MAKER INNOVATIONS SERIES

RISC-V Assembly Language Programming

Unlock the Power of the RISC-V Instruction Set

—

Stephen Smith

Apress®

Maker Innovations Series

Jump start your path to discovery with the Apress Maker Innovations series! From the basics of electricity and components through to the most advanced options in robotics and Machine Learning, you'll forge a path to building ingenious hardware and controlling it with cutting-edge software. All while gaining new skills and experience with common toolsets you can take to new projects or even into a whole new career.

The Apress Maker Innovations series offers projects-based learning, while keeping theory and best processes front and center. So you get hands-on experience while also learning the terms of the trade and how entrepreneurs, inventors, and engineers think through creating and executing hardware projects. You can learn to design circuits, program AI, create IoT systems for your home or even city, and so much more!

Whether you're a beginning hobbyist or a seasoned entrepreneur working out of your basement or garage, you'll scale up your skillset to become a hardware design and engineering pro. And often using low-cost and open-source software such as the Raspberry Pi, Arduino, PIC microcontroller, and Robot Operating System (ROS). Programmers and software engineers have great opportunities to learn, too, as many projects and control environments are based in popular languages and operating systems, such as Python and Linux.

If you want to build a robot, set up a smart home, tackle assembling a weather-ready meteorology system, or create a brand-new circuit using breadboards and circuit design software, this series has all that and more! Written by creative and seasoned Makers, every book in the series tackles both tested and leading-edge approaches and technologies for bringing your visions and projects to life.

More information about this series at https://link.springer.com/bookseries/17311.

RISC-V Assembly Language Programming

Unlock the Power of the RISC-V Instruction Set

Stephen Smith

Apress®

RISC-V Assembly Language Programming: Unlock the Power of the RISC-V Instruction Set

Stephen Smith
Gibsons, BC, Canada

ISBN-13 (pbk): 979-8-8688-0136-5 ISBN-13 (electronic): 979-8-8688-0137-2
https://doi.org/10.1007/979-8-8688-0137-2

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Miriam Haidara
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Paper in this product is recyclable

This book is dedicated to my beloved wife and editor Cathalynn Labonté-Smith.

Table of Contents

About the Author

Stephen Smith is a software architect, located in Gibsons, BC, Canada. He's been developing software since high school, or way too many years to record. He is an expert in Artificial Intelligence and Assembly Language programming, earned his Advanced HAM Radio License, and enjoys mountain biking, hiking, and nature photography. He volunteers for Sunshine Coast Search and Rescue. He is the author of *Raspberry Pi Assembly Language Programming: ARM Processor Coding*, *Programming with 64-Bit ARM Assembly Language: Single Board Computer Development for Raspberry Pi and Mobile Devices*, and *RP2040 Assembly Language Programming: ARM Cortex-M0+ on the Raspberry Pi Pico*, all published by Apress. Also, he writes his popular technology blog, at smist08.wordpress.com.

About the Technical Reviewer

Stewart Watkiss is a keen maker who has created numerous physical computing projects using a variety of computers and microcontrollers. He is author of the Apress titles *Learn Electronics with Raspberry Pi* and *Beginning Game Programming with Pygame Zero*. He studied at the University of Hull, where he earned a master's degree in electronics engineering, and more recently at Georgia Institute of Technology, where he earned a master's degree in computer science.

Stewart also volunteers as a STEM ambassador, helping teach programming and physical computing to schoolchildren and at Raspberry Pi events. He has created numerous resources for those wanting to learn more about electronics and computing which are available on his website (`www.penguintutor.com`).

Acknowledgments

No book is ever written in isolation. I want to especially thank my wife Cathalynn Labonté-Smith for her support, encouragement, and expert editing.

I want to thank all the good folks at Apress who made the whole process easy and enjoyable. A special shout-out to Nirmal Selvaraj, my production editor, who kept the whole project moving quickly and smoothly. Thanks to Miriam Haidara, the acquisitions editor, for mobile tech and maker programs, who got the project started. Thanks to Stewart Watkiss, my technical reviewer, who helped make this a far better book.

Introduction

The heart of every computer and smart device is a Central Processing Unit (CPU). Historically, each CPU has been produced by a single company which tightly controls the instruction set and internal workings of the CPU. RISC-V (pronounced Risk Five) is a new approach based on open-source principles. Anyone can implement a RISC-V CPU using a standardized instruction set without requiring any special licensing or royalty payments. As a result, RISC-V CPUs can be at a much lower cost than proprietary CPUs and have exploded in the low-cost microcontroller space. Now that more powerful RISC-V CPUs are appearing, leading to Single Board Computers (SBCs) similar to the Raspberry Pi, along with inexpensive low-end laptops and tablets, this book presents the inner workings of typical RISC-V CPUs and details the open-source Assembly Language instruction set they all implement.

Assembly Language is the native, lowest level way to program a computer. Each processing chip has its own Assembly Language. This book teaches programming RISC-V CPUs either running 32- or 64-bits.

Learning how a computer works, and mastering Assembly Language, is an excellent way to get into the nitty-gritty details. These low-cost microcontrollers and SBCs provide ideal platforms to learn advanced concepts in computing.

Even though all of these devices are low-powered and compact, they're still sophisticated computers, many of which are capable of running the full Linux operating system. Due to the RISC-V instruction set standard, learning to program on one RISC-V processor is directly applicable to all RISC-V processors.

In this book, we cover how to program RISC-V processors at the lowest level, operating as close to the hardware as possible. Readers will learn the following:

- How to format instructions and combine them into programs, as well as details of the operative binary data formats
- How to program RISC-V instruction set extensions, such as floating-point instructions
- How to control integrated hardware devices by reading and writing to the hardware control registers directly
- How to interact with the Linux operating system and microcontroller SDKs

The simplest way to learn these tasks is with a RISC-V SBC such as the Starfive Visionfive 2. The book also details how to emulate the RISC-V CPU on an Intel/AMD computer using the QEMU emulator, as well as how to program the Espressif ESP32-C3 microcontroller. All the tools needed to learn Assembly Language programming are open source and readily available.

This book contains many working programs to play with, use as a starting point, or study. The only way to learn programming is by doing it, so do not be afraid to experiment, as it is the only way to learn.

Even if Assembly Language programming isn't used in your day-to-day life, knowing how the processor works at the Assembly Language level and knowing the low-level binary data structures will make you a better programmer in all other areas. Knowing how the processor works will let you write more efficient C code and can even help with Python programming.

Enjoy this introduction to Assembly Language. Learning it for one processor family helps with learning and using any other processor architectures encountered throughout a programmer's career.

CHAPTER 1

Getting Started

Most people are familiar with Intel or AMD microprocessors lying at the heart of their desktop, laptop, or server, and, similarly, most cell phones and tablets use ARM microprocessors. RISC-V is the new entry built around open-source concepts. Before getting into the details of RISC-V, let's first look at the history and evolution of modern microprocessors.

History and Evolution of the RISC-V CPU

At the heart of every computer, there is at least one Central Processing Unit (CPU) that executes the programs that you run. Most modern computers have several CPUs, the main one to run the operating system and user programs, then several helper CPUs to offload tasks like network communications. Solid-state drives (SSDs) and hard drives (HDDs) each contain a CPU to read and store the data. Nearly every household appliance from microwaves to thermostats contains a CPU that interprets button pushes and performs the appliance's function. A modern automobile typically contains 1000 to 3000 CPU chips to control everything from the infotainment system to the power steering to the braking systems. CPUs range in price from a few cents to thousands of dollars, and each runs its own special machine code to execute programs.

Early CPUs, such as the IBM 360 mainframe, were large, complicated, and comprised of thousands of discrete components. In 1971, Intel debuted the Intel 4004, a single chip 4-bit CPU that allowed the

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_1

proliferation of handheld calculators. This led to more advanced CPUs such as the MOS Technology 6502 which had the right combination of being both affordable yet powerful enough to create both the personal computer and video game industries with products like the Apple II and Nintendo Entertainment System.

IBM entered the PC market in 1981 using the more complex Intel 8088 CPU. This CPU was powerful and flexible with 16-bit addressing and variable length machine code instructions. This started a trend of single-chip CPUs adding complexity with the goal of becoming mainframes on the desktop. Intel quickly introduced new chips with more and more advanced functionalities to empower 32-bit and then 64-bit processing, vector operations, memory protection, multi-tasking, and more. This process added to the chip's complexity along with the underlying instruction set.

Competitors saw this and thought they could compete with Intel’s Complex Instruction Set Computers (CISC) by introducing simplified Reduced Instruction Set Computers (RISC). These computers wouldn’t have the baggage of maintaining compatibility with older chips, they would start out at 32- or 64-bits. Their instructions would be a fixed width and there would be fewer of them to allow faster instruction decoding. Each instruction would run in 1 clock cycle. Early RISC processors were typically used in high-end UNIX workstations, so consequently they never achieved the shipment volume and economies of scale to effectively compete with Intel.

In 1982, Acorn Computers, which produced the BBC Microcomputer, was looking to upgrade their line of computers from using the now aging 6502. The engineers at Acorn did not want to move to the Intel architecture and become another PC clone. Instead, they wanted the simplicity of the 6502, but modernized. They took the daring step of designing their own CPU, called the ARM CPU based on the RISC philosophy.

If the ARM was only used in Acorn Computers, it probably would have died out. However, towards the end of the 1990s, Steve Jobs decided to use the ARM processor in the Apple iPod. The reason being that because of its simplified instruction set, an ARM processor didn't use much power, and this allowed longer battery life in a portable device.

With the success of the iPod, ARM reached a reasonable production volume for more research and development (R&D), then with the introduction of the iPhone, ARM usage exploded as ARM processors are used in nearly every phone and tablet in production. Intel finally had competition from a RISC processor.

Today, most Intel processors implement a RISC core that executes the Intel machine code using multiple RISC instructions. ARM has become more complex as it supports both 32- and 64-bit instructions using different formats. A major complaint against both is that to use compatible machine code with either ARM or Intel processors requires expensive licensing agreements with the parent companies.

Enter RISC-V, an open standards RISC-based machine, where anyone can produce a compatible CPU without paying anyone licensing fees. Creating a new CPU machine language is expensive as you need high-level language compilers and operating support. If you create a CPU that executes RISC-V instructions, then you can use the standard RISC-V version of Linux and the standard GNU language toolchain for compiling high-level languages. The RISC-V project started in 2010 at the University of California, Berkeley, and RISC-V is the fifth version of the architecture. It supports 32-, 64-, and 128-bit processors. It is designed to run on everything from microcontrollers costing pennies to superscalar supercomputers costing millions. We've reached a point where many RISC-V CPUs and computers are appearing. This book will explore the architecture and describe how to program these RISC-V CPUs at the lowest machine code level.

What You Will Learn

You will learn Assembly Language programming for RISC-V CPUs running in 32- or 64-bit mode. Everything you will learn is directly applicable to all RISC-V devices from the smallest microcontrollers to the largest superscalar supercomputers. Learning Assembly Language for one processor gives you the tools to learn it for another processor, such as ARM processors typically used in mobile devices.

In all devices, the RISC-V processor isn't just a CPU, it's a system on a chip. This means that most of the computers are all on one chip. When a company is designing a device, they can select various modular components to include on their chip. Typically, this contains a RISC-V processor with multiple cores, meaning that it can process instructions for multiple programs running at once. It likely contains several co-processors for things like floating-point calculations, a Graphics Processing Unit (GPU), and various communications protocols.

The RISC-V instruction set is modular. Each RISC-V CPU will implement a set of modules to provide the functionality its users require. First there is a base module consisting of the basic integer instructions. There are three common base instruction sets—one for 32-bit, one for 64-bit, and a slimmed down subset of the 32-bit one for minimal embedded processors.

The base set provides sufficient functionality to perform integer arithmetic, logic, comparisons, and branches. Then additional instructions are added in optional modules, for instance, integer multiplication and division are in the "M" module. Low-cost embedded processors will only implement a few basic modules, whereas a more powerful CPU intended to be used in a full Linux-based computer might implement a dozen modules. Table 1-1 contains the three main base versions of the instruction sets that are covered in this book.

Table 1-1. *Base Instruction Sets*

Name	Description
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit
RV64I	Base Integer Instruction Set, 64-bit

Table 1-2 contains the main instruction set extensions. For a complete list, consult the specifications posted at riscv.org.

Table 1-2. *Instruction Set Extensions*

Name	Description
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
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
G	Shorthand for the IMAFDZicsr_Zifencei base and extensions
C	Standard Extension for Compressed Instructions

Working at this low level is technical and time-consuming, so why would a programmer want to write Assembly Language code?

Ten Reasons to Learn Assembly Language Programming

Most programmers write in a high-level programming language like Python, C#, Java, JavaScript, Go, Julia, Scratch, Ruby, Swift, or C. These highly productive languages are used to write major programs from the Linux operating system to websites like Facebook, to productivity software like LibreOffice. If you learn to be a good programmer in a couple of these, you can potentially find a well-paying interesting job and write some great programs. If you create a program in one of these languages, you can easily get it working on numerous operating systems on multiple hardware architectures. You never have to learn the details of all the bits and bytes, and these can remain safely under the covers.

When you program in Assembly Language, you are tightly coupled to a given CPU and moving your program to another requires a complete rewrite of your program. Each Assembly Language instruction does only a fraction of the amount of work, so to do anything takes a lot of Assembly statements. Therefore, to do the same work as, say, a Python program takes an order of magnitude larger amount of effort, for the programmer. Writing in Assembly Language is harder, as you must solve problems with memory addressing and CPU registers that are all handled transparently by high-level languages. So why would you want to learn Assembly Language programming?

Here are 10 reasons to learn and use Assembly Language:

1. **To write more efficient code**: Even if you don't write Assembly Language code, knowing how the computer works internally allows you to write more streamlined code. You can make your data structures easier to access and write code in a style that allows the compiler to generate more effective code. You can make better use of computer resources, like co-processors, and use the given computer to its fullest potential.

2. **To write your own operating system**: The core of the operating system that initializes the CPU handles hardware security and multi-threading/multi-tasking requires Assembly code.
3. **To create a new programming language**: If it is a compiled language, then you need to generate the Assembly code to execute. The quality and speed of your language is largely dependent on the quality and speed of the Assembly Language code it generates.
4. **To make computers run faster**: The best way to make Linux faster is to improve the GNU C Compiler. If you improve the RISC-V Assembly Language code produced by GNU C, then every program compiled by GCC benefits.
5. **To interface a computer to a hardware device**: When interfacing a computer through USB or GPIO ports, the speed of data transfer is highly sensitive as to how fast a program can process the data. Perhaps, there are a lot of bit-level manipulations that are easier to program in Assembly Language.
6. **To do faster machine learning or 3D graphics programming**: Both applications rely on fast matrix mathematics. If you can make this faster with Assembly and/or using the co-processors, then you can make AI-based robots or video games that much better.

7. **To boost performance**: Most large programs have components written in different languages. If a program is 99% C++, the other 1% could be Assembly, perhaps giving a program a performance boost or some other competitive advantage.

8. **To create a new single-board computer**: New boards have some Assembly Language code to manage peripherals included with the board. This code is usually called a BIOS (basic input/output system).

9. **To look for security vulnerabilities in a program or piece of hardware**: Look at the Assembly code to do this, otherwise you may not know what is really going on, and hence where holes might exist.

10. **To look for Easter eggs in programs**: These are hidden messages, images, or inside jokes that programmers hide in their programs. They are usually triggered by finding a secret keyboard combination to pop them up. Finding them requires reverse engineering the program and reading Assembly Language.

Now that we have some background on Assembly Language, we'll look at some choices for running RISC-V Assembly Language code.

Running Programs on RISC-V Systems

Due to the modular nature of RISC-V, there are a lot of possibilities for running programs. This book will provide details on the following three specific systems:

1. **RISC-V-based Linux computer such as the Starfive Visionfive 2**: This is the easiest way to play with RISC-V Assembly Language, since everything is done on the same computer. The CPU on the Visionfive 2 board supports the RV64GC set of RISC-V extensions.
2. **RISC-V simulator running on an Intel/AMD-based Windows or Linux computer**: Not everyone has a RISC-V computer yet, but don't let that stop you. You can simulate RISC-V running full Linux.
3. **RISC-V microcontroller**: There are many of these such as the Espressif ESP32-C3 DevKit. First of all, write a program, then compile it on another computer, such as a Linux laptop, and finally download the compiled program to the microcontroller to run. This is an important usage of RISC-V Assembly Language, but you will be limited to a few basic instruction modules, namely, RV32IMC.

There are many other possibilities that also work, but you will need to adapt the instructions for the supported systems. Next, we look at how to set up each of these to assemble and run a simple Assembly Language program to print out "Hello RISC-V World!"

Coding a Simple "Hello World" Program

Computers operate on binary data consisting of zeros and ones; however, humans do not think this way. As a result, humans develop tools to assist us in interacting with computers. One such tool is the GNU Assembler which takes a human readable version of an Assembly Language

program and converts it to the zeroes and ones that the RISC-V processor understands. In this section, we will present a simple program without going into detail. The details will be filled in over the following chapters. In this chapter, we'll take the program and look at several ways of Assembling and running it. First, we'll look at running it on the Starfive Visionfive 2 SBC under Linux.

Hello World on the Starfive Visionfive 2

The state of Linux on RISC-V computers is evolving quickly. At the time of writing this book, the Debian Linux install image for the Starfive Visionfive 2 computer is quite minimal and does not even include a web browser. Please follow the instructions in its Quick Start Guide for the most up-to-date information. These instructions will also give an idea of what is required for other RISC-V-based Linux SBCs.

At the time of the writing of this book, these are the instructions to set up the Visionfive 2. The steps are given in more detail in the Visionfive's Quick Start Guide. The Visionfive can run Linux from a microSD card, an M.2 SSD drive, or from its internal firmware.

Before you start, you will need the following:

- The Starfive Visionfive 2 board, USB keyboard, USB mouse, and HDMI capable monitor
- An microSD (Secure Digital) card
- A burner program, like Balena Etcher
- Secure Shell (SSH) program
- Quick Start Guide
- Internet connection

First, configure two dip switches on the motherboard to boot from the preferred device. These instructions follow what is needed to boot from a microSD card, but as long as the correct instructions are followed, you can run Linux from any supported configuration.

