13554b66
(gdb)
```

Since the last thing we did was a "s[tep]", pressing enter (shown as "<CR>"), repeats the step once again. Printing register `t1`, shows us the value in decimal, which is inconvenient here. To print in hexadecimal, use the "/x" argument after the "p[rint]" command name.

```
(gdb)
29      sll    a2,a1,3        # a2 = a1 * 8
(gdb)
30      sll    a1,a1,1        # a1 *= 2
(gdb)
31      add   a0,a2,a1        # a0 = a1 * 10 => 200 cycles
(gdb) p $a2
$1 = 792
(gdb) p $a1
```

```

$2 = 198
(gdb) s
32          csrr    t3,mpccr
(gdb) p $a0
$3 = 990
(gdb)

```

In this sequence, we stepped through the execution of lines 29 to 31, where the multiply by ten was performed by two shifts and an add. Examining the result in `$a0` confirms that 99 was indeed multiplied by 10.

As we step through the rest of the `measure()` function, the computed return value may report a wildly large number. This is because the `mpccr` values returned in `measure.S:28` and `measure.S:32` include a number of cycles that are going on in the background as a result of being debugged in single stepping mode. With that in mind, let's step until the function returns:

```

(gdb) fr
#0  xit () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/measure.S:34
34  xit:  sub    a0,t3,t1
(gdb) s
35          ret
(gdb)
app_main () at /Users/ve3wwg/riscv/repo/14/rdcycle/main/main.c:14
14          printf("shift  cycles = %u\n",cycles);
(gdb) p cycles
$15 = 716845243
(gdb)

```

We see here that the value of `cycles` is much larger than it should be. But this is understandable since we single-stepped through the code while the cycles kept ticking.

Let's assume that you're now satisfied that the program is executing correctly and that you want it to continue unhindered. We can "`c[ontinue]`" the program:

```

(gdb) c
Continuing.

```

At this point, `gdb` will seem to hang and not give you a prompt. Let's find out why. Press `Control-C` to interrupt `gdb`:

```

^C
Thread 2 received signal SIGINT, Interrupt.
[Switching to Thread 1070133268]
0x42006342 in esp_vApplicationIdleHook () at /Users/ve3wwg/esp32c3/esp-idf/

```

```

components/esp_system/freertos_hooks.c:50
50         for (int n = 0; n < MAX_HOOKS; n++) {
(gdb) bt
#0  0x42006342 in esp_vApplicationIdleHook () at /Users/ve3wwg/esp32c3/esp-idf/
components/esp_system/freertos_hooks.c:50
#1  0x40384960 in prvIdleTask (pvParameters=<optimized out>) at /Users/ve3wwg/
esp32c3/esp-idf/components/freertos/tasks.c:3973
#2  0x40385b32 in vPortSetInterruptMask () at /Users/ve3wwg/esp32c3/esp-idf/
components/freertos/port/riscv/port.c:306
Backtrace stopped: frame did not save the PC
(gdb)

```

After interrupting gdb with Control-C, gdb reports to us that it is stuck in a FreeRTOS routine. This is what happens after `app_main()` returns, since control has now returned to the environment that called our `app_main()`.

While we're here, let's take note of the fact that gdb is FreeRTOS thread-aware. We can report on the threads that are running using `"i[nfo] "th[reads]"`:

```

(gdb) i th
  Id  Target Id                                Frame
* 2   Thread 1070133268 (Name: IDLE)         0x42006342 in esp_vApplicationIdleHook
()
    at /Users/ve3wwg/esp32c3/esp-idf/components/esp_system/freertos_hooks.c:50
  3   Thread 1070128416 (Name: esp_timer) 0x40385ba4 in vPortClearInterruptMask
(mask=1)
    at /Users/ve3wwg/esp32c3/esp-idf/components/freertos/port/riscv/port.c:329
(gdb)

```

We see that we are now in the FreeRTOS IDLE thread, while another thread also exists as the `esp_timer` thread. If our `app_main()` program was still running, it would have been shown as running as the main thread. You can change threads if you want to examine where it is executing by using `"t[hread] <n>"`, where `"<n>"` is the thread number shown.

15.5.3. Quitting gdb

When you're done with gdb, simply use the `"q[uit]"` command. If you need to restart gdb, nothing further should need to be done on the OpenOCD end, unless it got tripped up with a hardware error (like having the USB cable removed). Simply use:

```
$ idf.py gdb
```

as before.

15.6. JTAG With a Serial Window

Now that you have JTAG debugging conquered, let's try doing so again, but this time with a serial window available, so that you can view the printed output. This procedure must be followed in a prescribed sequence because starting the ESP32-C3 monitor causes a hardware reset. If OpenOCD were running at the time, it would error out creating complications.

Configuration

This procedure assumes that the "Channel for console output" has been set to "USB Serial/JTAG Controller". The default for ESP32-C3 projects is to use "UART0". To check this, perform the following in your build directory (and environment).

```
$ idf.py menuconfig
```

Then select the option "Component config →" shown in Figure 15.3.

```
(Top)
Espressif IoT Development Framework Configuration
SDK tool configuration --->
Build type --->
Application manager --->
Bootloader config --->
Security features --->
Boot ROM Behavior --->
Serial flasher config --->
Partition Table --->
Compiler options --->
Component config --->
Compatibility options --->

[Space/Enter] Toggle/enter [ESC] Leave menu [S]
[O] Load [?] Symbol info [/]
[F] Toggle show-help mode [C] Toggle show-name mode [A]
[Q] Quit (prompts for save) [D] Save minimal config (advan
```

Figure 15.3: `idf.py menuconfig`.

From Component config, select "ESP System Settings →" shown in Figure 15.4.