1. Configure the two dip switches to eMMC.
2. Download the Debian Linux image for an SD card from Starfive's website: https://debian.starfivetech.com/. Next, burn it onto an SD card using a program like Balena Etcher.
3. Place the SD card in the Visionfive 2 and turn the Visionfive on.
4. If you are using a hardwired Internet connection, you can skip this step. When it boots to Linux, login, the password is "starfive." Run the Setup program and select your Wifi and enter the password.
5. Since the base image doesn't contain a web browser, it is easiest to perform the following steps using SSH from a host computer on the same Wifi network. This way you can copy/paste commands from the Quick Start Guide into the SSH program. Use "starfive.local" as the host name.

   ```
   ssh user@starfive.local
   ```

6. Next, the file system needs to be resized. If you don't do this, you can't install any programs, since the base image has little free space. This is a technical step and you should follow the instructions from the Quick Start Guide carefully as you are deleting and recreating disk partitions.

7. Now we perform the long step of installing the extra software that includes web browsers and the development tools required for this book. Use the scp command to copy the "install_package_and_dependencies.sh" script to the computer and run it. Enter the scp command from the host computer:

   ```
   scp install_package_and_dependencies.sh user@starfive.local:/home/user
   ```

8. Then the following commands from the SSH remote session.

   ```
   chmod +x install_package_and_dependencies.sh
   ./install_package_and_dependencies.sh
   ```

   Running this takes several hours, so it's a good time to go for lunch. After this script completes, reboot the Visionfive 2.

9. Part of the previous shell script installs the GNU development tools. If using a different Linux distribution of SBC, this may need to be done manually using the command:

   ```
   sudo apt install build-essential
   ```

10. After the computer reboots, login normally. There are now web browsers, office tools, and software development tools.

11. Install a simple GUI text editor to use for programming. Use a terminal-based text editor like vi if you wish. GEdit is a good simple choice that can be installed with the following command:

    ```
    sudo apt install gedit
    ```

With the computer setup, we are ready to write, assemble, and run a simple "Hello World" program. In this chapter, we won't worry about the details of how this program works, rather we are ensuring we can assemble, link, and run programs. We will examine how everything works in detail in the following chapters.

Either download the source code from the Apress Github site or type in the program in Listing 1-1 and save it as **HelloWorld.S**, where capitalization is important.

Listing 1-1. The Linux version of the Hello World program

```
#
# Risc-V Assembler program to print "Hello RISC-V World!"
# to stdout.
#
# a0-a2 - parameters to linux function services
# a7 - linux function number
#

.global _start      # Provide program starting address
                    to linker

# Setup the parameters to print hello world
# and then call Linux to do it.

_start: addi  a0, x0, 1      # 1 = StdOut
        la    a1, helloworld # load address of helloworld
        addi  a2, x0, 20     # length of our string
        addi  a7, x0, 64     # linux write system call
        ecall                # Call linux to output the string
```

```
# Setup the parameters to exit the program
# and then call Linux to do it.

        addi    a0, x0, 0   # Use 0 return code
        addi    a7, x0, 93  # Service command code 93
                              terminates
        ecall               # Call linux to terminate
                              the program

.data
helloworld:      .ascii "Hello RISC-V World!\n"
```

Anything after a hash sign (#) is a comment. The code lines consist of an optional label, followed by an opcode, possibly followed by several parameters. To compile and run this program, create a file called **build** that contains the contents of Listing 1-2:

Listing 1-2. Build file for Hello World

```
as -o HelloWorld.o HelloWorld.S
ld -o HelloWorld HelloWorld.o
```

After saving the build file, convert it to an executable file with the following command:

```
chmod +x build
```

Now compile the program by running the build file and then the resulting executable program with following commands:

```
./build
./HelloWorld
```

Figure 1-1 shows the result of running this. bash -x is used to show the commands being executed.

```
user@starfive: ~/Chapter1
user@starfive:~/Chapter1$ bash -x ./build
+ as -o HelloWorld.o HelloWorld.S
+ ld -o HelloWorld HelloWorld.o
user@starfive:~/Chapter1$ ./HelloWorld
Hello RISC-V World!
user@starfive:~/Chapter1$
```

Figure 1-1. *Compiling and running the Hello World program*

We've now written and executed our first Assembly Language program. Next, let's look at running Hello World on a RISC-V emulator.

Programming Hello World in the QEMU Emulator

Even without owning a genuine RISC-V processor, you can still emulate a RISC-V-based computer using the QEMU emulator. QEMU is similar to virtualization software like VMWare, except that the computer being virtualized isn't required to have the same CPU as the host system.

In this case, we'll give instructions on how to install and run the QEMU emulator on a Windows computer and emulate/virtualize a RISC-V-based Linux system. Next, we will run exactly the same programs that we run on the genuine RISC-V-based Starfive Visionfive 2.

Install QEMU on Windows

Here are the instructions to install QEMU on Windows and get a RISC-V version of Linux up to the first login prompt. Before you start, you will need the following:

- QEMU software.
- 7-Zip or any uncompression program that understands xz format.

- `fw_jump.elf` and `uboot.elf`. These are easy to install on Linux, but a bit hard to find for Windows, so they are placed on the Apress Github site under Chapter 1.

The QEMU documentation is extensive, but overwhelming. The following are the steps required for this book:

1. Install QEMU. Download the Windows install image. Run the downloaded program saying Yes to the Windows security prompts.
2. Add c:\Program Files\Qemu to the system PATH, assuming the default installation folder.
3. Download the Ubuntu Server preinstalled image from `https://ubuntu.com/download/risc-v`.
4. Use 7-Zip or any uncompression program that understands xz format to extract the .img file from what you downloaded.
5. Rename the .img file as ubunturv.img, or change the name in the commands that follow.
6. Add free space to the image with

   ```
   qemu-img resize ubunturv.img +10G
   ```

   This gives room to install programs once you start Linux.
7. To run the image, use `fw_jump.elf` and `uboot.elf`. These then act as the emulated board's BIOS.
8. Now we are ready to run QEMU with the rather long command:

   ```
   qemu-system-riscv64 ^
           -machine virt ^
   ```

```
-cpu rv64 ^
-m 2G ^
-device virtio-blk-device,drive=hd ^
-drive file=ubunturv.img,if=none,id=hd ^
-device virtio-net-device,netdev=net ^
-netdev user,id=net,hostfwd=tcp::2222-:22 ^
-bios fw_jump.elf ^
-kernel uboot.elf ^
-append "root=LABEL=rootfs console=ttyS0" ^
-nographic
```

In a Windows **cmd** prompt, at the time of this writing, not all the cursor and other special characters translate properly. Rather than logining in at the Linux prompt directly, use a **ssh** command to login with

```
ssh -p 2222 ubuntu@localhost
```

Then in the **ssh** window, the cursor keys and other special characters work properly.

Install QEMU on Linux

Now let's do the same thing for Linux.

1. Install QEMU with

   ```
   sudo apt install qemu-system-riscv64
   sudo apt install u-boot-qemu opensbi
   ```

2. Download the Ubuntu Server preinstalled image from https://ubuntu.com/download/risc-v.

3. From the downloaded Ubuntu image, extract the img file from the xz file with

   ```
   unxz < downloaded_file_name.img.xz > ubunturv.img
   ```

4. Expand the image, so there is some disk space to play around with

   ```
   sudo qemu-img resize ubunturv.img +10G
   ```

5. Now we are ready to run with the rather long qemu command:

   ```
   qemu-system-riscv64 \
           -machine virt \
           -cpu rv64 \
           -m 2G \
           -device virtio-blk-device,drive=hd \
           -drive file=ubunturv.img,if=none,id=hd \
           -device virtio-net-device,netdev=net \
           -netdev user,id=net,hostfwd=tcp::2222-:22 \
           -bios /usr/lib/riscv64-linux-gnu/opensbi/
            generic/fw_jump.elf \
           -kernel /usr/lib/u-boot/qemu-riscv64_smode/
            uboot.elf \
           -object rng-random,filename=/dev/urandom,id=rng \
           -device virtio-rng-device,rng=rng \
           -append “root=LABEL=rootfs console=ttyS0” \
           -nographic
   ```

Compiling in Emulated Linux

Whether using Windows or Linux to host QEMU, booting Linux under the emulator will eventually lead you to a login prompt. The initial user id/password is ubuntu/ubuntu. During the first time logging in, Linux will

force to change the password. The computer name is also ubuntu which can be changed if needed. Once logged in, we can add the GNU Compiler Collection with

```
sudo apt update
sudo apt install build-essential
```

This will also install **GDB** and a few other **GCC**-related tools.

Next, either type the **HelloWorld.S** program from the previous section into **vi** to save it to the new image or copy the program to this image using **scp**:

```
scp -P 2222 HelloWorld.S ubuntu@localhost:/home/ubuntu
```

Then compile and run the program:

```
as  -o HelloWorld.o HelloWorld.S
ld -o HelloWorld HelloWorld.o
./HelloWorld
```

Figure 1-2 shows the output of this process.

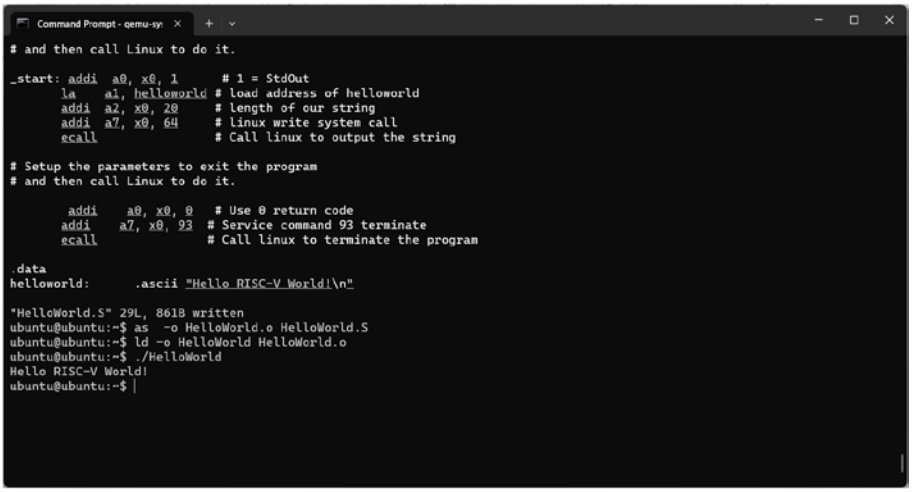

Figure 1-2. *Compiling and running Hello World in the QEMU emulator*

Now, let's look at running Hello World on a Espressif ESP32-C3 Devkit microcontroller.

About Hello World on the ESP32-C3 Microcontroller

The Espressif ESP32-C3 Devkit is a RISC-V-based microcontroller similar to a Raspberry Pi Pico. There are many RISC-V microcontrollers on the market, but this one has an extensive SDK allowing programming in MicroPython, C/C++, or Assembly Language. The documentation is extensive and Espressif has a lot of experience in the microcontroller world.

When developing for a microcontroller, you need a host computer where you write the code and compile your program. You then download the resulting program to run, usually through a USB cable. The host computer can be a Windows, MacOS, or Linux-based computer. The online Espressif *Getting Started Guide* is recommended to install the Espressif SDK on your computer. The examples in this book will be developed on a Windows-based computer, but this doesn't really matter. To install the Espressif SDK on Windows is quite simple, just requiring that a standard Windows installation program be run.

Projects in the Espressif SDK use the CMake system for building and are more complex than what we have seen so far. The easiest way to get a project running quickly is to take one of the SDK's example projects, then transform it into what is required. The Espressif SDK contains a Hello World program written in C. Let's convert it to a Hello World program written entirely in Assembly Language. Take the example program located at the following link: C:\Espressif\frameworks\esp-idf-v5.0.2\examples\get-started\hello_world, to create our hello_world project:

1. Copy the SDK example hello_world project to a new folder, say hello_world_asm.
2. Change main\CMakeLists.txt to replace the source file hello_world_main.c with hello_world_main.S.
3. Delete main\hello_world_main.c and create main\hello_world_main.S from Listing 1-2.

The "Hello World" program is a little different than the Linux version and contained in Listing 1-3:

Listing 1-3. Espressif version of the Hello World program

```
#
# Risc-V Assembler program to print "Hello RISC-V World!"
# to the Espressif SDK monitor program through the microUSB.
#
# a0 - address of helloworld string
#

.global app_main    # Provide program starting address
                      to linker

# Setup the parameter to print hello world
# and then call SDK to do it.

app_main:
        la a0, helloworld  # load address of helloworld
        call puts          # Call sdk to output the string
        j app_main         # loop forever

.data
helloworld:      .asciz "Hello RISC-V World!"
```

We build the program with:

```
idf.py build
```

Note Remember to use the special version of the Command (CMD) prompt on your desktop that the Espressif SDK install added. This CMD contains all the correct environment variables required for a successful build.

The first time CMD runs, it takes a long time since it needs to build the entire SDK as part of the process. On subsequent builds, it is much quicker as it only needs to build the files that have changed. Figure 1-3 shows the output of the build process on one of these following builds.

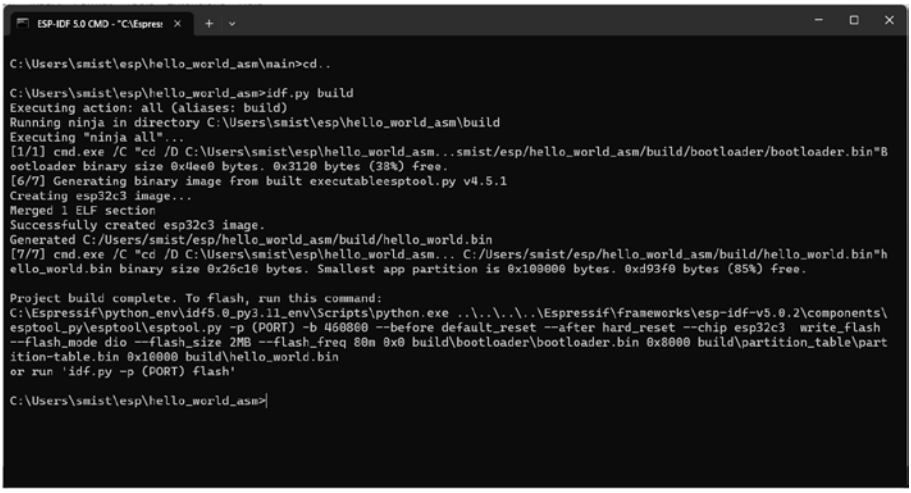

Figure 1-3. *Building the Espressif version of the Hello World program*

And then run it with the following:

```
idf.py -p com4 flash monitor
```

Figure 1-4 shows the end of the download process, followed by the start of the program.

Note The required com port might not be com4, to check the correct port, run the Windows Device Manager, open the "Ports (COM & LPT)" section, and note which COM port is associated with the "Silicon Labs CP210x USB to UART Bridge."

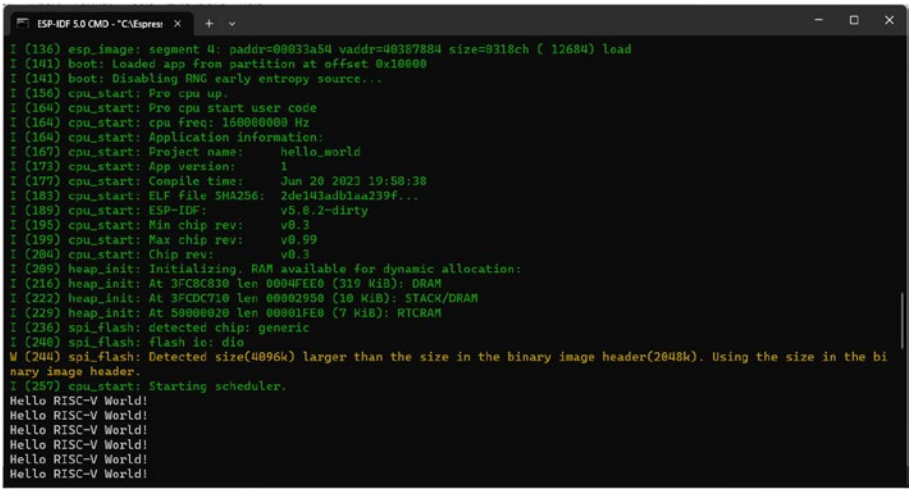

Figure 1-4. *Running Hello World on the Espressif RISC-V microcontroller*

In the Linux version of the program, the program terminated after printing the "Hello RISC-V World!" string once. Microcontrollers don't have an operating system to exit to, so they are usually implemented as an infinite loop that runs forever, as was done in this simple example. To exit the monitor, use **Control-]**.

Summary

In this chapter, we introduced the RISC-V processor and Assembly Language programming along with why Assembly Language is used. We covered how to set up three possible development environments for Assembly Language programming using RISC-V.

In Chapter 2, "Loading and Adding," we will start to understand the details of the programs presented, including the basics of Assembly Language instructions and CPU registers. We'll learn how numbers are represented in a computer and how basic integer addition and subtraction work.

Exercises

1. Setup your RISC-V development environment. Whether it is on a Linux Single Board Computer (SBC) such as the Starfive Visionfive 2, running in the QEMU emulator or using a microcontroller like the Espressif ESP32-C3.
2. Modify the "Hello RISC-V World!" string to a different string. For the Linux versions, remember to modify the string length as well.

CHAPTER 2

Loading and Adding

The goal of this chapter is to load various integer values into the CPU and perform basic addition and subtraction operations on these values. This sounds simple, but at the low level of Assembly Language, this can get complicated. In this chapter, we will go step-by-step through the various forms of the **add**, **sub**, and **Shift** instructions, to lay the groundwork on how all instructions work, especially in the way they handle parameters. This is so that in subsequent chapters, we can proceed at a faster pace as we encounter the rest of the RISC-V instruction set.

Before getting into the various Assembly Language instructions, we will discuss how the RISC-V CPU represents positive and negative numbers.

Computers and Numbers

We typically represent numbers using base ten. The common theory is we do this because we have ten fingers to count with. This means a number like 387 is really a representation for

$$\begin{aligned} 387 &= 3 * 10^2 + 8 * 10^1 + 7 * 10^0 \\ &= 3 * 100 + 8 * 10 + 7 \\ &= 300 + 80 + 7 \end{aligned}$$

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_2

There is nothing special about using ten as our base, and a fun exercise in math class is to do arithmetic using other bases. In fact, the Mayan culture used base 20, perhaps because we have 20 digits: ten fingers and ten toes.

Computers don't have fingers and toes, just switches that are either on or off. As a result, it is natural for computers to use base two arithmetic. Thus, to a computer, a number like 1011 is represented by the following:

$$
\begin{aligned}
1011 &= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\
&= 1 * 8 + 0 * 4 + 1 * 2 + 1 \\
&= 8 + 0 + 2 + 1 \\
&= 11 \text{ (decimal)}
\end{aligned}
$$

This is great for computers, but we use four digits for the decimal number 11 rather than two digits. The big disadvantage for humans is that writing out binary numbers is tiring, because they take up so many digits.

Computers are incredibly structured, so all their numbers are the same size. When designing computers, it doesn't make sense to have a variety of differently sized numbers, so a few common sizes have become standard as described in the following paragraph.

A byte is eight binary bits or digits. In our preceding example with four bits, there are 16 possible combinations of 0s and 1s. This means four bits can represent the numbers 0 to 15. This means it can be represented by one base 16 digit. Base 16 digits are represented by the numbers 0 to 9 and then the letters A–F for 10–15. We can then represent a byte (eight bits) as two base 16 digits. We refer to base 16 numbers as hexadecimal (Figure 2-1).

Decimal	0 - 9	10	11	12	13	14	15
Hex Digit	0 – 9	A	B	C	D	E	F

Figure 2-1. *Representing hexadecimal digits*

Since a byte holds eight bits, it can represent 28 (256) numbers. Thus, the byte E6 represents

```
E6 = e * 16¹ + 6 * 16⁰
   = 14 * 16 + 6
   = 230 (decimal)
   = 1110 0110 (binary).
```

We call a 32-bit quantity a word, and it is represented by four bytes. You might see a string like B6 A4 44 04 as a representation of 32 bits of memory, or one word of memory, or perhaps the contents of one register. If we use a 64-bit RISC-V processor, then a word is still 32-bits, and 64-bit quantities are referred to as doublewords, and we would see a string of eight bytes to represent one doubleword of data.

If this is confusing or scary, do not worry. The tools will do all the conversions for you. It's just a matter of understanding what is presented to you on the screen. Also, if an exact binary number needs to be specified, that is usually done in hexadecimal, although all the tools accept all the formats.

A handy tool is the Linux Gnome Calculator (Figure 2-2). The Gnome Calculator has a Programming Mode which shows a numbers representation in multiple bases at once. This calculator is included with Ubuntu Linux with the Gnome desktop. To install it on the Visionfive 2, enter the following command line:

```
sudo apt-get install gnome-calculator
```

In "Programmer Mode," conversions can be done with numbers shown in several formats at once.

Figure 2-2. *The Gnome Calculator*

This is how data is represented on a computer. There is more complexity in how signed integers are represented and how arithmetic works.

Negative Numbers

In the previous section, we discussed how computers represent positive integers as binary numbers, called unsigned integers, but what about negative numbers? An initial thought may be to make one-bit represent whether the number is positive or negative. This is simple, but it turns out it requires extra logic to implement, since now the CPU must look at the sign bits, then decide whether to add or subtract and in which order.

There is a simple representation of negative numbers that works without any special cases or special logic called two's complement.

About Two's Complement

The great mathematician, John von Neumann, of the Manhattan Project, came up with the idea of the **two's complement** representation for negative numbers, in 1945, when working on the Electronic Discrete Variable Automatic Computer (EDVAC) computer—one of the earliest electronic computers.

Two's complement originated by observing how addition overflows. Consider a one-byte hexadecimal number like 01, when 0xFF (all binary ones) is added:

```
0x01 + 0xFF = 0x100
```

The result is 0x100.

However, if limited to 1-byte numbers, then the 1 is lost resulting in 00:

```
0x01 + 0xFF = 0x00
```

The mathematical definition of a number's negative is a number that when added to it makes zero, mathematically, FF is -1. Two's complement is formed from any number by taking

2^N - number

where N is the number of bits in the integer. In the example, the two's complement of 1 is

2^8 - 1 = 256 - 1 = 255 = 0xFF

This is why it's called two's complement. An easier way to calculate the two's complement is to change all the 1s to 0s and all the 0s to 1s, then add 1. For example, if this is done to 1 this is the result:

```
0xFE + 1 = 0xFF
```

Two's complement is an interesting mathematical oddity for integers that are limited to having a maximum value of one less than a power of two (which is all computer representations of integers).

Why would we represent negative integers this way on computers? As it turns out, this makes addition simple for the computer to execute. Adding signed integers is the same as adding unsigned integers. There are no special cases, just discard the overflow and everything works out. This means less circuitry is required to perform the addition, and as a result it can be performed faster. Consider the following example:

```
5 + -3
```

3 in 1-byte is 0x03 or 0000 0011

Inverting the bits is

```
1111 1100
```

Add 1 to get

```
1111 1101 = 0xFD
```

Now add

```
5 + 0xFD = 0x102 = 2
```

Since we are limited to 1 byte or 8 bits.

Performing these computations by hand is educational, but practically tools, such as the Gnome Calculator from the previous section, do the work on our behalf. Now that we know how numbers are represented, we can look at Assembly Language instructions.

RISC-V Assembly Instructions

In this section, basic architectural elements of the RISC-V processor are introduced as well as the form of its machine code instructions. Being a RISC computer theoretically makes learning Assembly easier. There are fewer instructions and each one is simple, so the processor can execute each instruction quickly.

The first few chapters of this book cover the standard RISC-V integer Assembly Language instructions. Therefore, the following topics are deferred to later chapters where they are covered in detail to avoid confusion:

- Interacting with other programming languages
- Accessing hardware devices
- Instructions for the floating-point processor

In technical computer topics, there are often chicken and egg problems in presenting the material. The purpose of this section is to introduce terms and concepts used later, so they are familiar when they are covered in full detail.

CPU Registers

In all computers, data is not operated in the computer's memory, rather it's loaded into a CPU register, then the data processing or arithmetic operation is performed on the registers. The registers are part of the CPU circuitry allowing instant access, whereas memory is a separate component and there is a transfer time for the CPU to access it.

The RISC-V processor is based on a load-store architecture where there are two basic types of instructions:

1. Instructions that either load memory into registers or store data from registers into memory
2. Instructions that perform arithmetical or logical operations between two registers

To add two numbers, do the following:

1. Load one number into one register and the other number into another register.
2. Perform the add operation on the two registers putting the result into a third register.
3. Copy the answer from the results register into memory.

As the preceding instructions show, it can take quite a few instructions to perform simple operations.

The RISC-V processor has access to 31 general purpose integer registers, a program counter (pc) and a zero register.

- **x0:** The zero register, hardcoded to all bits zero
- **x1** to **x31:** These 31 registers are general purpose that can be used for anything, although some have standard agreed upon usage that will be covered later.
- **pc:** The program counter, that is, the memory address of the currently executing instruction.

Note If using a 64-bit RISC-V processor (RV64I), then each register is 64-bits in size; if using a 32-bit RISC-V processor (RV32I), then each register is 32-bits in size.

If running an embedded version (RV32E), then there are only 15 general purpose 32-bit registers (**x1** to **x15**).

Some of the RISC-V extensions add additional registers, for instance, for floating point numbers.

RISC-V Instruction Format

Each RISC-V binary instruction is 32-bits long, and fitting all the information for an instruction into 32-bits is quite an accomplishment that requires using every bit to tell the processor what to do. There are several instruction formats, and it can be helpful to know how the bits for each instruction are packed into 32-bits.

Since there are 32 registers (the 31 general purpose registers plus the zero register), it takes five bits to specify a register. Thus, if three registers are needed, then 15 bits are taken up specifying these.

Having small fixed length instructions allows the RISC-V processor to load multiple instructions quickly. It doesn't need to start decoding an instruction to know how long it is and, therefore, where the next instruction starts. This is a key feature to allow processing parallelism and efficiency.

To give you an idea for register-only integer computational instructions, let's consider the format for a common class of instructions that we'll deal with early on. This is called an R-type instruction. Figure 2-3 shows the format of the instruction and what the bits specify.

31-25	24-20	19-15	14-12	11-7	6-0
funct7	rs2	rs1	funct3	rd	opcode

Figure 2-3. *R-type instruction format for register-only integer computational instructions*

Let's look at each of these fields:

- **funct7** and **funct3** combine to specify the operation such as addition or subtraction.
- **rs2** and **rs1** are the source registers.

- **rd** is the destination register.
- **opcode** specifies the type of instruction, in this case a register-only computation.

Creating Assembly Language code directly in hexadecimal is painful. Fortunately, there are tools to format the bits correctly for us.

About the GCC Assembler

Writing Assembler code in binary as 32-bit instructions would be painfully tedious. Enter GNU's Assembler which gives the power to specify everything that the RISC-V CPU can do but takes care of getting all the bits in the right place for you. The general way to specify assembly instructions is as follows:

```
label:     opcode    operands
```

The label: part is optional and only required if the instruction is to be the target of a **Branch** instruction.

There are quite a few opcodes; each one is a short mnemonic that is human readable and easy for the Assembler to process. They include:

- **add** for Addition
- **ld** for Load a Register
- **j** for Jump

There are several different formats for the operands and they will be described as the instructions that use are covered.

Anything after a hash sign "#" is a comment. Anything between a /* and a */ is also a comment. The /* */ comments can span multiple lines.

The register names must be in lowercase, so the convention for RISC-V Assembly Language is to have all instructions entirely in lowercase.

Knowing how numbers are represented, the RISC-V's registers, and the general format of Assembly Language instructions, let's look at specifics.

Adding Registers

The simplest arithmetic instruction adds two registers and places the result in a third register.

```
add  rd, rs1, rs2
```

This instruction adds the contents of rs2 and rs2, then places the result in register rd. For example,

```
add  x5, x6, x7
```

computes **x5** = **x6** + **x7**. If running 64-bits then this is a 64-bit addition, else if 32-bits then this is a 32-bit addition.

The source and destination register can be the same so:

```
add  x5, x5, x6
```

will add x6 to x5 computing: **x5** = **x5** + **x6**.

32-bits in a 64-bit World

When using a 64-bit RISC-V processor, a programmer may not want to always perform 64-bit computations and instead stick to performing 32-bit computations. Performing either a 32-bit or 64-bit addition requires one clock cycle, so computation time is not saved doing 32-bit arithmetic; however, memory bandwidth and the amount of memory used are saved. Often the bottleneck in CPU performance is how fast data can be moved in and out of main memory. Loading 32-bit quantities takes half the memory bandwidth of 64-bit quantities, and this can result in major performance gains. To add the lower 32-bits of two registers in a 64-bit processor, use the following:

```
addw rd, rs1, rs2
```

For example,

```
addw x7, x5, x6
```

adds the lower 32-bits of registers **x5** and **x6** placing the 32-bit result in **x7**.

A common use of addition is to move the contents of registers around.

Moving Registers

The RISC-V designers worked hard to minimize the number of instructions required, and the clever usage of existing instructions is always preferred to adding more instructions.

About Pseudoinstructions

Consider

```
add  x5, x0, x6
```

This instruction adds the contents of register **x6** to the zero register and puts the result in **x5**. This essentially moves **x6** to **x5**. Thus, we don't need an instruction:

```
mv   x5, x6
```

However, the GNU Assembler implements this as a pseudoinstructions which get translated to an **add** instruction at build time.

("mv x5, x6" actually translates to "addi x5, x6, 0" which will be described shortly).

Remember that with RISC-V instructions being only 32-bits, none can be wasted so the RISC-V designers were careful to avoid redundancy as it would have been a waste of valuable bits to have such a separate **mv** instruction.

Knowing all these tricks would make programs unreadable and put a lot of pressure on programmers to know all the clever tricks the RISC-V designers used to reduce the number of real instructions in the processor. The solution is to have the GNU Assembler know all these tricks and do the translations for the programmer.

In this book, pseudoinstructions are used to make the programs readable but point out when they're used to help understand what's going on.

Registers can now be added, but how are numbers loaded into a register?

About Immediate Values

There is a second form of the **add** instruction, as follows:

```
addi rd, rs, imm
```

where imm is a 12-bit immediate value which can have values between -2^{11} and $2^{11}-1$ or -2048 to 2047. **addi** then calculates rd = rs + imm. You may also use an **add** instruction in your code, and the GNU Assembler is smart enough to convert it to an **addi** instruction for you based on the operands provided, for example,

```
addi x7, x6, 15
add  x7, x6, 15   # The assembler will change this to
                  # addi for you.
```

calculates **x7** = **x6** + 15.

There is then a corresponding load immediate (**li**) pseudoinstruction to load an immediate value into a register:

```
li   rd, imm
```

For example,

```
li   x7, 15
```

To load 15 into register **x7**. This example will be translated into an **addi** instruction. These instructions are examples of RISC-V I-type instructions, whose format is shown in Figure 2-4.

31-20	19-15	14-12	11-7	6-0
imm	rs1	funct3	rd	opcode

Figure 2-4. *I-type format for immediate operations*

Having 32-bit instructions is common for RISC type CPUs, but this discussion shows a problem with how to load a 32-bit or 64-bit register. To do this takes multiple instructions. To start with, there is a special RISC-V instruction to load the upper twenty bits of a register.

Loading the Top

The load upper immediate (**lui**) instruction is used to build 32-bit constants. It allows the loading of 20-bits into the upper 20-bits of a 32-bit register. It loads the same bits in a 64-bit register, namely, bits 11 to 31 in the middle of the register. All other bits are set to zero. This instruction is an example of a U-type instruction shown in Figure 2-5.

31-12	11-7	6-0
imm	rd	opcode

Figure 2-5. *U-type instruction for a 20-bit immediate value*

The format of the lui instruction is

```
lui   rd, imm
```

where the immediate is 20-bits and will be loaded into the bits 11 to 31 of register rd.

If a 32-bit RISC-V CPU is running, then all 32-bits of a register can load in two instructions. For instance, to load 0x12345678 into register **x5**, use the following:

```
lui  x5, 0x12345
addi x5, x5, 0x678
```

lui loads the 20-bit quantity 0x12345 into the top 20 bits of register **x5** and then **addi** adds in the lower 12 bits.

For 64-bits, this is still handy, since many constants are 32-bits or less and using **lui** and **addi** lets these values load in only two instructions. To load all 64-bits of a 64-bit register, the left shift instruction will be introduced next.

Shifting the Bits

To complete loading a 64-bit register, the instruction we require is Shift Left Logical (**sll**). The shift operation will shift the bits in a register left by the number of indicated bits. Zero values come in from the right to take the place of the shifted bits. The number of bits can be specified either in a register (R-type instruction) or immediate (I-type instruction).

```
sll  rd, rs1, rs2
slli rd, rs, imm
```

sll shifts **rs1** left by the number of bits contained in **rs2**. **slli** shifts the bits in **rs** left by **imm** bits. Both place the results in **rd**.

Note Shifting a value left by one bit has the same effect as multiplying it by two. In fact, shifting a value left by n bits has the effect of multiplying the value by 2^n, for example,

```
slli x7, x5, 2
```

This calculation shifts the bits in register **x5** left two bits and stores the result in **x7**. This essentially multiplies **x5** by 4.

Now let's see how **slli** helps us load larger integers into registers.

Loading Larger Numbers into Registers

To load all the bits of a 64-bit register, a combination of **lui**, **addi**, and **slli** instructions are often necessary. Let's load a 64-bit register with the full 64-bit value: 0x1234567890ABCDEF and load it into register **x5**. To keep this example simple, use a temporary register x6 to load and position the various values. The code in Listing 2-1 presents this as a complete program. How to examine the results will be presented in Chapter 3, "Tooling Up," in the GNU Debugger (**gdb**) section.

Listing 2-1. Example code to load all 64-bits in a register

```
# Load 0x1234567890ABCDEF into register x5

.global _start      # Provide program starting address
                      to linker

_start: lui  x5, 0x12345   # x5 = 0x12345000
        slli x5, x5, 32    # x5 = 0x1234500000000000
        addi x6, x0, 0x679 # x6 = 0x679
        slli x6, x6, 32    # x6 = 0x67900000000
        add  x5, x5, x6    # x5 = 0x1234567900000000
        addi x6, x0, 0xFFFFFFFFFFFFF90B  # x6 = 0x90B
```

```
        slli x6, x6, 20    # x6 = 0x90B00000
        add  x5, x5, x6    # x5 = 0x1234567890B00000
        addi x6, x0, 0xFFFFFFFFFFFFFBCD  # x6 = 0xBCD
        slli x6, x6, 8     # x6 = 0xBCD00
        add  x5, x5, x6    # x5 = 0x1234567890ABCD00
        addi x5, x5, 0xEF  # x5 = 0x1234567890ABCDEF

# Setup the parameters to exit the program
# and then call Linux to do it.
        mv    a0, x0       # Use 0 return code
        li    a7, 93       # Service command code 93 terminates
        ecall              # Call linux to terminate
                             the program
```

Note Notice that comments are included on each line to help those reading the code understand what is going on. This is important in Assembly Language programming, since readers will have trouble understanding the bigger picture from such small instructions.

When providing the Assembler with a negative hex number as the immediate value, it has to be fully sign-extended to 64-bits or the GNU Assembler will give a warning.

If one of the immediate values is negative, then the preceding constant needs to be incremented by one. To understand why that occurs, see Exercise 2-7.

For large 64-bit quantities, this is a lot of work. Fortunately, the GNU Assembler **li** pseudoinstruction expands to multiple instructions depending on the value of imm.

```
        li   rd, imm
```

where rd is the target register to load and imm is either 32-bit or 64-bit integer, depending if a 32-bit or 64-bit RISC-V processor is running. The Assembler will translate this into a number of **lui**, **addi**, and **ssli** instructions to accomplish the task, for example,

```
li   x5, 0x1234567890ABCDEF
```

Let's have a quick look at the other two shift instructions before continuing.

More Shift Instructions

Since there is a left shift instruction, it should be no surprise that there is a right shift instruction. This comes in two variants, one is shift right logical (**srl**) where 0s come in from the left, and the other is shift right arithmetic (**sra**), where the sign bit is preserved, so negative numbers do not suddenly turn positive. There are both R-type and I-type versions of both these instructions:

```
srl  rd, rs   # shift right logical
srli rd, imm  # shift right logical immediate
sra  rd, rs   # shift right arithmetic
srai rd, imm  # shift right arithmetic immediate
```

As with **add**, if the **i** is left off, the Assembler can generate the correct code based on the operands.

To finish off the chapter, we have one more operation, the inverse of addition, that is, subtraction.