```
(Top) > Component config
Espressif IoT Development Framework Configuration
ESP HTTPS server --->
Hardware Settings --->
IPC (Inter-Processor Call) --->
LCD and Touch Panel --->
ESP NETIF Adapter --->
PHY --->
Power Management --->
ESP System Settings --->
High resolution timer (esp_timer) --->
Wi-Fi --->
Core dump --->
FAT Filesystem support --->
Modbus configuration --->
vvvvvvvvvvvvvvv
[Space/Enter] Toggle/enter [ESC] Leave menu [S]
[O] Load [?] Symbol info [/]
[F] Toggle show-help mode [C] Toggle show-name mode [A]
[Q] Quit (prompts for save) [D] Save minimal config (advan
```

Figure 15.4: `idf.py menuconfig: Component config`.

Then look for the line "Channel for console output" in Figure 15.5.

```
(Top) > Component config > ESP System Settings
Espressif IoT Development Framework Configuration
[ ] Generate and use eh_frame for backtracing
Memory protection --->
(32) System event queue size
(2304) Event loop task stack size
(3584) Main task stack size
Main task core affinity (CPU0) --->
(2048) Minimal allowed size for shared stack
Channel for console output (Default: UART0) --->
Channel for console secondary output (No secondary con
[*] Interrupt watchdog
(300) Interrupt watchdog timeout (ms)
[*] Initialize Task Watchdog Timer on startup
[ ] Invoke panic handler on Task Watchdog timeout
vvvvvvvvvvvvvvvv
[Space/Enter] Toggle/enter [ESC] Leave menu [S]
[O] Load [?] Symbol info [/]
[F] Toggle show-help mode [C] Toggle show-name mode [A]
[Q] Quit (prompts for save) [D] Save minimal config (advan
```

Figure 15.5: menuconfig: ESP System Settings.

If it already shows "USB Serial/JTAG Controller" then you are set. Otherwise, enter into that menu and change the selection so that there is an "X" in the line USB Serial/JTAG Controller" as shown in Figure 15.6.

```
config > ESP System Settings > Channel for console output
Espressif IoT Development Framework Configuration
( ) Default: UART0
(X) USB Serial/JTAG Controller
( ) Custom UART
( ) None

[Space/Enter] Toggle/enter [ESC] Leave menu [S]
[O] Load [?] Symbol info [/]
[F] Toggle show-help mode [C] Toggle show-name mode [A]
[Q] Quit (prompts for save) [D] Save minimal config (advan
```

Figure 15.6: menuconfig: Channel for console output.

Make certain you save your changes before quitting the menuconfig. After making changes you will need to build and flash your project again.

Procedure

Using this procedure, the following general steps must be performed in order:

1. Configure your project to use the USB Serial console with menuconfig, build and flash your ESP32-C3 device (as shown in the previous section).
2. Plug in your ESP32-C3 device USB cable to the PC (or unplug and replug).

3. In a new terminal session, establish your ESP environment (if necessary) and start `idf.py` monitor.
4. In its own window (and environment), launch OpenOCD
5. In a third window (and environment) launch `idf.py` gdb.

When OpenOCD starts, it will again cause a reset of the device, so the monitor window may show some output. Once the gdb window starts, you should again see the output:

```
$ idf.py gdb
Executing action: gdb
GNU gdb (crosstool-NG esp-2021r2-patch3) 9.2.90.20200913-git
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-host_apple-darwin12
--target=riscv32-esp-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
/Users/ve3wwg/.gdbinit:1: Error in sourced command file:
No symbol table is loaded. Use the "file" command.
0x40000000 in ?? ()
JTAG tap: esp32c3.cpu tap/device found: 0x00005c25 (mfg: 0x612 (Espressif
Systems), part: 0x0005, ver: 0x0)
Hardware assisted breakpoint 1 at 0x42004c38: file /Users/ve3wwg/riscv/repo/14/
rdcycle/main/main.c, line 7.
[New Thread 1070133268]
[New Thread 1070128416]
[Switching to Thread 1070132924]

Thread 1 hit Temporary breakpoint 1, app_main () at /Users/ve3wwg/riscv/repo/14/
rdcycle/main/main.c:7
7   app_main(void) {
(gdb)
```

Now if you were to step or run through the program, the console output should appear in your monitor window.

Problems

Problems can trip up the OpenOCD software, so it is sometimes necessary to recover from them. If OpenOCD doesn't start correctly try the following:

1. In the monitor window, exit the monitor using Control-].
2. In the gdb window, quit the debugger (if running).
3. In the OpenOCD window, Control-C to terminate it.
4. Unplug the USB cable.

For Linux/MacOS systems, check and kill any rogue OpenOCD processes still running:

```
$ ps -ef | grep openocd
501 76202 1690  0 2:48pm ttys005  0:02.16 openocd -f board/esp32c3-builtin.cfg
$ kill -9 76202
```

Then repeat the procedure. Sometimes the ESP32-C3 device can be messed up by the code flashed into it. If you suspect that, then include a `vTaskDelay()` call at the start of `app_main()` to give you time to get the monitor started and the OpenOCD process. In that manner, OpenOCD can reset the CPU before the monitor run can mess things up. Then work through gdb to locate the source of your inflicted problem.

15.6. Miscellaneous

The gdb debugger is too large to fully cover its power in one chapter. However, a few more things are worth mentioning:

1. When running gdb from Fedora Linux (without using JTAG), you need to start the program running with the `r[un]` command. But before you do that, most users set a breakpoint at `main()` with the use of `b[reakpoint] main` first. Then after you start the program running, it will pause upon entry to the `main()` function.
2. Breakpoints are also useful when using JTAG. They permit you to skip the execution of a large body of code, until you get to the point where you are interested in scrutinizing. But there may be restrictions based upon whether the code is in ROM, flash, or RAM.
3. To delete a breakpoint, use the `d[ele]te <n>` where `<n>` is the breakpoint number.
4. You can list breakpoints with `i[n]fo b[reakpoints]`.
5. The `p[rint]` command can also display the contents of structures and classes. This is far better than instrumenting a program with oodles of `printf()` calls.

Don't be afraid to use the gdb help system and peruse the fine manual.

15.7. Summary

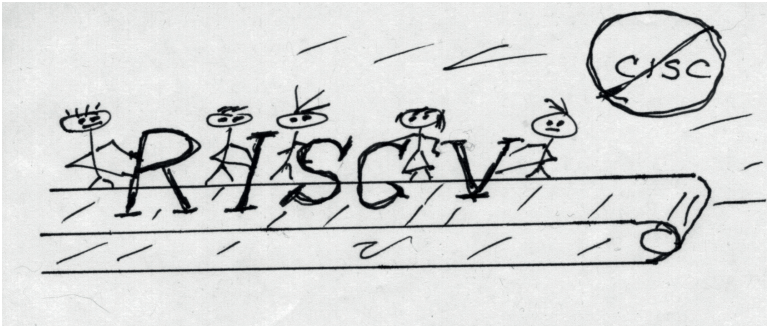
There is considerably more that could be explained about gdb's debugging capabilities. But what you have seen in this chapter should get you started using gdb to locate bugs. JTAG is wonderful in that it allows you to debug the actual hardware that you are using. Only do be aware that there are a number of restrictions that the user should be aware of. Espressif

has documented these on their website.[5] The use of JTAG and gdb can save the user time and frustration, by tracing assembly code one instruction at a time.

Bibliography

- [1] <https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-guides/jtag-debugging/index.html>
- [2] https://www.gnu.org/software/bash/manual/html_node/Command-Line-Editing.html
- [3] http://web.mit.edu/gnu/doc/html/features_7.html
- [4] https://developer.apple.com/library/archive/documentation/DeveloperTools/gdb/gdb/gdb_4.html
- [5] <https://www.sourceware.org/gdb/documentation/>
- [6] <https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-guides/usb-serial-jtag-console.html>

Chapter 16 • Inline Assembly



The inline RISC-V assembly line

With the utility and productivity of C/C++, there are likely to be times when you just want to invoke a few assembly language instructions directly from your high-level code. The GCC compiler makes this possible through the "asm" (or "__asm__") extension. This chapter gets you started with the basics of this advanced facility.

16.1. Keyword Asm

The GNU GCC compiler permits the "asm" keyword as a language extension to allow the programmer to supply assembler language elements. When compiling with options -ansi or some of the -std options, you must use the keyword "__asm__" instead, since the keyword is an extension to the standard.

16.2. Basic asm Form

The most basic form of this extension uses the following format:

```
asm [volatile] ("AssemblerInstruction(s)");
```

This most basic form can be provided both inside and outside of functions (the extended form can only be used inside functions).

Let's see this simple mechanism at work. I'll reuse the rdcycle project from chapter 14 but use copies of the files for this chapter so that we can mess with it. Listing 16.1 is our first demonstration of the code change to main.c. Take note of line 10, where a "nop" (no operation) instruction was added inline with the C code.

```
19 #include <stdio.h>
2 #include <stdint.h>
3
4 extern uint32_t measure(int mul);
5
6 void
7 app_main(void) {
8     uint32_t cycles;
```

```

 9
10     asm volatile ( "nop" );
11
12     for ( int x=0; x<10; ++x ) {
13         cycles = measure(1);
14         printf("multiply cycles = %u\n",cycles);
15         cycles = measure(0);
16         printf("shift   cycles = %u\n",cycles);
17     }
18     fflush(stdout);
19 }

```

Listing 16.1: Program `~/riscv/repo/16/rdcycle/main/main.c` with a basic asm statement.

If you need convincing that everything still works, build, flash and monitor the program as follows:

```

$ cd ~/riscv/repo/16/rdcycle
$ idf.py build
$ idf.py -p <<<yourport>>> flash monitor

```

Now let's examine the assembler listing for the main.c program in Listing 16.2, which has been edited to reduce the page count. You can view the full listing by generating it as follows:

```

$ ~/riscv/repo/listesp main/main.c

```

```

...snip...
 6             void
 7             app_main(void) {
 8                 uint32_t cycles;
 9
10 0008 0100             asm volatile ( "nop" );
11
12
13
14
15
16
17
18
19
20
21
22             nop
23             # 0 "" 2
24             #NO_APP
25 000a 232604FE        sw     zero,-20(s0)
26 000e 81A8            j      .L2
...snip...

```

Listing 16.2: Edited assembler listing for main.c with nop added.

The important thing to note is how the compiler inserts the nop instruction at line 22 of the listing. On the left side of the listing, you see at offset 0008 the assembled nop opcode

0100, in hexadecimal. Of course, this doesn't do much for the program but serves as an illustration.

16.2.1. Keyword `volatile`

The `volatile` keyword often comes up in the context of compiler code optimization. Using it with the `asm` keyword is no exception. The `volatile` keyword directs the compiler not to optimize out your inline assembler code. Otherwise, it might assume that the code is not needed and suppress it. This can be troublesome at higher optimization levels.

16.2.2. Multiple Instructions

The "nop" example was extremely simple. However, it is possible to include multiple assembler statements as follows:

```
asm volatile (  
    "nop\n"  
    "\tnop\n"  
);
```

This results in assembling two nop instructions:

```
10 0008 0100      asm volatile (  
11 000a 0100          "nop\n"  
12                  "\tnop\n"  
13                  );  
22                  nop  
23                  nop
```

Notice a few things here. We used the newline (`\n`) character to separate the two assembler instructions. The tab character (`\t`) was put in front of the second opcode to indicate that no label was present. A space works equally well. You might not need to do this for RISC-V, but it makes the assembler listing easier to read. As you can see, lines 10 and 11 report the two assembled nop instructions as we expected.

It is also possible to provide multiple opcodes on one line and in one string. Remember that C/C++ will concatenate string constants if they are listed one after the other. So, in the last example, the compiler would have compiled:

```
asm volatile ( "nop\n" "\tnop\n" );
```

the same as:

```
asm volatile ( "nop\n\tnop\n" );
```

I believe that making code friendly for the reader is important. So listing opcodes on their own lines is polite.

16.2.3. Behind the Scenes

It should be pointed out that the C/C++ compiler does *not* interpret what is provided in the asm strings. What is supplied is merely copied with optional substitution to the temporary file before it is assembled. Thus, errors in the opcodes will not be detected until the temporary file is assembled.

16.3. Extended Asm

In the extended asm formats, it is possible for you to read, write or read/write C/C++ language variables from the assembler opcodes. The two extended forms are listed below:

```
asm [volatile] ( "AssemblerTemplate"
    : "OutputOperands"
    [ : "InputOperands" [ : "Clobbers" ] ])

asm [volatile] goto ( "AssemblerTemplate"
    :
    : "InputOperands"
    : "Clobbers"
    : "GotoLabels")
```

The "goto" keyword informs the compiler that there may be a jump to one of the labels listed. Extended asm statements must be used inside a C/C++ function.

16.3.1. Assembler Template

Just like the basic form, the string represents an assembler language template. Substitutions are made by the compiler before the string is submitted to the assembler language file. Special tokens are introduced with the percent (%) character. Special tokens are shown in Table 16.1, ready to be used when necessary.

Token	Description
%%	Represents a single percent (%) character.
%{	Represents a left curly bracket ({}).†
%	Represents a vertical bar ().†
%}	Represents a right curly bracket (}).†
%=	Output a unique asm instance number

Table 16.1: Special Assembler Tokens.

† While GCC documents these, they don't seem to be accepted by the RISC-V compiler for the ESP32-C3.

The following example illustrates a listing extract where some of these special symbols are used. Extracted listing lines 10 through 14 show the source lines that went into the asm block, while listing lines 22 to 24 show what was actually written into the assembly file. Notice that '%=' caused "10" to be substituted and confirmed in listing line 23. It is also

used as an immediate constant on line 24. The single '%' was written in place of the '%%', confirmed in listing line 23.

```

10 0008 0100          asm volatile (
11 000a 93025002      "nop\n"
12 000e 2943         "%=: li t0, '%'\n"
13                   " li t1, %= \n"
14                   : :
22                   nop
23                   10: li t0, '%'
24                   li t1, 10

```

16.3.2. Output Operands

Following the `AssemblerTemplate` is the optional output operands field. This field has the following general format:

```
[ [asmSymbolicName] "constraint" ( cexpression )
```

The `[asmSymbolicName]` parameter is optional. When used, it gives the parameter a name for references within the `AssemblerTemplate` like `"%[name]"`. When omitted, use a zero-based parameter reference of the form `'%0'`, `'%1'` etc. for parameters listed as output operands. When the parameter is provided, you must put the square brackets around the name.

16.3.2.1. Constraint

The constraint string normally begins with a '=' or '+' character for *output* parameters (there are other possibilities, but they have limited use). Table 16.2 lists their functions. Characters that follow this leading character, identify possibilities where the value resides (there can be more than one). Table 16.3 lists the constraint characters discussed in this chapter.

Character	Meaning
'='	This operand is written to by this asm block. The previous value held is discarded and replaced by new data.
'+'	This operand is both read and written by this asm block.

Table 16.2: Starting constraint characters (Output).

Character	Meaning
'm'	A memory operand is permitted, with any kind of address that the machine generally supports.
'r'	A register operand is permitted provided that it is a general register.

Table 16.3: Simple constraint characters.

The GCC notes some special exceptions like the following. If the constraint used on an Output Operand starts with a '+' (rather than '='), then that counts as two parameters (input + output). So, when specifying '%5', for example, make sure to take this into account. For code reading sanity and correctness, it is far safer to use the '%[name]' form instead.

Output Operand Example 1

Reading about cryptic rules and syntax can be confusing. So, let's illustrate some concrete examples. Listing 16.3 provides a partial assembly language listing for a program that initializes variables `cycles` and `ninety5` from inlined assembly language code. Yes, this is a silly way to do it, but we justify it in the name of learning.

Note: In these first two Output Operand examples, I have left out the Clobbers clause so that it can be explained later. Technically, these examples should list register `t1` as being clobbered (even though I was able to get away without it this time).

```

6           void
7           app_main(void) {
8               uint32_t cycles;
9               uint32_t ninety5;
10
11 0008 1303F005    asm volatile (
12 000c 232264FE        " li t1,95\n"
13 0010 232404FE        " sw t1,%[ninety5]\n"
14                " sw x0,%[cycles]\n"
15                : [ninety5] "=m" (ninety5), [cycles] "=m" (cycles)
25                li t1,95
26                sw t1,-28(s0)
27                sw x0,-24(s0)

```

Listing 16.3: The program `app_main()` initializing two variables from inline assembly.

The first output clause specified from line 15 is:

```
[ninety5] "=m" (ninety5),
```

1. This tells the compiler that we will refer to the variable in the assembler code as "%[ninety5]".
2. The constraint character '=' indicates that the output value will be overwritten and that any previous value of `ninety5` is discarded.
3. The 'm' in the constraint indicates that the output variable is in memory.
4. Finally, the C expression '(ninety5)' indicates to the compiler where the value is going.

The second output clause from line 15 is:

```
[cycles] "=m" (cycles)
```

1. This indicates that "%[cycles]" is how the variable for cycles is referenced in the assembler code.
2. The constraint "=m" indicates that the output value will be overwritten and is in memory.
3. The C expression "cycles" is used to provide the location for the operand output.

The output of the assembly from Listing 16.3 is:

```

25             li t1,95
26             sw t1,-28(s0)
27             sw x0,-24(s0)

```

From this, we see how the compiler has inserted the target address for ninety5 as -28(s0), and cycles as -24(s0). The compiler knows the offsets of these variables on the stack, relative to the save register s0. This frees the programmer from having to figure it out.

Note: It might seem that the `asmSymbolicName` is redundant ([cycles] vs (cycles)). But what is specified in brackets (cexpression) can be a C/C++ expression, which need not be a simple variable name.

Output Operand Example 2

In this example, we perform the same work as the first example, except that we use the positional parameters "%0" and "%1" instead.