About Subtraction

The subtraction instruction is quite simple:

```
sub  xd, rs1, rs2
```

which calculates **xd** = **rs1** – **rs2**, for example,

```
sub  x5, x6, x7
```

calculates **x5** = **x6** - **x7**.

Note There is no **subi** instruction. Since the immediate in **addi** can be negative, there is no need for such an instruction.

Summary

This chapter covered how both positive and negative integer numbers are represented in a RISC-V CPU, along with introducing the CPU's register set where arithmetic operations take place. Both addition and subtraction were explained, along with how to load values into the CPU registers using Assembly Language instructions. The set of RISC-V shift operations were introduced as well as the Assembler's pseudoinstructions that make Assembly Language programming easier as well as gave some examples of using all these instructions.

In Chapter 3, "Tooling Up," better ways to build programs and start debugging programs with the GNU Debugger (**gdb**) are given.

Exercises

1. Compute the 8-bit two's complement for -79 and -23.
2. What are the negative decimal numbers represented by the bytes 0xF2 and 0x83?
3. Write out the bytes for 0x23 shifted left by three bits.
4. Write out the bytes for 0x4300 right shifted by five bits.
5. The hex value 0x00f30393 is the machine code for an **addi** instruction. Determine the two registers used along with the immediate value being added.
6. For the instruction,

   ```
   add x5, x6, x7
   ```

   construct the hex digits of the R-type machine code instruction given func7 = 0, func3 = 0, and the opcode = 0x33.
7. In Listing 2-1, one was added to a constant, if the following constant was negative. Examine a case with small numbers to see why. Imagine the immediate is four bits and the target is an 8-bit number, such as the byte 0x6E. Work through the following steps:

   a. Load 6 into an 8-bit register.

   b. Shift is left 4 bits to get 0x60

   c. Add the sign extended value 0xFE (remember the result can only be 8-bits).

   d. What is the result?

   e. If step a. is changed to load 7, then what is the result?

CHAPTER 3

Tooling Up

In this chapter, we will learn to automate building programs on Linux by using **GNU Make** for the Starfive Visionfive 2 and for the QEMU emulator. Also, by using **CMake** for Espressif ESP32-C3, that is included with the Espressif SDK. We will then learn to debug these programs with the **GNU Debugger** (**GDB**) on all three platforms.

GNU Make

Under RISC-V Linux, the programs were built using a simple shell script to run the **GNU Assembler** then the **Linux linker/loader**. Moving forward, a more sophisticated tool is needed to build programs. **GNU Make** is the standard Linux utility to do this, and it comes preinstalled on many versions of Linux. A **GNU Make** file contains the following questions:

1. Specify the rules on how to build one thing from another.
2. List the targets you want to build and the files they depend on.

Then

1. **GNU Make** examines the file date/times to determine what needs to be built.
2. **GNU Make** issues the commands to build the components.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_3

Let's build our HelloWorld program from Chapter 1, "Getting Started," using **make**. First, create a text file named **makefile** containing the code in *Listing 3-1*.

Listing 3-1. Simple makefile for HelloWorld

```
HelloWorld: HelloWorld.o
        ld -o HelloWorld HelloWorld.o

HelloWorld.o: HelloWorld.S
        as -o HelloWorld.o HelloWorld.S
```

Note The command make is particular, and the indented lines must start with a tab not spaces, or you will get an error.

To build our file, type

```
make
```

Rebuild a Project

If we already built the program, then this won't do anything, since make sees that the executable is older than the **.o** file and that the **.o** file is older than the **.S** file. To force a rebuild type,

```
make -B
```

Rather than specify each file separately along with the command to build it, define a build rule, for example, building a **.o** file from a **.S** file.

Rule for Building .S files

Listing 3-2 shows a more advanced version, where a rule for building a **.o** file from a **.S** file is defined. The dependency still needs to be specified; however, the rule no longer needs to be included. As command line parameters are added to the **as** command and become more sophisticated, there is now a centralized location in which to do this.

Listing 3-2. HelloWorld makefile with a rule

```
 %.o : %.S
as $< -o $@

HelloWorld: HelloWorld.o
        ld -o HelloWorld HelloWorld.o
```

Make now knows how to create a **.o** file from a **.S** file. Make was instructed to build **HelloWorld** from **HelloWorld.o** and make can look at its list of rules to figure out how to build **HelloWorld.o**. There are some strange symbols in this file that represent the following:

- **%.S** is like a wildcard meaning any **.S** file.
- **$<** is a symbol for the source file.
- **$@** is a symbol for the output file.

There's a lot of good documentation on **make**, so it is not necessary to go into a lot of detail here.

Define Variables

Listing 3-3 shows how to define variables. What follows are instructions for the centralization of the list of files we want to assemble.

Listing 3-3. Adding a variable to the HelloWorld makefile

```
OBJS = HelloWorld.o

%.o : %.S
        as $< -o $@

HelloWorld: $(OBJS)
        ld -o HelloWorld $(OBJS)
```

With this code, source files can be added by adding the new file to the **OBJS=** line and **make** takes care of the rest. This is an introduction to **GNU Make**—there is a lot more to this powerful tool. As the book progresses, new elements will be introduced to **makefiles** as needed. Next, we look at **CMake**. If you are not using a microcontroller like the ESP32-C3, feel free to skip ahead to the section on **gdb**.

Build with CMake

CMake is an open source, build automation tool that is cross-platform and compiler independent. The goal of using **CMake** in the Espressif SDK is to hide the messy details of using the various compiler toolchains on the host computer, whether it's Linux, Windows, or MacOS. With **CMake**, the project is built from the same **CMakeLists.txt** file, but the details of how to run the GNU Assembler do not need to be known. To fully cover, **CMake** requires a full book, so only what needs to be known for Assembly Language programming is covered.

CMake knows about the main C compilers and Assemblers, including building C and Assembly Language files using the GNU toolchain. The Espressif SDK adds **CMake** files to give specific options, such as which Espressif product is being used, and lets **CMake** know where all the SDK files are located. The goal is to specify the target executable name and list the files that need to be built, then **CMake**, with the help of some definition

files in the SDK, does all the work. **CMake** doesn't actually build a product, instead it creates a Makefile for the GNU Make tool which was covered in the previous section. GNU Make is then run from the command line to do the compiling. The Espressif SDK includes the Python program **idf.py** to automate this process.

The HelloWorld program in Chapter 1, "Getting Started," for the Espressif ESP32-C3 contains two **CMakelist.txt** files. The first is Listing 3-4, located in the main project folder.

Listing 3-4. The main CMakelist.txt file for HelloWorld

```
# The following lines of boilerplate have to be in your
project's
# CMakeLists in this exact order for cmake to work correctly
cmake_minimum_required(VERSION 3.16)

include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(hello_world)
```

Like our Assembly Language, anything after a hash mark (#) is a comment. The first statement, as follows, specifies the minimum version of **CMake** required to build the project:

```
cmake_minimum_required(VERSION 3.16)
```

This is the recommended value from the SDK and indicates the minimum version to build the SDK files. There is a lot of version checking of tools and libraries in **CMake** and the SDK to ensure projects are built reliably.

Next is the include statement, as follows:

```
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
```

The include statement incorporates the code from the specified file into the program file and executes it. To see the value of IDF_PATH under Windows, use the echo command, as follows:

```
echo %IDF_PATH%
C:\Espressif\frameworks\esp-idf-v5.0.2
```

This file then includes several more files that set up all the rules for building the SDK files, checks the versions of several tools, and finally applies all the configurable options to set things up for the processor. The last statement is as follows:

```
project(hello_world)
```

This statement defines our project name as hello_world and sets what the target executable will be called. If conventions for the structure of an Espressif SDK project are followed, then the SDK files will do most of the work for you. This is why we have the source file in a main subfolder. In the main subfolder is another **CMakelist.txt** file, shown in Listing 3-5, to specify the source code files.

Listing 3-5. The CMakelist.txt file in the main folder

```
idf_component_register(SRCS "hello_world_main.S"
                       INCLUDE_DIRS "")
```

Listing 3-5 contains only one line and is where to add source files.

Note Source files can be of different types, for example, an Assembly Language, a C and a C++ file. Based on the file extension, **CMake** creates the correct build rules into the generated **makefile**. Usually, as the project grows, all that is needed is to add files here and **CMake** takes care of the rest.

We are now more skilled at building projects; next we look at debugging our projects.

Debugging with GDB

Most high-level languages come with tools to easily output any strings or numbers to the console, a window, or a web page. Often when using these languages, programmers don't bother using the debugger, instead they rely upon libraries that are part of the language. Later, we will look at how to leverage the libraries that are part of other languages, but calling these takes a bit of work. We will also develop a helpful library to convert numbers to strings, to use the techniques used in HelloWorld in Chapter 1, "Getting Started," to print out work.

When programming with Assembly Language, being proficient with the debugger is critical to success. Not only will this help with Assembly Language programming, but also it is a great tool to use with high-level language programming.

We used **x0** to **x31** as the RISC-V integer register names. These names all work in **gdb**; however, **gdb** reports back the register names using their convention as to how they are used in function calls. This will be explained in detail in Chapter 6, "Functions and the Stack." Table 3-1 lists all the registers along with alternate names, or ABI names. In our HelloWorld program in Listing 1-1, the register names **a0** and **a7** were used when setting up the Linux function call to terminate the program. Table 3-1 shows which registers these are.

Table 3-1. *Alternate Register Names*

Register	ABI Name	Description
x0	zero	Hardwired zero register
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary register 0
x6	t1	Temporary register 1
x7	t2	Temporary register 2
x8	s0/fp	Saved register 0/frame pointer
x9	s1	Saved register 1
x10	a0	Function argument 0/return value 0
x11	a1	Function argument 1/return value 1
x12	a2	Function argument 2
x13	a3	Function argument 3
x14	a4	Function argument 4
x15	a5	Function argument 5
x16	a6	Function argument 6
x17	a7	Function argument 7
x18	s2	Saved register 2
x19	s3	Saved register 3
x20	s4	Saved register 4
x21	s5	Saved register 5

(*continued*)

Table 3-1. (*continued*)

Register	ABI Name	Description
x22	s6	Saved register 6
x23	s7	Saved register 7
x24	s8	Saved register 8
x25	s9	Saved register 9
x26	s10	Saved register 10
x27	s11	Saved register 11
x28	t3	Temporary register 3
x29	t4	Temporary register 4
x30	t5	Temporary register 5
x31	t6	Temporary register 6

Debugging a simple program that works on either 32-bit or 64-bit RISC-V CPUs will be demonstrated next. How to get **gdb** up and running on all three of our environments will also be described. After that all the commands and techniques are the same. The simple program, Listing 3-6, uses a number of the instructions learned in Chapter 2, "Loading and Adding." Save this file and name it **simple.S**.

Listing 3-6. A simple program for us to debug

```
# This is a simple Assembly Language program
# to practice using gdb with.

.global _start      # Provide program starting address
                      to linker

_start:
```

```
# load register x5 with 0x12345678
        lui  x5, 0x12345
        addi x5, x5, 0x678

        li   x6, 10      # load x6 = 10

        add  x7, x5, x6  # add 10 to x5
        sub  x8, x7, x6  # subtract  10 from x7

# Setup the parameters to exit the program
# and then call Linux to do it.

        mv      a0, x0      # Use 0 return code
        li      a7, 93      # Service command code 93
                              terminates
        ecall               # Call linux to terminate
                              the program
```

Listing 3-6 is the version for 64-bit Linux to convert the program to run on an ESP32-C3, see Exercise 3-1.

GDB comes pre-installed on most Linux distributions and is included in the Espressif SDK. However, it is missing from the Starfive distribution and needs to be installed via

```
sudo apt-get install gdb
```

Preparation to Debug

There are two cases to consider when preparing to debug a program:

1. Debugging on Linux such as on a Starfive Visionfive 2 or using the QEMU emulator
2. Using the ESP32-C3 microcontroller

How to set up each case will be covered; once **gdb** is running, we proceed for all three environments.

Setup for Linux

The GNU Debugger (**gdb**) can debug a program as it is, but this isn't the most convenient way to go. For instance, in the HelloWorld program, the label is **helloworld**. If the program is debugged as is, the debugger won't know anything about this label, since the Assembler changed it into an address in a .data section. There is a command line option for the GNU Assembler to include a table of all the source code labels and symbols, so they can be used in the debugger. This makes the program executable a bit larger.

Often a debug flag is set while the program is being developed the program, then the debug flag is removed before releasing the program. Unlike some high-level programming languages, the debug flag doesn't affect the machine code generated, so the program behaves the same in both debug and non-debug mode.

Do not leave the debug information in a program for release, because it makes the program executable larger, but more importantly it is a wealth of information for hackers to reverse engineer a program. There are several cases where hackers caused damages ranging from mischief to significant security breaches, and enormous financial and data losses because the program still had debugging information present.

To add debug information to a program, assemble it with the **-g** flag. In Listing 3-7, a debug flag is added to the **makefile** which compiles the simple program in Listing 3-6.

Listing 3-7. Makefile with a debug flag

```
OBJS = simple.o
ifdef  DEBUG
DEBUGFLGS = -g
else
DEBUGFLGS =
endif
```

```
%.o : %.S
        as $(DEBUGFLGS) $< -o $@

simple: $(OBJS)
        ld -o   simple $(OBJS)
```

This **makefile** sets the debug flag if the variable DEBUG is defined. Define it on the command line for **make** with

```
make DEBUG=1
```

Or from the command line define an environment variable with

```
export DEBUG=1
```

To clear the environment variable, enter

```
export DEBUG=
```

When switching between **DEBUG** and **non-DEBUG**, run **make** with the **-B** switch to build everything.

Tip Create shell scripts **buildd** and **buildr** to call **make** with and without **DEBUG** defined.

Start GDB

To start debugging the **simple** program, enter the following command:

```
gdb simple
```

This command yields the abbreviated output:

```
GNU gdb (Debian 8.3.1-1) 8.3.1
 Copyright (C) 2019 Free Software Foundation, Inc.
...
Reading symbols from simple...
 (gdb)
```

Set Up gdb for the ESP32-C3

The ESP32-C3 is typical of microcontrollers, in that **gdb** operates on the host computer and it uses a program called **OpenOCD** to communicate with and control the microcontroller. The Espressif SDK installs the required tools to do this. This section assumes a USB connection is used to do this. Please consult the Espressif SDK to ensure the following procedure has not changed. Using a USB connection is different than the microUSB port on the board, as it is a separate USB connection that is only used for debugging.

The problem is that the standard USB port requires the processor to continuously run to work and will stop the processor at breakpoints, which severs the USB connection. The alternate USB port has a support chip to keep it alive even when **gdb** pauses the RISC-V processor's execution. To use this alternate, USB port requires wire splicing and soldering. To prepare the USB port

1. Take a USB cable that has a USB-A connector at one end and cut it in half to expose the wires within.
2. Within the outer insulating later and grounding shield, there should be four wires: black, red, white, and green. Connect these wires to various pins on the ESP32-C3 dev board as indicated in Table 3-2.

Table 3-2. *Connecting the USB to the ESP32-C3 Dev Board*

ESP32-C3 Pin	USB Cable Color	USB Signal
GPIO18	White	D-
GPIO19	Green	D+
5V	Red	5V Power
GND	Black	Ground

Although the USB wires could be soldered directly to the pins on the ESP32-C3 dev board, it is preferable to connect them to wires with ends that can be inserted into a breadboard. Figure 3-1 shows the individual USB wires soldered to wires to connect to a breadboard.

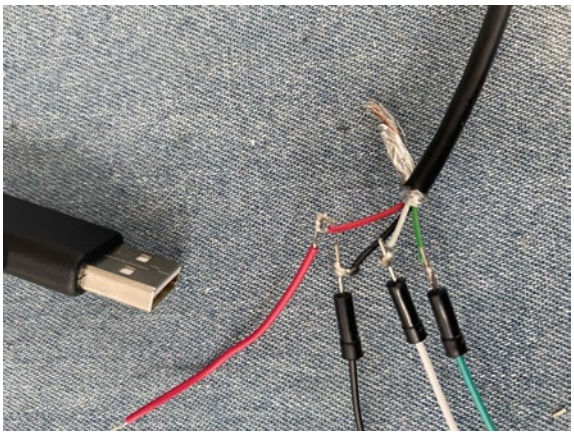

Figure 3-1. *Soldering the individual USB wires to breadboard connector wires*

3. With the wires soldered, the ESP32-C3 dev board and wires can be inserted into the breadboard as shown in Figure 3-2.

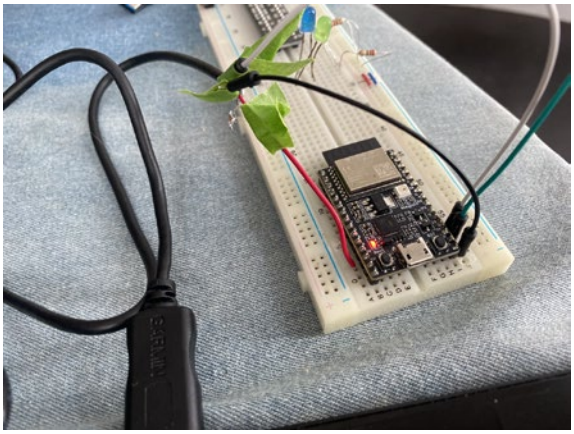

Figure 3-2. *The wires and ESP32-C3 dev board inserted in the breadboard*

4. Plug the USB cable into a computer, which powers on the ESP32-C3 and allows debugging to start.
5. Run **OpenOCD** with

   ```
   openocd -f board/esp32c3-builtin.cfg -c
   "program_esp build/simple.bin 0x10000 verify"
   ```

   This command downloads the simple program to the ESP32-C3 firmware and then starts listening for **gdb** commands.

Note These commands assume you are in your project folder in the command prompt setup by the Espressif SDK.

Note The program can be downloaded with the **idf.py** command, but then two USB connections going at once are needed. If using the microUSB for download, then leave off the -c parameter portion of the **openocd** command.

6. Create a file **gdbinit**, shown in Listing 3-8. This file includes setting a breakpoint at app_main, which saves a step in the next section.

Listing 3-8. Gdbinit file from Espressif SDK 5.1 documentation

```
target remote :3333
set remote hardware-watchpoint-limit 2
mon reset halt
maintenance flush register-cache
thb app_main
c
```

7. Run **gdb** with the command line:

```
riscv32-esp-elf-gdb -x gdbinit build/simple.elf
```

Now that **gdb** can start to debug the program in three environments, we will look at debugging the program.

Debugging with GDB

We will single step through the simple program with **gdb** running, examining the contents of the registers as it proceeds. The command prompt **(gdb)** is where commands are typed, then press the **tab** key for command completion. For a shortcut, enter the first letter or two of a command.

On Linux, to run the program, type:

```
run
```

(or **r**). On the ESP32-C3, use the continue command:

```
continue
```

(or **c**).

The program runs to completion, as if it ran normally from the command line. In the ESP32-C3 case, press Control + C to terminate the program.

To list our program, type

```
list
```

(or **l**).

This lists 10 lines. Type

```
l
```

for the next 10 lines. Type

```
list 1,1000
```

to list the entire program.

Notice that **list** gives the source code for the program, including comments. This is a handy way to find line numbers for other commands. To see the raw machine code, **gdb** can disassemble the program with, first for Linux:

```
disassemble _start
```

Then, for the ESP32-C3:

```
disassemble app_main
```

The **disassemble** command shows the actual code produced by the Assembler with no comments. The **mv** pseudoinstruction is translated, in this case into a **li** instruction.

To stop the program, set a breakpoint. In this case, stop the program at the beginning to single step through examining registers as we go. To set a breakpoint, use the **breakpoint** command (or **b**), first for Linux

```
b _start
```

Then for the ESP32-C3

```
b app_main
```

However, this is done automatically in **gdbinit**.

A line number, or a symbol, can be specified for the breakpoint, as in this example, now if the program runs, it stops at the breakpoint:

```
(gdb) b _start
Breakpoint 1 at 0x100b0: file simple.S, line 10.
(gdb) r
Starting program: /home/user/Chapter3/simple

Breakpoint 1, _start () at simple.S:10
10 lui x5, 0x12345
```

Step through the program with the **step** command (or **s**). See the values of the registers with **info registers** (or **i r**):

```
(gdb) s
11 addi x5, x5, 0x678

(gdb) i  r
ra             0x2aaaaee994 0x2aaaaee994
sp             0x3ffffff040 0x3ffffff040
gp             0x2aaab953f8 0x2aaab953f8
tp             0x3ff7e60780 0x3ff7e60780
```

```
t0              0x12345000 305418240
t1              0x3ff7ef7cc4 274742607044
...
t4              0x95cc4 613572
t5              0x2aaabb3 44739507
t6              0x3f0b27f 66105983
pc              0x100b4 0x100b4 <_start+4>
```

We see **0x12345000** put in **x5** (**t0**) as expected.

Continue stepping or enter **continue** (or **c**), to continue to the next **breakpoint** or to the end of the program. As many breakpoints can be set as desired. All breakpoints can be viewed with the **info breakpoints** (or **i b**) command. A breakpoint is deleted with the **delete** command, specifying the breakpoint number to delete.

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000100b0 simple.S:10
breakpoint already hit 1 time
(gdb) delete 1
(gdb) i b
No breakpoints or watchpoints.
```

Memory has not been addressed much yet, but **gdb** has good mechanisms to display memory in different formats. The main command is x, with following the format:

```
x /Nfu addr
```

where

- N is the number of objects to display.
- f is the display format where some common ones are
  - t for binary

    - x for hexadecimal
    - d for decimal
    - i for instruction
    - s for string
  - u is unit size and is any of
    - b for bytes
    - h for halfwords (16-bits)
    - w for words (32-bits)
    - g for giant words (64-bits)

Some examples using our code stored at memory location _start, or 0x100b0, are shown as follows:

```
(gdb) x /4ubft _start
0x100b0 <_start>: 10110111 01010010 00110100 00010010
(gdb) x /4ubfi _start
   0x100b0 <_start>: lui t0,0x12345
=> 0x100b4 <_start+4>: addi t0,t0,1656
   0x100b8 <_start+8>: li t1,10
   0x100bc <_start+12>: add t2,t0,t1
(gdb) x /4ubfx _start
0x100b0 <_start>: 0xb7 0x52 0x34 0x12
(gdb) x /4ubfd _start
0x100b0 <_start>: -73 82 52 18
```

To exit **gdb**, type **q** (for **quit** or type **control-d**).

Table 3-3 provides a quick reference to the GDB commands we introduced in this chapter. As we learn new things, we'll need to add to our knowledge of **gdb**. It is a powerful tool to help us develop our programs. Assembly Language programs are complex and subtle, and **gdb** is great at showing us what is going on with all the bits and bytes.

Table 3-3. *Summary of Useful GDB Commands*

Command (short form)	Description
break (b) line	Set breakpoint at line
run (r)	Run the program
step (s)	Single step program
continue (c)	Continue running the program
quit (q or control-d)	Exit gdb
control-c	Interrupt the running program
info registers (i r)	Print out the registers
info break	Print out the breakpoints
delete n	Delete breakpoint n
x /Nuf expression	Show contents of memory

It's worthwhile to single step through the sample programs and examine the registers at each step to ensure understanding of what each instruction is doing. Even if a bug is unknown to exist, many programmers like to single step through their code to look for problems and to convince themselves that their code is good. Often two programmers do this together as part of the pair programming agile methodology.

Summary

In this chapter, the **GNU Mak**e and **CMake** programs were introduced that will be used to build programs. These are powerful tools used to handle all the rules for the various compilers and linkers needed. Next, the GNU Debugger was introduced, that will allow the created programs to be troubleshot.

Instructions for debugging when a program is running on Linux, as well as debugging when a program is running on an ESP32-C3 microcontroller, were given using **GDB**. Unfortunately, programs have bugs and a method for single stepping through them and examining all the registers and memory is critical. **GDB** is a technical tool, but it's indispensable in figuring out what our programs are doing.

In Chapter 4, "Controlling Program Flow," conditionally executing code, branching, and looping, the core building blocks of programming logic will be presented.

Exercises

1. Create an Espressif SDK project for the program in Listing 3-6. One way to proceed is to clone one of the example projects like hello_world, then modify it to use **simple.S** as the source file. Modify **simple.S** to use **app_main** rather than **_start** as the entry point and global definition, then replace the Linux exit code with a "j app_main" instruction.

2. Setup and build **simple.S** for debug either on Linux or an ESP32-C3. Step through the program, examine the registers, and check that you understand the function of each instruction.

3. Create a **makefile** for the program in Listing 2-1. Include support for both release and debug builds.

4. Build the program from Listing 2-1 for debug and single step through for comprehension of the changes each instruction makes to the registers.

CHAPTER 4

Controlling Program Flow

With a handful of Assembly Language instructions given in previous chapters, they can be performed one after the other in a linear fashion. So far, in this book, programs have been built and debugged. Now, programs will become more interesting by using conditional logic—**if/then/else** statements, from high-level languages. Loops—**for** and **while** statements from high-level languages will also be introduced. With these additional instructions, all the basics for coding program logic will be complete.

Note Small code snippets will demonstrate the concepts to start. However, these snippets will not work on their own, but in the source code for this book. The **CodeSnippets.S** file puts them together in a program to run and step through in **gdb**.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_4

Creating Unconditional Jumps

There are two unconditional jump instructions: jump and link (**jal**), which takes a 20-bit immediate operand to jump to an offset from the current program counter (**pc**) and jump and link register (**jalr**), which jumps to the address stored in rs added to a 12-bit immediate operand.

```
jal  rd, label
jalr rd, rs, imm
```

The **rd** register has the value of **pc**+4 stored inside it when the instruction executes. This provides a return address for a subroutine to return to. Calling functions will be covered in detail in Chapter 6, "Functions and the Stack."

The **jalr** instruction is an I-type instruction and the **jal** instruction is a J-type instruction as shown in Figure 4-1.

31	30-21	20	19-Dec	11-07	6-0
imm[20]	imm[10:1]	imm[11]	imm[9-12]	rd	opcode

Figure 4-1. *Format of a J-type instruction such as* ***jal***

The rationale for breaking up the immediate value is to share some bits with the I-type instruction used for the **jalr** instruction; therefore, the sign bit is in the same place along with bits one to ten. This makes hand coding the instruction difficult, but the Assembler handles this complexity eliminating the need to hand code allowing some simplification in the electronic circuitry of a RISC-V CPU.

Note There is no zero-bit encoded. This means all jumps must be made to an even address. Since each instruction in RV32I and RV64I is 4-bytes and must be aligned on a word boundary, this isn't a restriction and allows a slightly greater jump range. The RV32E specification allows 16-bit instructions, which can all be reached via these instructions.

The range of the **jal** instruction is ±1 megabyte. The immediate is 20 bits and 2^{20} is 1 megabyte, since it does not include the low-order bit, it gives us the full range. To jump further away, load the address into a register and use the **jalr** instruction.

There are several pseudoinstructions that represent simple forms of the **jal** and **jalr** instructions. For instance, the simplest jump instruction is as follows:

```
j label
```

This sets **rd** to be the **zero** register since the return address is not needed. This instruction is like a **goto** statement in some high-level languages.

If Listing 4-1 is encoded, the program is in a closed loop and hangs the terminal window until **Control + C** is entered. This was seen in ESP32-C3 programs that run in an infinite loop.

Listing 4-1. A closed loop branch instruction

```
_start:   li x5, 1
               j _start
```

Infinite loops do not lead to interesting programs, so next some smarts will be added to programs with conditional branches to implement **if/then/else** logic.

Understanding Conditional Branches

The conditional branch instructions take two registers as operands, compare these registers, and branch if the comparison is true. The forms of the branch instructions are:

```
beq  rs1, rs2, label    # branch if equal
bne  rs1, rs2, label    # branch if not equal
blt  rs1, rs2, label    # branch if rs1 < rs2 (signed)
bge  rs1, rs2, label    # branch if rs1 >= rs2 (signed)
bltu rs1, rs2, label    # branch if rs1 < rs2
                          (unsigned)
bgeu rs2, rs2, label    # branch if rs1 >= rs2
                          (unsigned)
```

These six instructions are B-Type instructions specified in Figure 4-2.

31	30-25	24-20	19-15	14-12	11-8	7	6-0
imm[12]	imm[10-5]	rs2	rs1	funct3	imm[4-1]	imm[11]	opcode

Figure 4-2. *Format of a B-type instruction for the conditional branches*

Note Since the immediate is a multiple of 2 and 12 bits in size, the range of the branch is ±4096 bytes from the branch instruction.

Remember that memory addresses are either 32-bit or 64-bits depending on the processor used. In either case to jump to an arbitrary address, a **jalr** instruction is required. The other instructions are used to jump within functions or modules, which is sufficient for most programming constructs. This added complexity is a result of keeping RISC-V instructions at a fixed 32-bits in size, but the benefit is faster and simpler program execution.

Using Branch Pseudoinstructions

It might look like the list of instructions is incomplete, after all there is **blt,** but then no matching **bgt**. This is because the instruction is not necessary as the operands of **blt** can be reversed giving the **bgt**. Similarly, comparing to zero is common, so pseudoinstructions are provided for these. Following are the branch pseudoinstructions, with what they translate into in the comments:

```
beqz rs1, label     # beq rs, x0, label     # Branch if = zero
bnez rs1, label     # bne rs, x0, label     # Branch if ≠ zero
blez rs1, label     # bge x0, rs, label     # Branch if ≤ zero
bgez rs1, label     # bge rs, x0, label     # Branch if ≥ zero
bltz rs1, label     # blt rs, x0, label     # Branch if < zero
bgtz rs1, label     # blt x0, rs1, label    # Branch if > zero
bgt rs1, rs2, label # blt rs2, rs1, label   # Branch if >
ble rs1, rs2, label # bge rs2, rs1, label   # Branch if ≤
bgtu rs1, rs2, label # bltu rs2, rs1, label # Branch if >,
                                             unsigned
bleu rs1, rs2, label # bltu rs2, rs1, label # Branch if ≤,
                                             unsigned
```

This book uses these pseudoinstructions to keep the intent of the algorithm as clear as possible, as they appear frequently in the examples that follow.

Constructing Loops

With branch and comparison instructions available, let us look at constructing some loops modeled on what is found in high-level programming languages.

Create FOR Loops

To do the Basic For loop,

```
FOR I = 1 to 10
        ... some statements...
NEXT I
```

Implement this as shown in Listing 4-2.

Listing 4-2. Basic For Loop

```
        li   x5, 1        # x5 holds I
loop:   # The body of the loop goes here.
        # Most of the logic is at the end
        addi x5, x5, 1    # I = I + 1
        li   x6, 10       # end of loop for comparison
        ble  x5, x6, loop # IF I < 10 goto loop
```

It takes few instructions to implement a loop and the logic is usually simple.

Code While Loops

To code a While Loop such the following:

```
WHILE X < 5
        ... other statements ....
END WHILE
```

In Assembly Language, code as shown in Listing 4-3.

Listing 4-3. While Loop

```
# x5 is X and has been initialized
loop: li   x6, 5
      bge  x5, x6, loopdone
      # ... other statements in the loop body ...
      j    loop
loopdone: # program continues
```

Note Initializing the variables and changing the variables is not part of the **while** statement. These are separate statements that appear before and in the body of the loop.

A while loop only executes if the statement is initially true, so there is no guarantee that the loop body will ever be executed.

Coding If/Then/Else

In this section, we will code

```
IF <expression> THEN
        ... statements ...
ELSE
        ... statements ...
END IF
```

In Assembly Language, evaluate <expression> and have the result end up in a register that to compare. For now, assume that <expression> is simply of the form:

```
        register comparison immediate-constant
```

In this way, it can be evaluated with a single conditional branch instruction. For example, to implement the pseudo-code

```
IF x5 < 10 THEN
        .... if statements ...
ELSE
        ... else statements ...
END IF
```

We can code this as Listing 4-4.

Listing 4-4. If/then/else statement

```
        li   x6, 10
        bge  x5, x6, elseclause
        ... if statements ...

        j    endif
elseclause:

        ... else statements ...

endif:        # continue on after the /then/else ...
```

This is simple, but it is still worth putting in comments to be clear which statements are part of the if/then/else and which statements are in the body of the if or else blocks.

Tip Adding a blank line can make the code much more readable.

Manipulating Logical Operators

For the upcoming sample program, the bits in the registers need to be manipulated. The RISC-V logical operators provide several tools for us to do this, as follows:

```
        and  xd, xs1, xs2
        andi xd, xs, imm
        or   xd, xs1, xs2
        ori  xd, xs, imm
        xor  xd, xs1, xs2
        xori xd, xs, imm
```

These operate on each bit of the registers separately. They are either R-type or I-type instructions encoded similarly to **add** and **addi**.

Using AND

The instruction **and** performs a bitwise logical and operation between each bit in the two inputs, putting the result in **xd**. Remember that logical **AND** is true (1) if both arguments are true (1) and false (0) otherwise, for example.

In this example, use **AND** to mask off a byte of information. To store only the high order byte of a 32-bit quantity in register **x5**, see Listing 4-5.

Listing 4-5. Using and to mask a byte of information

```
# mask off the high order byte
        li   x6, 0xFF000000    # mask value to x6
        and  x5, x5, x6
# right shift the byte down to the low order position.
        srli x5, x5, 24
```

Using XOR

The **xor** instruction performs a bitwise exclusive or operation between each bit in the two operands, putting the result in **xd**. Remember that exclusive OR is true (1) if exactly one argument is true (1) and false (0) otherwise.

Using OR

The **or** instruction performs a bitwise logical or operation between each bit in the two operands, putting the result in **xd**. Remember that logical OR is true (1) if one or both arguments are true (1) and false (0) if both arguments are false (0), for example,

```
ori  x6, x6, 0xFF
```

This sets the low-order byte of **x6** to all 1 bits (0xFF) while leaving the other bytes unaffected.

Adopting Design Patterns

When writing Assembly Language code, there is a great temptation to be creative. For instance, we could do a loop ten times by setting the tenth bit in a register, then shifting it right until the register is zero. This works, but it makes reading a program difficult. If the program is left and returned to in a month, the programmer will be scratching their head as to what the program does.

Design patterns are typical solutions to common programming patterns. If a few standard design patterns are adopted for performing loops and other programming constructs, it will make reading programs much easier.

Design patterns make programming more productive since an example from a collection of tried-and-true patterns can be used for most situations.

Tip In Assembly Language, document which design pattern used, along with documenting the registers used.

Therefore, loops are implemented and **if**/**then**/**else** in the pattern of a high-level language. If this is done, it makes programs more reliable and quicker to write. In Chapter 6, "Functions and the Stack," how to use the macrofacility in the GNU Assembler is explained to help with this.

Converting Integers to ASCII

As a first example of a loop, convert a 64-bit register to ASCII to display the contents on the console. In the HelloWorld program in Chapter 1, "Getting Started," the Linux system call number 64 was used to output the "Hello World!" string. In this program, the hex digits in the register will be converted to ASCII characters, digit by digit. ASCII is one way that computers represent all the letters, numbers, and symbols as integers that a computer can process, for instance,

- **A** is represented by 65
- **B** by 66
- **0** by 48
- **1** by 49 and so on.

The key point is that the letters A to Z are contiguous as are the numbers 0 to 9. See Appendix D, "ASCII Character Set," for all 255 characters.

Note For a single ASCII character that fits in one byte, enclose it in single quotes, for example, 'A'. If the ASCII characters are going to comprise a string, use double quotes, for example, "Hello World!"

Listing 4-6 is some high-level language pseudo-code for what will be implemented in Assembly Language.

Listing 4-6. Pseudo-code to print a register

```
outstr = memory where we want the string + 17
// (string is form 0x123456789ABCDEF0 and we want
// the last character)
FOR x6 = 16 TO 1 STEP -1
        digit = x5 AND 0xf
        IF digit < 10 THEN
                asciichar = digit + '0'
        ELSE
                asciichar = digit + 'A' - 10
        END IF
        *outstr = asciichar
        digit = digit shifted right 4 bits
        outstr = outstr - 1
NEXT x6
```

Listing 4-7 is the Assembly language program to implement this. It uses what was learned about **loops**, **if/else**, and **logical** statements. The file should be **printdword.S**.

Listing 4-7 is coded for a 64-bit RISC-V CPU running Linux. Exercise 4-3 is to convert this to 32-bits to run on an ESP32-C3, plus both projects are included in the source code for this book.

Listing 4-7. Printing a register in ASCII.