```

27  #include <stdio.h>
2
3
4      extern uint32_t measure(int mul);
5
6      void
7      app_main(void) {
8          uint32_t cycles;
9          uint32_t ninety5;
10
11 0008 1303F005      asm volatile (
12 000c 232264FE          " li t1,95\n"
13 0010 232404FE          " sw t1,%0\n"
14                          " sw x0,%1\n"
15                          : "=m" (ninety5), "=m" (cycles)
25                          li t1,95
26                          sw t1,-28(s0)
27                          sw x0,-24(s0)

```

In this example, the `[asmSymbolicName]` has been omitted from both output parameter specifications. Because of this, the inline code uses "%0" to refer to the first parameter ninety5, and "%1" to refer to cycles. An examination of listing lines 25 to 27 reveals that

the same code was generated. When there is a large number of output parameters, the use of the [asmSymbolicName] is recommended for code clarity.

16.3.3. Input Operands

The input operands take values from the C/C++ language side and make them available to the assembly code. Like the output parameters, multiple parameters are separated by commas. Reviewing the statement again, we see that InputOperands follows the OutputOperands separated by a colon (:) character. If there are no output operands, then the colon (:) must be specified.

```
asm [volatile] ( "AssemblerTemplate"
    : "OutputOperands"
    [ : "InputOperands" [ : "Clobbers" ] ] )
```

The general format for each input operand is as follows:

```
[ [asmSymbolicName] ] "constraint" (cexpression)
```

The optional [asmSymbolicName], the constraint and cexpression are used in a manner similar to the output parameters, except that this time the data is coming *from* the C program. A full C program example is illustrated in Listing 16.4.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 void
5 app_main(void) {
6     uint32_t a=33, b=75, m;
7
8     asm volatile (
9         " lw a0,%1\n"           // a0 = a
10        " lw a1,%2\n"           // a1 = b
11        " mul t0,a0,a1\n"       // t0 = a0 * a1
12        " sw t0,%0\n"           // m = t0
13        : "=m" (m)              // Outputs
14        : "m" (a), "m" (b)      // Inputs
15        : "a0", "a1", "t0"     // Clobbers
16    );
17
18    printf("%u * %u => %u\n",a,b,m);
19    fflush(stdout);
20 }
```

Listing 16.4: Program `~/riscv/repo/16/inmul/main/main.c`.

First build, flash and monitor it to convince yourself that it works. Here the variable `a` is multiplied by the value in `b` and the product is stored in the variable `m`. The `printf()` call in

line 18 will report the inputs and the produced product m.

```
$ cd ~/riscv/repo/16/inmul
$ idf.py build
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
33 * 75 => 2475
```

The reported product is good. Now let's break down what was coded in Listing 16.1.

1. Line 9 loads the value from variable *a*, into register *a0*. Notice that the input value for *a* is identified as "%1" because the expression (*a*) is the second *parameter* within the entire *asm* block.
2. In the same manner, line 10 loads the value from variable *b* into register *a1*. The value for *b* is coded as "%2" since it is the third *parameter* in the *asm* block.
3. Line 11 performs the multiplication, placing the product into temporary register *t0*.
4. Line 12 stores a word value from register *t0* into the expression "%0", which is the variable *m* (line 6).
5. Notice that the input parameter list uses a constraint of "m", whereas the output constraint used "=m". Both reference a value in memory, but the output needs the '=' to indicate how the value will be stored/updated. Inputs, on the other hand, can simply be fetched from memory.
6. Line 15 identifies the registers that were clobbered by our assembly code.

16.3.3.1. Clobbers

For some platforms there can be side effects from executing certain opcodes that change register values. Or we may simply assign registers to compute intermediate results. These must be identified in the clobbers clause so that the compiler is informed. In Listing 16.4, the registers *a0*, *a1* and *t0* were identified in the clobbers clause. This informs the compiler that these registers were used ("clobbered").

The compiler must choose registers to use for input and output operands. Registers listed in the clobbered list are not used by the compiler. Hence clobbered registers become available for any use in your assembler code. Additionally, the stack pointer register must not appear in the clobber list and must not be altered. The stack pointer must have the same value upon exit as it had upon entry.

Note: Incorrect code can result when modified registers are not identified in the clobbers clause. For example, imagine if the compiler placed an address in register *t0*, to be used later on. But in the *asm* block that follows, you modified register *t0*. If register *t0* is not referenced in an input or output clause, then the compiler would be completely oblivious to the fact that the address in *t0* was lost.

There are two special clobber arguments listed in Table 16.4. The "cc" argument has no value to RISC-V and can be ignored because it has no flags register. The "memory" argu-

ment may, however, be necessary if your code is reading/modifying memory outside of the parameters provided for input/output.

Argument	Meaning
cc	Indicates that the flags register was modified. On platforms like RISC-V that don't support a flags register, this argument is simply ignored.
memory	Indicates that the code reads/modifies memory locations other than those listed in the input/output operands. This may cause the compiler to flush certain variables held in registers.

Table 16.4: Special Clobber Arguments.

16.4. Bit Multiply

Listing 16.5 illustrates the next example demonstrating how the "cexpression" part can participate. In this program, we use the mulh and mul instruction pair to compute and return the 64-bit unsigned product.

```

1  #include <stdio.h>
2  #include <stdint.h>
3
4  void
5  app_main(void) {
6      uint32_t a=9300000, b=7500000;
7      union {
8          uint64_t m64;
9          uint32_t m32[2];
10     } u;
11
12     asm volatile (
13         " lw a0,%[a]\n"           // a0 = a
14         " lw a1,%[b]\n"           // a1 = b
15         " mulh t1,a0,a1\n"        // t1 = high a0 * a1
16         " mul t0,a0,a1\n"        // t0 = low a0 * a1
17         " sw t0,%[low]\n"        // m = t0 (low word)
18         " sw t1,%[hi]\n"        // m = t1 (high word)
19         : [low] "=m" (u.m32[0]), // Output: low
20           [hi] "=m" (u.m32[1])   // high
21         : [a] "m" (a), [b] "m" (b) // Inputs
22         : "a0", "a1", "t0", "t1" // Clobbers
23     );
24
25     printf("%u * %u => %llu\n",a,b,u.m64);
26     fflush(stdout);
27 }
```

Listing 16.5: ~/riscv/repo/16/inmul64/main/main.c.