```
#
# Assembler program to print a register in hex
# to stdout.
#
# a0-a2 (x10-x12) - parameters to linux function services
# a1 - is also address of byte we are writing
# x5 - register to print
# x6 - loop index
# x7 - current character
# a7 (x17) - linux function number
#

.global _start  # Provide program starting address

_start:
        # x5 contains the value to print.
        li   x5, 0x1234FEDC4F5D6E3A
        la   a1, hexstr   # start of string
        add  a1, a1, 17   # start at least sig digit
# The loop is FOR x6 = 16 TO 1 STEP -1
        li   x6, 16       # 16 digits to print
loop:   and  x7, x5, 0xf  # mask of least sig digit
# If x7 >= 10 then goto letter
        li   x28, 10      # is 0-9 or A-F
        bge  x7, x28, letter
# Else its a number so convert to an ASCII digit
        addi x7, x7, '0'
        j    cont         # goto to end if
letter: # handle the digits A to F
        addi x7, x7, ('A'-10)
```

```
cont: # end if
        sb   x7, 0(a1)    # store ascii digit
        addi a1, a1, -1   # decrement address for next digit
        srli x5, x5, 4    # shift off the digit

        # next W5
        addi x6, x6, -1   # step x6 by -1
        bnez x6, loop     # another for loop if not done

# Setup the parameters to print our hex number
# and then call Linux to do it.
li      a0, 1             # 1 = StdOut
        la   a1, hexstr   # string to print
        li   a2, 19       # length of our string
        li   a7, 64       # linux write system call
        ecall             # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.
        li   a0, 0       # Use 0 return code
        li   a7, 93      # Service code 93 terminates
        ecall            # Call linux to terminate

.data
hexstr:      .ascii  "0x123456789ABCDEFG\n"
```

If we compile and execute the program, the following would be seen as expected:

```
user@starfive:~/Chapter4$ make -B DEBUG=1
as -g printdword.S -o printdword.o
ld -o printdword printdword.o
user@starfive:~/Chapter4$ ./printdword
0x1234FEDC4F5D6E3A
user@starfive:~/Chapter4$
```

The best way to understand this program is to single step through it in **gdb**, watch how it is using the registers and updating memory.

Make sure it is understood why the following masks off the low-order digit:

```
and  x7, x5, 0xf  # mask of least sig digit
```

Since **and** requires both operands to be 1 in order to result in 1, and'ing something with 1's (like 0xf) keeps the other operator as is, whereas and'ing something with 0's always makes the result 0.

In our loop, we shift **x5**, 4 bits right with:

```
srli x5, x5, 4    # shift off the digit
```

This shifts the next digit into position for processing in the next iteration.

Note This is destructive to **x5** and the original number will be lost during this algorithm.

We've already discussed most of the elements present in this program, but there are a couple of new elements, they are:

Using Expressions in Immediate Constants

```
addi x7, x7, ('A'-10)
```

This demonstrates a couple of new tricks from the GNU Assembler.

1. Include ASCII characters in immediate operands by putting them in single quotes.
2. Place simple expressions in the immediate operands. Above the GNU Assembler translates 'A' to 65, subtracts 10 to get 55, and uses that as the immediate operand.

This makes the program more readable, the intent can be seen, rather than if 55 had just been coded here. There is no penalty to the program in doing this, since the work is done when the program was assembled, not when it was run.

Storing a Register to Memory

```
        sb x7, 0(a1)        # store ascii digit
```

The **Store Byte** (**sb**) instruction saves the low-order byte of the first register into the memory location contained in **a1**. The syntax **0(a1)** is how to specify a memory address that will have an immediate added to it, in this case zero.

Accessing data in memory is the topic of Chapter 5, "Thanks for the Memories," where far greater detail will be given.

Why Not Print in Decimal?

In this example program, converting to a hex string is easily done because using **andi** 0xf is equivalent to getting the remainder when dividing by 16. Similarly, shifting the register right four bits is equivalent to dividing by 16. To convert decimal, base ten, to string, requires getting the remainder of dividing by ten, and later dividing by ten.

So far, there has not been a divide instruction. This places converting to decimal beyond the scope of this chapter and it will be deferred until Chapter 10, "Multiply and Divide." Generally, the hex representation of registers is more useful to programmers anyway, and it can always be converted to any format desired with the Gnome Calculator.

Performance of Branch Instructions

RISC-V CPUs execute instructions in a pipeline. Individually, an instruction requires multiple clock cycles to execute, if the CPU implements a three-stage pipeline, then each instruction is executed in three steps, one for each of

1. Load the instruction from memory to the CPU
2. Decode the instruction
3. Execute the instruction

However, the CPU works on three instructions at once, each at a different step, so on average one instruction every clock cycle is executed. But what happens when we branch?

When a branch is executed, the next instruction is already decoded and loaded the instruction two ahead. When branching, this work is thrown away and starts over. This means that the instruction after the branch will take three clock cycles to execute. Newer RISC-V processors have more sophisticated, longer pipelines and can sometimes continue by guessing which branch will be taken, but ultimately you can overload these mechanisms and cause a pipeline stall.

If you put a lot of branches in your code, you suffer a performance penalty, perhaps slowing your program by a factor of three. Another problem is that if you program with a lot of branches, this leads to **spaghetti code**—meaning all the lines of code are tangled together like a pot of spaghetti, understandably quite hard to maintain.

When I first learned to program in high school and my undergraduate years before structured programming was available, I used the Basic and Fortran programming languages to write complex code. I know firsthand that deciphering programs full of branches is a challenge.

Early high-level programming languages relied on the **goto** statement, which led to hard-to-understand code. This led to the structured programming we see in modern high-level languages, that don't need a **goto** statement. We can't entirely do away with branches, since RISC-V doesn't have structured programming constructs, but we need to structure our code along these lines to make it both more efficient and easier to read. Another great use is for a few good design patterns.

Using Comparison Instructions

The conditional branch instructions compare two registers and then possibly branch based on the result. Suppose you want to compare two registers, but only want to know the result, rather than branching. There is a subset of the conditional branch instructions that set a destination register based on the result of the less than comparison:

```
slt   rd, rs1, rs2  # rd = 1 if rs1 < rs2, else rd = 0
slti  rd, rs, imm   # rd = 1 if rs < imm, else rd = 0
sltu  rd, rs1, rs2  # like slt, but for unsigned
                      integers
sltiu rd, rs, imm   # like slti, but for unsigned
                      integers
```

Then there are a set of pseudoinstructions for comparing to zero:

```
seqz  rd, rs    # set rd = 1 if rs = 0
snez  rd, rs    # set rd = 1 if rs ≠ 0
sltz  rd, rs    # set rd = 1 if rs < 0
sgtz  rd, rs    # set rd = 1 if rs > 0
```

Note At first glance, it seems strange that there could be equal and not equal to zero pseudoinstructions when the original set of instructions only has less than. The trick is to use unsigned integers, then equal to zero is the same as less than 1, so

```
seqz  x7, x5
```

Translates to

```
sltiu x7,x5,1
```

This is quite clever and is key to making the instruction set as small as possible, which results in a faster simpler CPU. It also demonstrates how pseudoinstructions greatly enhance code readability.

Summary

In this chapter, the key instructions for performing program logic with loops and **if** statements were studied. These included the instructions for unconditional and conditional branches. We discussed several design patterns to code the common constructs from high-level programming languages in Assembly Language. We looked at the statements for logically working with the bits in a register. We examined how we could output the contents of a register in hexadecimal format.

In Chapter 5, "Thanks for the Memories," the details of how to load data to and from memory will be given.

Exercises

1. Create an Assembly Language framework to implement a SELECT/CASE construct. The format is

```
SELECT number
        CASE 1:
                << statements if number is 1 >>
        CASE 2:
                << statements if number is 2>>
        CASE ELSE:
                << statements if not any other case >>
END SELECT
```

2. Construct a DO/WHILE statement in Assembly language. In this case, the loop always executes once before the condition is tested:

```
DO
        << statements in the loop >>
UNTIL condition
```

3. Modify the earlier-mentioned printdword program to print the hex representation of a 32-bit register for the ESP32-C3.

4. The utility objdump will show what is in an object or executable file. It can be used to disassemble programs. Compile the CodeSnippets program from the Chapter 4, “Controlling Program Flow,” source code. Run the command line:

```
objdump -d CodeSnippets
```

which prints out the Assembly Language using pseudoinstructions and ABI register names. To get the raw code, use the following command:

```
objdump -d -M no-aliases,numeric CodeSnippets
```

Play with these on the sample executables to see the differences.

5. Code an IF/THEN/ELSE loop where the code in the THEN clause is greater than 4096 bytes in length. In this case, a conditional branch instruction cannot reach the ELSE clause. How do you branch around the THEN block of code?

CHAPTER 5

Thanks for the Memories

In this chapter, we put data to and from RISC-V-based computer's memory. So far, we've used memory to hold Assembly Language instructions, now we will learn how to define data in memory, to load memory into registers for processing, and to write the results back to memory.

The RISC-V processor uses a **load-store architecture**. This means that the instruction set is divided into two categories: one to load and store values from and to memory and the other to perform arithmetic and logical operations between the registers. Most operations have been arithmetical and logical up to this point, and the final operation is **load-store**, another category.

Memory addresses are either 32-bits or 64-bits depending on the CPU, while instructions are 32-bits, creating the same problems experienced in Chapter 2 "Loading and Adding," where tricks were required to load 32-bits or 64-bits into a register using 32-bit instructions. In this chapter, these same tricks are used for loading addresses, along with a few new ones. The goal is to load a memory address in one instruction in as many cases as possible.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_5

Defining Memory Contents

Before loading and storing memory, memory to operate on must be defined first. The GNU Assembler contains several directives to help define memory to use in a program. Some examples are given and summarized in Table 5-1. Listing 5-1 shows how to define bytes, words, 64-bit integers, and ASCII strings.

Listing 5-1. Some sample memory directives

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
        .word 0x1234ABCD, -1434
        .quad 0x123456789ABCDEF0
        .ascii      "Hello World\n"
```

The first line defines seven bytes all with the same value. We can define bytes in decimal, octal (base 8), binary, hex, or ASCII. Anywhere numbers are defined; expressions that the GNU Assembler uses will evaluate when it compiles our program.

Most memory directives start with a label, so they can be accessed from the code. The only exception is if a larger array of numbers is being defined that extend over several lines.

The .byte statement defines one or more bytes of memory. Listing 5-1 shows the various formats that can be used for the contents of each byte, as follows:

- A decimal integer starts with a non-zero digit and contains decimal digits 0–9.
- An octal integer starts with zero and contains octal digits 0–7.
- A binary integer starts with 0b or 0B and contains binary digits 0–1.

- A hex integer starts with 0x or 0X and contains hex digit 0-F.
- A floating-point number starts with 0f or 0e followed by a floating-point number.

Note Be careful not to start decimal numbers with zero (0), since this indicates the constant is an octal (base 8) number.

The example shows how to define a word, a quad (64-bit integer), and an ASCII string, as seen in the HelloWorld program in Chapter 1, "Getting Started." There are two prefix operators that can be placed in front of an integer:

- Negative (-) will take the two's complement of the integer.
- Complement (~) will take the one's complement of the integer.

For example,

```
.byte -0x45, -33, ~0b00111001
```

Table 5-1 lists the various data types we can define this way.

Table 5-1. *The List of Memory Definition Assembler Directives*

Directive	Description
.ascii	A string contained in double quotes
.asciz	A zero-byte terminated ascii string
.byte	1-byte integers
.double	Double precision floating point values
.float	Floating point values
.octa	16-byte integers
.quad	8-byte integers
.short	2-byte integers
.word	4-byte integers

To define a larger set of memory, there are a couple of mechanisms to do this without having to list and count them all, such as the macro

```
.fill repeat, size, value
```

This repeats a value of a given size, repeat times, for example, the macro

```
zeros:      .fill 10, 4, 0
```

creates a block of memory with 10 4-byte words all with a value of zero. The following code

```
.rept count
...
.endr
```

repeats the statements between **.rept** and **.endr**, count times. This can surround any code in your Assembly Language, for instance, you can make a loop by repeating your code count times, for example,

```
rpn:  .rept 3
        .byte 0, 1, 2
        .endr
```

is translated to

```
.byte 0, 1, 2
.byte 0, 1, 2
.byte 0, 1, 2
```

In ASCII strings, the special character "**n**" is used for a new line. There are a few more for common unprintable characters as well as to give the ability to put double quotes in strings. The "\" is called an escape character, which is a metacharacter to define special cases. Table 5-2 lists the escape character sequences supported by the GNU Assembler.

Table 5-2. *ASCII Escape Character Sequence Codes*

Escape Character Sequence	Description
\b	Backspace (ASCII code 8)
\f	Form feed (ASCII code 12)
\n	New line (ASCII code 10)
\r	Return (ASCII code 13)
\t	Tab (ASCII code 9)
\ddd	An Octal ASCII code (ex \123)
\xdd	A Hex ASCII code (ex \x4F)
****	The "\" character
\"	The double quote character
\anything-else	Anything-else

Aligning Data

These data directives put the data in memory continuously byte by byte. However, the RISC-V processor often requires data to be aligned on word boundaries, or some other measure. We can instruct the Assembler to align the next piece of data with an .align directive. For instance, consider

```
.data
        . byte     0x3F
        .align     4
        .word      0x12345678
```

The first byte is word aligned, but because it is only one byte, the next word of data will not be aligned. If it needs to be word aligned, then add the ".align 4" directive to make it word aligned. This results in three wasted bytes, but with gigabytes of memory, this shouldn't be too much of a worry.

Note Aligned data loads faster on most CPUs as the memory bus only loads aligned data, so loading non-aligned data takes two memory accesses.

To align everything to the end of the current section, the .balign statement accomplishes this.

RISC-V Assembly Language instructions must be word aligned, so if data is inserted in the middle of some instructions, then an .align directive is required before the instructions continue, or the program will crash when run. Usually the Assembler will give you an error when alignment is required and throwing in an ".align 4" directive is a quick fix.

Next, we discuss where these data definitions appear in program source code files.

About Program Sections

Assembly Language programs are broken up into sections. The default section is .text which is where the Assembly Language instructions appear. This is the default section, so it typically isn't explicitly specified in source code files. There is a .data section in Hello World program in Chapter 1, "Getting Started," but this is yet to be discussed.

The data definitions in this chapter can be placed in either of these program sections; however, any data placed in the **.text** section is read-only, and on a processor that supports memory protection, a runtime fault will occur if it is written to. The .data section, on the other hand, is read-write and can be read or written to freely. Listing 5-2 shows the structure of a typical source code file with both instructions and data.

Listing 5-2. Sections structure of a typical source code file

```
.text # default section, so this directive is usually left out
# instructions and read-only data
        j     label
.quad   0x123456
Label:  addi  x5, x5, 1
.data
# memory that can be read or written to.
        .word  0x1234
```

The reason the .text section is read-only is to prevent viruses or other malware modifying a program that is running in memory. Self-modifying programs can be written, but then the code must be placed in a read-write section and great care must be taken to maintain the integrity of the program.

Placing data in the .text section is useful for read-only data, such as constants or memory addresses to be accessed with a 12-bit immediate, so the data must be within 2048 bytes of the referencing instruction.

Now that data can be added to programs, let's look at how it is actually stored in memory.

Big vs. Little Endian

Data stored in memory has the bytes stored in the reverse order to what may be expected. In fact, a 32-bit representation of 1 stored in memory is

01 00 00 00

Rather than

00 00 00 01

Most processors pick one format, or the other to store numbers. Motorola and IBM mainframes use what is called Big Endian, where numbers are stored in the order of most significant digit to least significant digit, in this case

00 00 00 01

Intel, ARM, and RISC-V processors use Little Endian format and stores the numbers in reverse order with the least significant digit first, namely,

01 00 00 00

Figure 5-1 shows how the bytes in integers are copied into memory in both Little and Big Endian formats. Notice how the bytes end up in reverse order to each other.

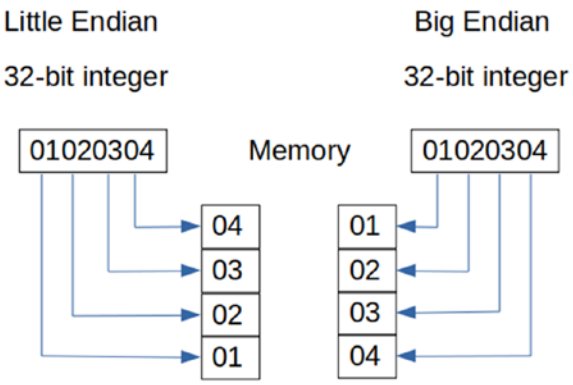

Figure 5-1. *How integers are stored in memory in Little vs. Big Endian format*

Pros of Little Endian

The advantage of Little Endian format is that it makes it easy to change the size of integers, without requiring any address arithmetic. To convert a 4-byte integer to a 1-byte integer, take the first byte. Assuming the integer is in the range of 0–255 and the other three bytes are zero. For example, if memory contains the 4 byte or word for 1, in Little Endian, the memory contains

01 00 00 00

If the 1-byte representation of this number is wanted, take the first byte, or for the 16-bit representation, take the first two bytes. The key point is that the memory address used is the same in all cases, saving an instruction cycle from adjusting it.

In the debugger, there are more representations, and these will be pointed out again as they occur.

Note Even though RISC-V uses Little Endian, many protocols like TCP/IP used on the Internet use Big Endian and so require a transformation when moving data from the computer to the outside world.

About Memory Addresses

Computer memory typically starts at address zero and then increments up to the size of installed memory. This address space might include several types of memory, including Read-Only Memory (ROM) and Random Access Memory (RAM). Each bank or type of memory will have a range of memory addresses assigned to it. Memory addresses are 32-bits on a 32-bit RISC-V CPU and 64-bits on a 64-bit CPU, the same size as the CPU registers.

For a microcontroller like the ESP32-C3, this is also how the program sees the memory; however, for the more sophisticated 64-bit CPUs running Linux, there are a couple of extra complexities. Linux supports running multiple processes at the same time, called multi-tasking. In this scenario, each process thinks it has complete access to the address space; however, this isn't the case.

Linux, with support from the RISC-V CPU, creates an address space for each process that contains various blocks of real memory that may not be contiguous. As a further complexity, Linux supports virtual memory, where if the operating system runs out of real RAM, it can save some parts to hard drive storage and give the appearance that there is much more memory than physically installed.

The good news is that each program is blissfully unaware of this complexity and can run as if it owns all the memory. Linux also controls access to memory based on security, so not all memory may

be accessed freely. Most programs don't need to worry about any of this, but if you are working on the Linux kernel or your own operating system, then you may need to know the internal details of all these mechanisms.