The variables `a` and `b` have been assigned extra large constants to prove that our product results in more than 32 bits, is valid. Due to the increased number of input and output values, the `[asmSymbolicName]` form was used in this specification for inputs and outputs. For example, `%"[a]"` refers to the input variable `a` (line 21).

A C language union was used in lines 7 through 10 to permit access of the 64-bit value of `u.m64` as a pair of 32-bit values `u.m32[0]` and `u.m32[1]`. Notice in the output clause, how the C expressions in the refer to `%"[low]"` as the expression `u.m32[0]` and `%"[hi]"` as `m.32[1]`.

```
19          : [low] "=m" (u.m32[0]),          // Output: low
20          [hi] "=m" (u.m32[1])           // high
```

Build, flash and monitor the program to prove that it works:

```
$ cd ~/riscv/repo/16/inmul64
$ idf.py build
$ idf.py -p <<<yourport>>> flash monitor
I (257) cpu_start: Starting scheduler.
9300000 * 7500000 => 69750000000000
```

If you have `bc` installed, check the result:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
9300000 * 7500000
69750000000000
^D
$
```

The results agree. To see the result in hexadecimal, try the following:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
obase=16
9300000 * 7500000
3F6FEFF91C00
^D
$
```


Given that the result is longer than 8 hexadecimal digits, we know the result size is greater than 32 bits.

16.5. Example `asm goto`

The last example to be explored is the use of C language labels referenced by the assembler code. To review, the general form is:

```
asm [volatile] goto ( "AssemblerTemplate"
    :
    : "InputOperands"
    : "Clobbers"
    : "GotoLabels")
```

The first thing to take special note of is that there can be *no output operands* with this form (note how the output operands clause is blank). If you try to specify outputs, you get the compile error:

```
expected ':' before string constant "asm goto"
```

This limits the usefulness of this form, but this is what we have to work with. The GCC documentation indicates that the `asm goto` statement is always considered volatile. I suggest that you always include it in case the compiler defaults change.

Note: The GCC document suggests that you can provide input/output operands in the `asm goto` statement using the '+' constraint. However, I was not able to succeed in this using the compiler version: `riscv32-esp-elf-gcc (crosstool-NG esp-2021r2-patch3) 8.4.0`. That functionality may vary with platform type.

The addition of the `goto` keyword permits the specification of C/C++ language labels that can be referenced by the `asm` code. When referencing the labels within the assembler code, use one of the following formats:

- `"%l<n>"`, for example, `"%l2"` (note the lowercase 'l' after the percent character).
- `"%l[label]"`, for example, `"%l[exception]"` (note the lowercase 'l' after the percent character).

Another restriction is that the total number of input + output + `goto` operands is limited to 30.

The asm goto program example is provided in Listing 16.6.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 void
5 app_main(void) {
6     uint32_t a=315, b=75, r=0;
7
8     asm volatile goto (
9         " lw a1,%[b]\n"           // a1 = b
10        " beqz a1,%l[except]\n"   // Jump if b=zero
11        : /* no outputs allowed for goto */
12        : [b] "m" (b)             // Inputs
13        : "a1"                     // Clobbers
14        : except
15    );
16
17    asm volatile (
18        " lw a0,%[a]\n"           // a0 = a
19        " lw a1,%[b]\n"           // a1 = b
20        " div a0,a0,a1\n"         // a0 /= a1
21        " sw a0,%[r]\n"           // r = result
22        : [r] "=m" (r)            // Outputs
23        : [a] "m" (a), [b] "m" (b) // Inputs
24        : "a0", "a1"              // Clobbers
25    );
26
27    printf("%u / %u => %u\n",a,b,r);
28    fflush(stdout);
29    return;
30
31 except:
32    printf("Division by zero!\n");
33    fflush(stdout);
34 }
```

Listing 16.6: Program `~/riscv/repo/16/except/main/main.c`.

Because the "goto" form does not permit the specification of output parameters, there were two asm blocks defined in this program. The first block (lines 8 to 15) tests if variable b is zero, and if so, branches to the C label "except". Otherwise, it just returns. Of course, it is silly to use asm to test for zero in this way, but it is justified for demonstration purposes.

The second block is a normal asm block so it can actually perform the division and return the result (lines 17 to 25). Again, this simple asm example is justifiable in the name of education.

Build, flash and monitor the program to see if the "happy path" works as expected:

```
$ cd ~/riscv/repo/16/except
$ cd idf.py build
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
315 / 75 => 4
```

The divide was indeed performed and is correct. Now edit the program main.c so that the value of b is zero. Change line 6 to read:

```
uint32_t a=315, b=0, r=0;
```

Repeat the build, flash and monitor:

```
$ idf.py -p <<<yourport>>> flash monitor
...
I (258) cpu_start: Starting scheduler.
Division by zero!
```

In this particular run, the goto was in fact performed to report "Division by zero!". In this example, the only way execution can proceed to the label "except" is from our asm code.

One of the best applications for the asm goto form is perhaps the management of a state machine. Where entry into the asm block moves the execution from one label to another, depending upon the current state.

16.6. Register Constraints

In addition to the 'm' option for 'memory' in the constraints string, there is the option of using 'r' for register. Listing 16.7 demonstrates the usefulness of the 'r' constraint in performing the multiply instruction without any loading or storing of values.

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 // #pragma GCC optimize ("-O3")
5
6 void
7 app_main(void) {
8     uint32_t a=33, b=75, m;
9 }
```

```

10     asm volatile (
11         " mul %[m],[a],[b]\n"        // m = a * b
12         : [m] "=r" (m)              // Outputs
13         : [a] "r" (a), [b] "r" (b)  // Inputs
14     );    // No clobbers
15
16     printf("%u * %u => %u\n",a,b,m);
17     fflush(stdout);
18 }

```

Listing 16.7: Program ~/riscv/repo/16/inmulrr/main/main.c.

Notice that in line 12, the constraint is "=r" indicating that the product for variable m is expected to be in a register ('=' indicates that the result will overwrite the original value). The constraint "r" is used in line 13 for both variables a and b. This means that both of these variables are expected to have values already in memory.

With the pragma optimize commented out (line 4), let's examine the assembler listing to see what the compiler does with this.

```

$ cd ~/riscv/repo/16/inmulrr
$ ~/riscv/repo/listesp main/main.c

```

```

1          .file   "main.c"
2          .option nopic
3          .text
4          .section      .rodata
5          .align 2
6          .LC0:
7 0000 2575202A      .string "%u * %u => %u\n"
7          20257520
7          3D3E2025
7          750A00
8          .text
9          .align 1
10         .globl  app_main
12         app_main:
13 0000 0111          addi   sp,sp,-32
14 0002 06CE          sw    ra,28(sp)
15 0004 22CC          sw    s0,24(sp)
16 0006 0010          addi   s0,sp,32
17 0008 93071002      li    a5,33
18 000c 2326F4FE      sw    a5,-20(s0)
19 0010 9307B004      li    a5,75
20 0014 2324F4FE      sw    a5,-24(s0)
21 0018 8327C4FE      lw    a5,-20(s0)

```

```

22 001c 032784FE          lw      a4,-24(s0)
23                          #APP
24                          # 10 "main/main.c" 1
   1                          #include <stdio.h>
   2                          #include <stdint.h>
   3
   4                          // #pragma GCC optimize ("-O3")
   5
   6                          void
   7                          app_main(void) {
   8                              uint32_t a=33, b=75, m;
   9
10 0020 B387E702          asm volatile (
11                              " mul %[m],[a],[b]\n" // m = a * b
12                              : [m] "=r" (m) // Outputs
25                              mul a5,a5,a4
26
27                          # 0 "" 2
28                          #NO_APP
29 0024 2322F4FE          sw      a5,-28(s0)
30 0028 832644FE          lw      a3,-28(s0)
31 002c 032684FE          lw      a2,-24(s0)
32 0030 8325C4FE          lw      a1,-20(s0)
33 0034 B7070000          lui    a5,%hi(.LC0)
34 0038 13850700          addi   a0,a5,%lo(.LC0)
35 003c 97000000          call   printf
35      E7800000
36 0044 97000000          call   __getreent
36      E7800000
37 004c AA87              mv      a5,a0
38 004e 9C47              lw      a5,8(a5)
39 0050 3E85              mv      a0,a5
40 0052 97000000          call   fflush
40      E7800000
41 005a 0100              nop
42 005c F240              lw      ra,28(sp)
43 005e 6244              lw      s0,24(sp)
44 0060 0561              addi   sp,sp,32
45 0062 8280              jr      ra
47                          .ident  "GCC: (crosstool-NG esp-2021r2-patch3)
8.4.0"

DEFINED SYMBOLS
                                *ABS*:0000000000000000 main.c
/var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//ccYqAtij.s:12
.text:0000000000000000 app_main

```

```

/var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//ccYqAtij.s:6
.rodata:0000000000000000 .LC0

UNDEFINED SYMBOLS
printf
__getreent
fflush

```

Listing 16.8: Assembler language listing of ~/riscv/repo/16/inmulrr/main/main.c.

Let's now breakdown the steps used by the compiler:

1. There is some initial stack frame setup in `app_main()` in lines 13 to 16.
2. Line 17 loads the value 33 into `a5` and then stores that in line 18 (assigns `a=33`).
3. Line 19 loads the value 75 into `a5` and then stores that in line 20 (assigns `b=75`).
4. Then the value of `b` is loaded into `a5` in line 21 (keep in mind this is unoptimized code).
5. The value of `a` is then loaded into `a4` at line 22.
6. The `asm` block's "mul" instruction is assembled (see lines 10 and 25 of the listing), to multiply registers `a5` and `a4`, placing the result in register `a5`.
7. The product in `a5` is stored into variable `m` in line 29.

All of this is confirmed by looking at the call to `printf()` that follows:

1. Argument 0 (`printf` text) is established in `a0` at line 34.
2. Argument 1 is provided in line 32 by loading `a` into `a1`.
3. Argument 2 is provided in line 31 by loading `b` into `a2`.
4. Argument 3 is provided in line 30 by loading `m` into `a3`.

Build, flash and monitor the program to prove that it works:

```

$ cd ~/riscv/repo/16/inmulrr
$ cd idf.py build
$ idf.py -p <<<yourport>>> flash monitor
...
I (257) cpu_start: Starting scheduler.
33 * 75 => 2475

```

Indeed, it does! Now uncomment the `#pragma` in line 4 of `main.c` and repeat the build. Then produce an assembler listing (as shown in Listing 16.9) to see what the optimized code looks like:

```

$ cd ~/riscv/repo/16/inmulrr
$ cd idf.py build
$ ~/riscv/repo/listesp main/main.c

```

```

1          .file "main.c"
2          .option nopic
3          .text
4          .section      .rodata.str1.4,"aMS",@progbits,1
5          .align 2
6          .LC0:
7 0000 2575202A      .string "%u * %u => %u\n"
7          20257520
7          3D3E2025
7          750A00
8          .text
9          .align 1
10         .globl app_main
12         app_main:
13 0000 4111          addi   sp,sp,-16
14 0002 06C6          sw    ra,12(sp)
15 0004 93061002      li    a3,33
16 0008 9307B004      li    a5,75
17         #APP
18         # 10 "main/main.c" 1
1          #include <stdio.h>
2          #include <stdint.h>
3
4          #pragma GCC optimize ("-O3")
5
6          void
7          app_main(void) {
8              uint32_t a=33, b=75, m;
9
10 000c B386F602      asm volatile (
11                 " mul %[m],[a],[b]\n" // m = a * b
12                 : [m] "=r" (m)      // Outputs
13                 : a3,a3,a5
14                 :
15                 :
16                 :
17                 :
18                 :
19                 :
20                 :
21                 :
22                 :
23 0010 37050000      lui   a0,%hi(.LC0)
24 0014 1306B004      li    a2,75
25 0018 93051002      li    a1,33
26 001c 13050500      addi  a0,a0,%lo(.LC0)
27 0020 97000000      call printf
27         E7800000
28 0028 97000000      call __getreent
28         E7800000
29 0030 B240          lw    ra,12(sp)
30 0032 0845          lw    a0,8(a0)

```

```

31 0034 4101                addi    sp,sp,16
32 0036 17030000            tail   fflush
32      67000300
34                          .ident  "GCC: (crosstool-NG esp-2021r2-patch3)
8.4.0"

DEFINED SYMBOLS
                *ABS*:0000000000000000 main.c
/var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//ccjVYvnV.s:12
.text:0000000000000000 app_main
/var/folders/jp/_ktfnf412kvbdznr4m9769tr0000gn/T//ccjVYvnV.s:6      .rodata.
str1.4:0000000000000000 .LC0

UNDEFINED SYMBOLS
printf
__getreent
fflush

```

Listing 16.9: The optimized main.c using register constraints.