To access memory, there are two steps:

1. Getting the memory address into a CPU register
2. Using that CPU register to load or store data

The rest of this chapter deals with solving these two problems. The RISC-V instruction set is unique in that it often allows a blurring of these two steps, saving an instruction and providing a performance gain.

Loading a Register with an Address

In this section, we will look at techniques to load a memory address into a register. This is a similar problem to what was encountered in Chapter 2, "Loading and Adding," where we have 32-bit instructions but need to load either a 32-bit or 64-bit integer into a register. There is an extra complexity that under Linux, memory addresses must be adjusted by the Linux program loader when a program is run, which means we have to follow some standard patterns, or the loader will not be able to do its job. Loading memory addresses is a common operation so it should be done in as few instructions as possible, as a result the RISC-V instruction set provides some tricks to allow this operation to be performed in one or two instructions in most situations.

It is important to understand how memory addresses are handled, rather than relying entirely on the helpful pseudoinstructions which hide much of the detail. When single stepping in the debugger, the underlying instructions will be encountered rather than the pseudoinstructions.

As an Assembly Language programmer, it is important to know which techniques generate the fewest instructions, to encode the fastest code. To structure your program and data to load addresses in one instruction is a good performance boost.

The main techniques to access memory quickly are based on relative addressing, where an offset is provided to another register that points to a known point in memory. First using addresses relative to the program counter (**pc**) will be presented.

PC Relative Addressing

In Chapter 1, "Getting Started," we introduced the **la** pseudoinstruction to load the address of the "Hello World!" string. This is an easy way to load addresses but can hide a lot of details. A key instruction that is often generated by the **la** pseudoinstruction is the Add Upper Immediate to Program Counter (**auipc**) instruction:

```
auipc        rd, imm
```

auipc is a U-type instruction similar to the **lui** instruction. **auipc** takes the current **pc**, adds the 20-bit immediate, to bits 12 to 31 of the **pc,** and sets the lowest 12-bits to zero. If this appears confusing, Figure 5-2 demonstrates a 64-bit example to make this discussion concrete.

pc = 0x1234567812345678
auipc x5, 0x65432

bits	64-32	31-12	11-0
pc	0x12345678	0x12345	0x678
imm		0x65432	
x5	0x12345678	0x77777	0x000

Figure 5-2. *Given a specific pc, the table shows how the bits of the destination register are set*

Like the **lui** instruction, **auipc** must be paired with an I-type instruction to achieve the complete result. The next question is how do we know what the value of the **pc** is, so we can calculate the immediate values to add to it? The GNU Assembler provides some built-in functionality to help with this. To work properly, this design pattern must be followed exactly or a cryptic error from either the Assembler or the linker will be sent. The two built-in functions are **%pcrel_hi** and **%pcrel_lo**. Listing 5-3 shows how they are used.

Listing 5-3. Example of using pc-relative addressing

```
label:
        auipc x5, %pcrel_hi(msg)
        addi  x5, x5, %pcrel_lo(label)
        # x5 now contains the address of msg.
...
.data
msg: .asciz  "This is a message."
```

Note These instructions must be used as a pair in adjacent instructions, or an error will appear.

The label must be before the first instruction. **%pcrel_hi** takes the label of the address of what is wanted, and **%pcrel_lo** takes the label in the code. This is necessary since the loader may need to fix up the offset when the program is loaded. The data and instruction parts of the program could be given any arbitrary addresses when loaded. Relocation information is stored in the object file to assist the loader. Fortunately, this is handled by the tools if they are allowed to help.

To make things even easier, there is the load address (**la**) pseudoinstruction:

```
la   rd, label
```

This instruction will translate into a combination of **auipc** and **addi** instructions. For instance,

```
la   x5, msg
```

There are more tricks to loading addresses, but first use the address to load data.

Loading Data from Memory

In the HelloWorld program, only the address was needed to pass on to Linux, then it was used to print the string. Generally, these addresses are used to load data into a register.

There are several variations on the load instruction to load data given an address already loaded into a register, depending on the type of data being loaded:

```
l{type}   xd, imm(xa)
```

where type is one of the types listed in Table 5-3.

Table 5-3. *The Data Types for the Load/Store Instructions*

Type	Meaning
b	Signed byte (8-bits)
bu	Unsigned byte (8-bits)
h	Signed halfword (16-bits)
hu	Unsigned halfword (16-bits)
w	Signed word (32-bits)
wu	Unsigned word (32-bits)
d	Double word (64-bits), RV64I only

The signed versions will extend the sign across the rest of the register when data is loaded.

xd is the register to load the data into, **xa** is the register containing the memory address to load from, and imm is a 12-bit immediate to add to the memory address. These load instructions are all I-type instructions.

The 12-bit immediate can serve several useful purposes. Listing 5-4 shows the typical usage where we load an address into a register and then use that address to load the data we want.

Listing 5-4. Loading an address and then the value

```
# load the address of mynumbers into x5
        la    x5, mynumbers
# load the words stored at mynumbers into x28, x29, x30
        lw    x28, 0(x5)
        lw    x29, 4(x5)
        lw    x30, 8(x5)

.data
mynumbers:    .word 0x12345678, 0x9ABCDEF0, 0x1234
```

When single stepping in the debugger, the numbers loading into **x28**, **x29**, and **x30** can be seen.

Note The bracket syntax represents indirect memory access. This means loading the data stored at the address pointed to by **x5**, not move the contents of **x5** into **x28**.

This is one use case, but a more common use of the 12-bit immediate is to combine it with the 20-bits immediate from the **auipc** instruction.

Combining Loading Addresses and Memory

Previously a 12-bit immediate was used from an **addi** instruction to finish creating the complete memory address. However, the 12-bit immediate can be used from the load instruction. Listing 5-5 shows how to pair an **auipc** instruction with a **lw** instruction to load 32-bits of data from memory into a register.

Listing 5-5. Example of **`auipc`** and **`lw`** working together

```
label:
        auipc x6, %pcrel_hi(mynumber)
        lw    x6, %pcrel_lo(label)(x6)
        # x6 now contains 0x12345678.
...
.data
mynumber: .word    0x12345678
```

This way we are loading a value into a register in only two instructions. This is quite common, so there are pseudoinstructions that will perform this pairing for you. If you specify a label rather than an address register, then it is a pseudoinstruction that expands into instructions like those in Listing 5-5. For instance,

```
lw    x6, msg
```

Now that we have loading in hand, let's quickly look at storing.

Storing a Register

The Store Register **s{type}** instruction is a mirror of the **l{type}** instruction. We've seen the **sb** instruction a couple of times already in our examples. Store is an S-type instruction, where the only difference to an I-type is how the immediate is encoded in the instruction. For example,

```
label:
        auipc x6, %pcrel_hi(mynumber)
        sw    x7, %pcrel_lo(label)(x6)
```

Note The same register cannot be used for the address and value, since otherwise **auipc** will overwrite the value to be saved.

There are no unsigned versions of the store instruction; since it writes the exact length, sign extension is never needed.

There are matching pseudoinstructions to generate the **auipc**/store combinations automatically; however, for store, a temporary register must be provided, so that **auipc** doesn't overwrite the value being saved. For instance,

```
Sw x5, msg, x6 # x5 is being saved, x6 is a temp register
```

Since these pseudoinstructions expand to two instructions, it can make sense to use load address at the beginning, rather than using the combinations such as

```
        la    x5, wordvalue   # address of our variable in two
                                instructions
...
        lw    x6, 0(x5)       # load value in one instruction
...
        sw    x6, 0(x5)       # save value in one instruction
```

Loading an address takes two instructions, but can that be reduced to one instruction?

Optimizing Through Relaxing

The code has been using the **pc** as a base to create addresses; however, can a different register be used to get even tighter code? Consider the following code in Listing 5-6:

Listing 5-6. Example of using a register to optimize loading addresses

```
# Initialize x3 as a pointer to the start of the data section.
label1:      auipc x3, %pcrel_hi(x3init)
             addi  x3, x3, %pcrel_lo(label1)
# ... more code
# Now load the address of msg, perhaps in a loop body
             addi  x5, x3, 14 # msg2-x3init
# x5 now contains the address of msg2, obtained in one
instruction.
.data
```

```
x3init:
msg1:  .asciz "First message\n"
msg2:  .asciz "This is a message\n"
```

Typically, the label to initialize the address register should be placed in the middle of the items that need to be pointed to. 12-bit immediates reference ±2048, so make use of the negative side as well as the positive side.

Note For 32-bit addresses, these techniques can access any memory address. However, for 64-bit addresses, this is only building a 32-bit offset from the **pc**. If handling huge amounts of memory, then having base registers other than the **pc** is critical to access data that is far removed from the current value of the **pc**.

The GNU call chain will perform this optimization automatically; however, as an Assembly Language programmer, controlling this explicitly may be better than letting the toolchain do it. For that to work requires some initialization that is contained in the initialization code for the C runtime, but if the C runtime is not used, this won't be setup. This optimization is called address relaxation and is performed in the linker. To turn off this optimization includes the following flag in the arguments passed to **as**:

```
-mno-relax
```

The linker creates a label **__global_pointer$** which points to 2048 bytes into the data segment, then the linker assumes that register **x3**, also called **gp**, points to this address. Listing 5-7 shows how to set this up and allow this optimization to work.

Listing 5-7. Demonstration that allows automatic address relaxation

```
_start:
.option push
.option norelax
1:      auipc gp, %pcrel_hi(__global_pointer$)
        addi  gp, gp, %pcrel_lo(1b)
.option pop
# ...
1:      auipc x5, %pcrel_hi(msg)
        addi  x5, x5, %pcrel_lo(1b)
.data
.msg    .asciz "This is a message"
```

A few notes about this code fragment:

- The second **auipc**/**addi** pair that loads the address of `msg` will be converted to a single statement:

  ```
  addi  x5, gp, (__global_pointer$-msg)
  ```

- Labels that are numbers can be repeated and the notation 1b means the first previous occurrence of the label 1. This is explained in more detail in Chapter 6, "Functions and the Stack."

- With **gp** initialized at the beginning of the program, **-mno-relax** is not needed, but beware that the linker will modify the code and that can lead to surprises, or confusion when debugging.

- The commands **.option pu**sh and **.option pop** are Assembler directives that push and pop the state of the Assembler's settings, allowing them to be temporarily changed.

- To turn off the relaxation optimization temporarily, the command **.option no-relax** is used; since if it is on at this point, it optimizes the setting of **gp** away.

In the following programs, the **-mno-relax** flag will be set. Now let's see an example.

Converting to Uppercase

As an example for putting many of the instructions learned into action, consider looping through a string of ASCII bytes. To convert any lowercase characters to uppercase, Listing 5-8 gives pseudo-code for how to do this.

Listing 5-8. Pseudo-code to convert a string to uppercase

```
i = 0
DO
        char = inStr[i]
        IF char >= 'a' AND char <= 'z' THEN
              char = char - ('a' - 'A')
        END IF
        outStr[i] = char
        i = i + 1
UNTIL char == 0
PRINT outStr
```

In this example, **NULL** terminated strings are used. These are common in C programming. Here instead of a string being a length and a sequence of characters, the string is the sequence of characters, followed by a **NULL** (ASCII code 0 or \0) character. To process the string, simply loop until encountering the **NULL** character.

While the **FOR** and **WHILE** loops were already covered, the third common structured programming loop is the **DO/UNTIL** loop that puts the condition at the end of the loop, see Exercise 4-2.

In this construct, the loop is always executed once. In this case, this is desired, since if the string is empty, but still copying the **NULL** character is still required, the output string will then be empty as well.

This program does not overwrite the input string, instead leaves the input string alone and produces a new output string with the uppercase version of the input string.

As is common in Assembly Language programming, the logic is reversed to jump around the code in the **IF** block. Listing 5-9 shows the updated pseudo-code.

Listing 5-9. Pseudo-code for implementing the IF statement

```
        IF char < 'a' GOTO continue
        IF char > 'z' GOTO continue
        char = char - ('a' - 'A')
continue: // the rest of the program
```

The structured programming constructs of a high-level language are not available to assist, and this turns out to be quite efficient in Assembly Language.

Listing 5-10 is the Assembly code to convert a string to uppercase. Save this code as upper.S, and when compiling, use the **as** command line option **-mno-relax**.

Listing 5-10. Program to convert a string to uppercase

```
#
# Assembler program to convert a string to
# all upper case.
#
```

```
# a0-a2 - parameters to Linux function services
# x5 - address of output string
# x6 - address of input string
# x7 - current character being processed
# a7 - Linux function number
#

.global _start # Provide program starting address to linker

_start:
        la    x5, outstr    # address of output string
        la    x6, instr     # start of input string

# The loop is until null (zero) character is encountered.
loop:   lb    x7, 0(x6)     # load character
        addi  x6, x6, 1     # increment buffer pointer
# If x7 > 'z' then goto cont
        li    x28, 'z'      # load 'z' for comparison
        bgt   x7, x28, cont # branch if letter > 'z'?
# Else if x7 < 'a' then goto end if
        li    x28, 'a'      # load 'a' for comparison
        blt   x7, x28, cont # goto to end if not lowercase
# if we got here then the letter is lower case, so convert it.
        addi  x7, x7, ('A'-'a')
cont:   # end if
        sb    x7, 0(x5)     # store character to output str
        addi  x5, x5, 1     # increment buffer for next char
        li    x28, 0        # load 0 char for comparison
        bne   x7, x28, loop # loop if character isn't null

# Setup the parameters to print our hex number
# and then call Linux to do it.
        li    a0, 1         # 1 = StdOut
```

```
        la    a1, outstr    # string to print
        sub   a2, x5, a1    # get the len by sub'ing the
                              pointers
        li    a7, 64        # Linux write system call
        ecall               # Call Linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.
        li    a0, 0         # Use 0 return code
        li    a7, 93        # Service code 93 terminates
        ecall               # Call Linux to terminate
                              the program

.data
instr:  .asciz  "This is our Test String that we will
convert.\n"
outstr:    .fill  255, 1, 0
```

Compile and run the program, to get the desired output:

```
user@starfive:~/Chapter5$ make
as -mno-relax upper.S -o upper.o
ld -o upper upper.o
user@starfive:~/Chapter5$ ./upper
THIS IS OUR TEST STRING THAT WE WILL CONVERT.
user@starfive:~/Chapter5$
```

This program is quite short. Besides all the comments and the code to print the string and exit, there are only 15 Assembly Language instructions to initialize and execute the loop:

- **Four instructions**: Initialize our pointers for **instr** and **outstr**. There are two pseudoinstructions that expand to two instructions each.
- **Four instructions**: Make up the if statement.
- **Seven instructions**: For the loop, including loading a character, saving a character, updating both pointers, checking for a null character, and branching if not null.

In this example, we use the **lb** and **sb** instructions, since we are processing byte by byte.

To convert the letter to uppercase, we use

```
addi  x7, x7, ('A'-'a')
```

The lowercase characters have higher values than the uppercase characters, so we just use an expression that the Assembler will evaluate to get the correct number to subtract. Since there is no **subi** instruction, the code uses the **addi** instruction with a negative immediate.

When we come to print the string, we don't know its length and Linux requires the length. We use the instruction

```
sub   a2, x5, a1   # get the len by subbing the pointers
```

Here **a1** was loaded with the address of **outstr**. With **x5** holding the address of the current character location in the **outstr** in the loop, which as a result, it is now pointing 1 past the end of the string. Calculate the length by subtracting the address of the start of the string from the address of the end of the string. A counter could have been kept for this in the loop, but in Assembly Language in the attempt to be efficient, as few instructions as possible are wanted in the loops.

This example program shows how arrays are typically handed. In this case, the strings are treated as arrays of characters. Arrays are indexed by incrementing the address pointer. Each element is one byte, so each array is incremented by one each time. If the array was an array of words, then the address would be incremented by four, the size of one element, each time.

Summary

With this chapter's instructions and commands, data can be loaded from memory, operated on in the registers, then the result saved back to memory. How to load addresses and data using only two Assembly Language instructions with **pc**-relative addresses was examined. An example program applying many of the instructions learned was used.

In the next chapter, code reusability will be addressed. After all, the uppercase program would be handy if it can be called whenever desired.

Exercises

1. Create a small program to try out all the data definition directives the Assembler provides. Assemble the program and use **objdump** to examine the data. Add some align directives and examine how they move around.
2. Explain how the **auipc**/**addi** instructions lets any 32-bit address be built in only two 32-bit instructions.
3. Write a program that converts a string to all lowercase.

4. Write a program that converts any non-alphabetic character in a **NULL** terminated string to a space.

5. Use:

   ```
   objdump -d -M no-aliases,numeric upper
   ```

   To generate an Assembly Language listing with no pseudoinstructions and compare the raw instructions to upper.S. Do the pseudoinstructions make the program more readable? Do the pseudoinstructions obscure the details of what is going on too much?

CHAPTER 6

Functions and the Stack

In this chapter, how to organize code into small independent units called **functions** is explained. This allows the building of reusable components that can easily be used from anywhere by setting up parameters and calling them.

Typically, in software development, low-level components are a starting point. Next, they are built on to create higher- and higher-level modules. So far, in this book, how to loop, perform conditional logic, and perform some arithmetic were presented. Now, how to compartmentalize code into building blocks will be given.

Introducing the **stack**; a Computer Science data structure for storing data. To build useful reusable functions, a good way to manage register usage is needed, so that all these functions don't clobber each other. In Chapter 5, "Thanks for the Memories," instructions for storing data in main memory were given. The problem with this is that this memory exists for the duration that the program runs. With small functions that run quickly, like converting to uppercase, they might need a few memory locations while they run, but when they're done, they don't need this memory anymore. Stacks provide a tool to manage register usage across function calls and a tool to provide memory to functions for the duration of their invocation.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_6

Several low-level concepts are introduced first, then these are put together to effectively create and use functions. First is the abstract data type called a stack that is a convenient mechanism to store data for the duration of a function call.

About Stacks

In Computer Science, a stack is an area of memory where there are two operations:

- **push**: Adds an element to the area.
- **pop**: Returns and removes the element that was most recently added.

This behavior is also called a **LIFO** (Last In First Out) queue.