What does the code do now?

1. Line 15 loads the value for `a` into the register `a3`. Notice that no store to memory has been made yet to variable `a`.
2. Line 16 loads the value for `b` into the register `a5`. Likewise, the memory copy of variable `b` is left uninitialized.
3. Lines 10 and 19 show that the `"mul"` instruction multiplies registers `a3` and `a5`, replacing `a3` with the product.
4. The `printf()` format string address is loaded in line 23 in `a0` (first argument).
5. The value 33 for `a` is simply loaded into `a1` in line 25 (second argument).
6. The value 75 for `b` is simply loaded into `a2` in line 24 (third argument).
7. The product is already in register `a3`, so this gets passed to `printf()` as the fourth argument in the call to `print` in line 27.

From all of this, it should be clear that leaving the argument shuffle up to the C/C++ compiler works to your advantage. Use the `"r"` register constraint whenever it is practical to do so.

It is tempting to supply `"rm"` for an operand constraint. This does not work for RISC-V because if the operand is in memory, you will have written a load or store instruction like `"sw"`. You cannot store words to a register, causing the compiler to complain (its operand must be a memory reference). Likewise, you cannot supply a memory reference to an opcode like `"mul"`, for example. That opcode requires a register operand.

16.7. Summary

If you found the use of the `asm` statement tedious in this chapter, then don't be discouraged. It is somewhat cryptic but is useful when minor assembler help is required. It is otherwise not the best form for larger assembly language code segments. However, it is a good tool to keep in your bag of tricks. You'll also run across it in other people's code. All the more reason to master it.

There are some additional `asm` tricks for advanced use that didn't make it into this chapter. You can view those other options in detail at the `gnu` website.^[1] I hope, however, that you found this chapter a painless guide to getting started. Having mastered the basics, building upon that is easier for those who need it.

Please accept my thanks for allowing me to be your guide in this grand adventure. Re-ignite that passion for controlling your RISC-V machine at its most basic level: assembly language!

Bibliography

[1] <https://gcc.gnu.org/onlinedocs/gcc/Basic-Asm.html#Basic-Asm>

Index

32-bit register	83		
A		H	
ABI register	53	hello_world	25
Ada	148	HomeBrew	43
AMD instructions sets	14	I	
Application Binary Interface	53	IDLE thread	240
apt	37	Install Prerequisites	21
Arch	22	Instruction Set Architecture	13
arithmetic shift	118	J	
B		JTAG support	227
boot.bat script	48	L	
C		License Agreement	30
C/C++	81	Linux USB access	29
C compiler	62	LP64 Solaris model	62
CentOS	21	M	
cexpression	253	Machine Performance Counter Value	225
CHG340 chip	19	MacOS	22
CMU extensions	216	Manual Installation	19
COM port	33	matrix organized	164
CSR	186	MIPS	15
cycle snapshot	221	MISA register	59
D		MPCCR counter	222
Debian	21	MXL field	210
E		N	
embedded platforms	178	NaN-boxing	192
Environment Variables	25	NULL pointer	164
ESP-IDF	30	O	
ESP-IDF PowerShell	32	OpenOCD	226
F		OpenOCD session	230
Fedora Linux	18, 41	operating privilege mode	209
FLEN bits	185	overflow occurred	147
floating-point format	192	P	
FreeRTOS	212	Pre-Processor	197
G		pseudo-instructions	212
GCC	197	PuTTY	50
gdb debugger	87		

Q

QEMU emulator	13
QEMU Install	47

S

SRAM	81
SSH Authentication	45
stack layout	109
Stanford University	15
subi	130

T

tilda	233
-------	-----

U

UART0	241
Ubuntu	21
Unicode UTF-8	29
USB Serial/JTAG Controller	242
USB-UART bridge	19

V

variable bcount	119
-----------------	-----

W

WARL	210
------	-----

X

XLEN	51
XLEN-bit divisor	153
Xtensa	16

RISC-V Assembly Language Programming

Using the ESP32-C3 and QEMU

With the availability of free and open source C/C++ compilers today, you might wonder why someone would be interested in assembler language. What is so compelling about the RISC-V Instruction Set Architecture (ISA)? How does RISC-V differ from existing architectures? And most importantly, how do we gain experience with the RISC-V without a major investment? Is there affordable hardware available?

The availability of the Espressif ESP32-C3 chip provides a way to get hands-on experience with RISC-V. The open sourced QEMU emulator adds a 64-bit experience in RISC-V under Linux. These are just two ways for the student and enthusiast alike to explore RISC-V in this book.

The projects in this book are boiled down to the barest essentials to keep the assembly language concepts clear and simple. In this manner you will have "aha!" moments rather than puzzling about something difficult. The focus in this book is about learning how to write RISC-V assembly language code without getting bogged down. As you work your way through this tutorial, you'll build up small demonstration programs to be run and tested. Often the result is some simple printed messages to prove a concept. Once you've mastered these basic concepts, you will be well equipped to apply assembly language in larger projects.



Warren Gay is a datablocks.net senior software developer, writing Linux internet servers in C++. He got involved with electronics at an early age, and since then he has built microcomputers and has worked with MC68HC705, AVR, STM32, ESP32 and ARM computers, just to name a few.



Elektor International Media BV
www.elektor.com