The RISC-V instruction set does not contain any special instructions for manipulating a stack; however, the RISC-V organization publishes a document, *RISC-V ABIs Specification*, on how the main stack should work. This allows programs with components from many sources to all work together. When Linux or the ESP32-C3 SDK runs a program, an initial stack is set up for the program before control is passed to **_start** or **app_main**. The register **x2** is designated as the Stack Pointer (**sp**). Notice that **x2** is named **sp** in **gdb** and that when programs are debugged, it has a large value, something like 0x3ffffff040. This is a pointer to the current stack location.

To **push** registers to the stack and later **pop** the values from the stack, the same **load** and **store** instructions from Chapter 5, "Thanks for the Memories," are used. The only difference is that the stack pointer (**sp**) needs to be updated along with each access.

Note The RISC-V specification requires that **sp** is always 16-byte aligned. This means sp can only be added and subtracted with multiples of sixteen bytes. If **sp** is not 16-byte aligned, a bus error could result and the program will terminate. This is the same for both 32- and 64-bit processors.

To copy the single register **x5** to the stack, use the following code:

```
addi  sp, sp, -16 # must be a multiple of 16
sd    x5, 8(sp)   # save the value of x5 to the stack
```

The convention for the stack is that **sp** points to the last element on the stack and grows downwards. This is why **sp** contains a large address. The **addi** instruction allocates 16-bytes on the stack, by subtracting 16 from the **sp**. This must be done first, so an interrupt handler cannot overwrite the values.

The **sd** instruction copies **x5** to the memory location at **sp** + 8. Eight bytes are wasted here, since **x5** is only eight bytes in size. To keep the proper alignment, 16-bytes must be used. If running on a 32-bit processor, then **x5** is only four bytes and would be written it to 12(**sp**).

To pop the value at the top of the stack into register **x5**, use the following code:

```
ld    x5, 8(sp)
addi  sp, sp, 16
```

This does the reverse operation. It moves the data pointed to by 8(**sp**) from the stack to **x5** and then adds 16 to the **sp**.

Usually, a single **addi** is used followed by multiple store/load instructions to **push**/**pop** multiple registers as a set, for example,

```
addi  sp, sp, -32
sd    x5, 24(sp)
sd    x6, 16(sp)
sd    x7, 8(sp)
sd    x28, 0(sp)
...
ld    x28, 0(sp)
ld    x7, 8(sp)
ld    x6, 16(sp)
ld    x5, 24(sp)
addi  sp, sp, 32
```

Figure 6-1 shows the process of pushing a register onto the stack, and then Figure 6-2 shows the reverse operation of popping that value off the stack.

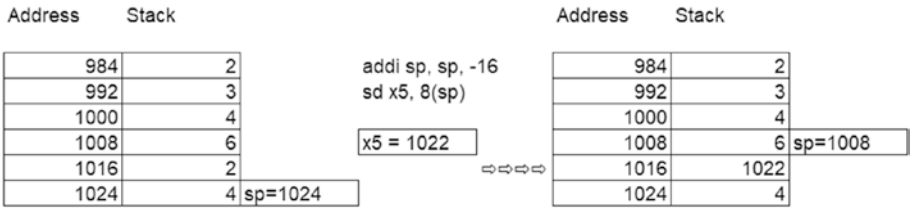

Figure 6-1. *Pushing* ***x5*** *onto the stack*

Figure 6-2. *Popping* ***x6*** *from the stack*

The stack is a useful location to store temporary values. All the functionality covered in Chapter 5, "Thanks for the Memories," can be used to **load** and **store** values there. In our usage implement them exactly as prescribed, so it will work well with code written in another language by other programmers. The details of calling functions are given and how the stack fits into this with the jump and link instructions in the next section.

Jump and Link

To call a function, first set up the ability for the function to return execution to after the point where the function is called. Do this with the **Return Address** (**ra**) register which is **x1**. To make use of **ra**, specify it in the jump instructions, **jal** or **jalr**.

To return from the function, the **jalr** instruction is used, specifying **ra** as the indirect jump register.

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

This instruction branches to the address stored in **ra** to return from the function. Since this is a common operation, there is a pseudoinstruction for this:

```
ret
```

It's important to use this instruction rather than some other branch instruction because the instruction pipeline knows about **ret** instructions and knows to continue processing instructions from where **ra** points. This way there's no performance penalty for returning from functions.

There are pseudoinstructions, where the return address is left off, it fills in **ra** (**x1**) automatically:

```
jal   offset    # jal ra, offset
jalr  rs        # jalr ra, rs, 0
```

If the function cannot be reached via a 12-bit offset, there is a **call** pseudoinstruction that will use **auipc** to build a 32-bit offset.

```
call   offset   # 1: auipc ra, %pcrel_hi(offset);
jalr ra, ra, %pcrel_lo(1b)
```

In Listing 6-1, the **jal** instruction stores the address of the following **li** instruction into **ra** then branches to **myfunc**. **Myfunc** does the useful work the function is written to do, then returns execution to the caller by having **ret** branch to the location stored in **ra**, which is the **li** instruction following the **jal** instruction.

Listing 6-1. Skeleton code to call a function and return

```
        # ... other code ...
        jal   myfunc
        li    x5, 4
        # ... more code ...
----------------------------
myfunc:         # do some work
                ret
```

There is only one **ra**, so you might be wondering what happens if another function is called? How do we preserve the original value of **ra** when function calls are nested?

Nesting Function Calls

Functions have now been successfully called and returned from a function, but without using the stack. Why was the stack introduced first and then not used? First, think of what happens if during its processing **myfunc** calls another function. Expect this to be common, as the code is written building on the functionality previously written. If **myfunc** executes a **jal** or

jalr instruction, then a new address is copied into **ra** overwriting the return address for **myfunc** and **myfunc** won't be able to return. A way is needed to keep a chain of return addresses as function after function is called. Not a chain of return addresses, but a stack of return addresses.

If **myfunc** is going to call other functions, then it needs to push **ra** onto the stack as the first thing it does and **pop** it from the stack just before it returns, for example, Listing 6-2 shows this process.

Listing 6-2. Skeleton code for a function that calls another function

```
        # ... other code ...
        jal   myfunc
        li    x5, 4
        # ... more code ...
-----------------------------
myfunc: addi  sp, sp, -16  # must be a multiple of 16
        sd    ra, 8(sp)    # push the return address to
                             the stack

# do some work ...
        jal   myfunc2
# do some more work...
        ld    ra, 8(sp)  # pop the return address from
                           the stack
        addi  sp, sp, 16
        ret
myfunc2:      # do some work ....
        ret
```

In this example, we see how convenient the stack is to store data that only needs to exist for the duration of a function call.

If a function, such as **myfunc**, calls other functions, then it must save **ra**, if it doesn't call other functions, such as myfunc2, then it doesn't need to save **ra**. Programmers often push and pop **ra** regardless, since if the function is modified later to add a function call, and the programmer forgets to add **ra** to the list of saved registers, then the program will fail to return and either go into an infinite loop or crash. The downside is that there's only so much bandwidth between the CPU and memory, so PUSHing and POPing more registers do take extra execution cycles. The tradeoff in speed versus maintainability is a subjective decision depending on the circumstances.

Calling and returning from the function is only half the story. Like in high-level languages we need to pass parameters (data) into our functions to be processed and then receive the results of the processing back in return values. Now we'll look at how to do this.

Function Parameters and Return Values

In high-level languages, functions take parameters and return the results. Assembly Language programming is no different. To do this, mechanisms could be invented, but this is counterproductive. Eventually, the code will interoperate with code written in other programming languages. The new super-fast Assembly Language functions from C code will need to be called, for example, with functions written in C.

To facilitate this, there is a set of design patterns for calling functions. The code will work reliably since others have already worked out all the bugs, if followed, plus the goal of writing interoperable code is also achieved.

The caller passes the first eight parameters in **a0** to **a7** (**x10** to **x17**). If there are additional parameters, then they are pushed onto the stack. If there are only two parameters, then only use **a0** and **a1** are used. This

means the first eight parameters are already loaded into registers and ready to be processed. Additional parameters need to be popped from the stack before being processed.

To return a value to the caller, place it in **a0** before returning. In fact, if you need to return more data, one of the parameters must be an address to a memory location where the additional data to be returned can be placed. This is the same as C where return data is placed through call by reference parameters.

Since both the caller and callee are using the same set of general-purpose registers, a protocol or convention to ensure that one doesn't overwrite the working data of the other is necessary. Next, the register management convention for the RISC-V processor is shown.

Managing the Registers

If a function is called, chances are it was written by a different programmer and the registers it will use will not be known. It would be extremely inefficient to reload all the registers every time a function is called. As a result, there are a set of rules to govern which registers a function can use and the responsibilities for saving each one.

- **x0**: The zero register cannot be changed.
- **x1** (**ra**): The called routine must preserve this as discussed in the last section.
- **x2** (**sp**): This can be freely used by the called routine but must be modified using the stack **push**/**pop** protocol.
- **x3** (**gp**): The global pointer is used by other language runtimes and the operating system; should remain unmodified.

- **x4** (**tp**): The thread pointer, like **gp**; should remain unmodified.
- **x5–x7** (**t0–t2**): Temporary registers that a function is free to use without saving. If a caller needs these, then it is responsible for saving them.
- **x8** (**s0** or **fp**): This is callee saved, so must be pushed to the stack if used in a function.
- **x9** (**s1**): This is callee saved, so must be pushed to the stack if used in a function.
- **x10–x17** (**a0–a7**): These are the function parameters. The function can use these for any other purpose modifying them freely. If the calling routine needs them saved, it must save them itself.
- **x18–x27** (**s2–s11**): These are callee saved, so must be pushed to the stack if used in a function.
- **x28–x31** (**t3–t6**): Corruptible registers that a function is free to use without saving. If a caller needs these, then it is responsible for saving them.

Summary of the Function Call Algorithm

Calling routine:

1. If any of **t1–t6** or **a0–a7** are needed save them.
2. Move the first eight parameters into registers **a0–a7**.
3. **Push** any additional parameters onto the stack.
4. Use **jal** or **jalr** to call the function

5. Evaluate the return code in **a0**.
6. Restore any of **t1**–**t6** or **a0**–**a7** saved in step 1.

Called function:

1. **Push ra** and **s0**–**s11** onto the stack if used in the routine.
2. Do the work.
3. Put the return code into **a0**.
4. **Pop ra** and **s0**–**s11** if pushed in step 1.
5. Use the **ret** instruction to return execution to the caller.

Note Steps can be saved by using **t0**–**t6** and **a0**–**a7** for function parameters, return codes, and short-term work. Then saving and restoring them around function calls is not needed. This is why **x5**, **x6**, **x7,** and **x28** were used in the examples so far.

These aren't all the rules. The coprocessors also have registers that might need saving. We'll discuss those rules when we discuss the coprocessors.

Next is a practical example that converts the uppercase program into a function that can be called with parameters to convert any strings wished.

Uppercase Revisited

Let's organize the uppercase example from Chapter 5, "Thanks for the Memories," as a proper function. The function can be moved into its own file and the makefile modified to make both the calling program and the uppercase function. To do this, first of all, create a file called **main.S** containing Listing 6-3 for the driving application.

Listing 6-3. Main program for uppercase example

```
#
# Assembler program to convert a string to
# all upper case by calling a function.
#
# a0-a2 - parameters to linux function services
# a1 - address of output string
# a0 - address of input string
# a7 - linux function number
#

.global _start              # Provide program starting address
                              to linker

_start: la    a0, instr     # start of input string
        la    a1, outstr    # address of output string

        jal   toupper

# Setup the parameters to print the resulting string
# and then call Linux to do it.
        mv    a2,a0         # return code is the length of
                              the string

        li    a0, 1         # 1 = StdOut
        la    a1, outstr    # string to print
```

```
        li    a7, 64        # linux write system call
        ecall               # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.
        li    a0, 0         # Use 0 return code
        li    a7, 93        # Service code 93 terminates
                              this program
        ecall               # Call linux to terminate
                              the program

.data
instr:  .asciz  "This is our Test String that we will
convert.\n"
outstr:      .fill  255, 1, 0
```

Next, create a file called **upper.S** containing Listing 6-4, the uppercase conversion function.

Listing 6-4. Function to convert strings to all uppercase

```
#
# Assembly Language function to convert a string to
# all upper case.
#
# a1 - address of output string
# a0 - address of input string
# s0 - original output string for length calc.
# t2 - current character being processed
# t3 - temp register for comparisons
#

.global toupper             # Allow other files to call
                              this routine
```

```
toupper:
        addi  sp, sp, -16  # allocate 16 bytes on stack
        sd    ra, 8(sp)    # push return address
        sd    s0, 0(sp)    # push s0 register
        mv    s0, a1       # save original outstr for len calc
# The loop is until null (zero) character is encountered.
loop:  lb     t2, 0(a0)    # load character
        addi  a0, a0, 1    # increment buffer pointer
# If x7 > 'z' then goto cont
        li    t3, 'z'      # load 'z' for comparison
        bgt   t2, t3, cont # branch if letter > 'z'?
# Else if x7 < 'a' then goto end if
        li    t3, 'a'      # load 'a' for comparison
        blt   t2, t3, cont # goto to end if not lowercase
# If we got here then the letter is lower case, so convert it.
        addi  t2, t2, ('A'-'a')
cont:   # end if
        sb    t2, 0(a1)    # store character to output str
        addi  a1, a1, 1    # increment buffer for next char
        li    t3, 0        # load 0 char for comparison
        bne   t2, t3, loop # loop if character isn't null

# Setup the parameters to print our hex number
# and then call Linux to do it.
        sub   a0, a1, s0   # get the len by sub’ing the
pointers

        ld    s0, 0(sp)    # pop s0
        ld    ra, 8(sp)    # pop ra
        addi  sp, sp, 16   # deallocate stack space
        ret                # Return to caller
```

To build these use the **makefile** in *Listing 6-5*.

Listing 6-5. Makefile for the uppercase function example

```
UPPEROBJS = main.o upper.o

ifdef DEBUG
DEBUGFLGS = -g
else
DEBUGFLGS =
endif

all: upper

%.o : %.S
        as -mno-relax $(DEBUGFLGS) $< -o $@

upper: $(UPPEROBJS)
        ld -o upper $(UPPEROBJS)
```

Note For the ESP32-C3 version of this function, the toupper function is renamed **mytoupper**. There is a C runtime toupper function that is included with the SDK infrastructure, so this name cannot be used.

The toupper function doesn't call any other functions, so there is no need to save **ra**. Also, **s0** is used rather than another temporary register, so it needs to be saved. These are used to demonstrate pushing and popping to the stack but could easily be avoided in this example.

Most C programmers will object that this function is dangerous. if the input string isn't **NULL** terminated, then it overruns the output string buffer—overwriting the memory past the end. The solution is to pass in a third parameter with the buffer lengths, and check in the loop that it stopped at the end of the buffer if there is no **NULL** character.

This routine only processes the core ASCII characters. It doesn't handle the localized characters for example, é won't be converted to: É.

In the uppercase function, no additional memory is required, since all the work is done with the available registers. When larger functions are coded, more memory is often needed for the variables than fit in the registers. Rather than add clutter to the **.data** section, these variables are stored on the stack. The section of the stack that holds local variables is called a stack frame.

Stack Frames

So far, the stack is used to store registers upon entry to a routine and then release them before exiting. Although the RISC-V instructions set doesn't contain explicit instructions to **push**/**pop** data to/from the stack, a strict **push**/**pop** protocol using the regular **store**/**load** instructions has been followed.

If a function requires additional memory for the duration of its run, additional space can be allocated to the stack for this data along with the space for the saved registers. Then this data can be accessed anytime in the routine via offsets to **sp**. There is a problem with this, in that it restricts the programmer from allocating more data from the stack, since this would

change the **sp** and mess up the offsets. To solve this problem, the register **x8**, which is also **s0**, can also be used as a frame pointer (**fp**). To do this, set it to the value of **sp** near the beginning of the routine, then it can safely use offsets from **fp**, without worrying about further changing **sp**.

Use of the frame pointer is entirely optional; however, a bonus is that debugging and diagnostic programs know about **fp** and can use it to give more informative information if the program crashes or a stack trace is requested.

Typically, the start of a function would:

1. Use **addi** to allocate enough space on the stack for all the saved registers and all local memory.
2. Save all necessary registers to the stack, including the frame pointer.
3. Use an **addi** instruction to set **fp** to the original value of **sp**.
4. Initialize our local variables.

Note Nothing special needs to be done to release this memory on exit, just the usual restoring of the saved registers and using **addi** to restore **sp** to its original value.

To make this concrete, a simple example is presented.

Stack Frame Example

Listing 6-6 is a simple skeletal example of a function that creates three variables on the stack. This example is sized for a 64-bit processor.

Listing 6-6. Simple skeletal function that demonstrates a stack frame

```
# Simple function that takes 2 parameters
# VAR1 and VAR2. The function adds them,
# storing the result in a variable SUM.
# The function returns the sum.
# It is assumed this function does other work,
# including calling other functions.

# Define our variables
                .EQU  VAR1, 0
                .EQU  VAR2, 8
                .EQU  SUM,  16

sumfn:
        # allocate enough on the stack for two registers and
        # three variables, rounded up to the next
          multiple of 16.
        addi  sp, sp, -48
        sd    ra, 32(sp)    # save the return address
        sd    fp, 24(sp)    # save s0/fp
        addi   fp, sp, 48   # set fp to the original sp
        sd   a0, VAR1(fp)   # save first param to memory.
        sd   a1, VAR2(fp)   # save second param to memory.

# Do a bunch of other work, but don't change fp.
# assuming a0 and a1 are used, wiping out originals
# Next restore the variables to registers and perform the
# addition
        ld   t0, VAR1(fp)
        ld   t1, VAR2(fp)
        add   t2, t0, t1
        sd   t2, SUM(fp)
```

```
# Do other work using t2 among other things

# Function Epilog
        ld    a0, SUM(fp)    # load sum to return
        ld    fp, 24(sp)     # restore s0/fp
        ld    ra, 32(sp)     # restore ra
        addi  sp, sp, 48     # release stack storage
        ret
```

Defining Symbols

In this example, the **.EQU** Assembler directive is introduced. This directive allows symbols to be defined that will be substituted by the Assembler before generating the compiled code, so the code is more readable. In this example, keeping track of which variable is on the stack makes the code hard to read and error prone. With the **.EQU** directive, each variable's offset is defined for the stack once.

The **.EQU** only defines numbers, so a whole "8(fp)" type string cannot be defined.

Macros

Another way to make the uppercase loop into a reusable bit of code is to use macros. The GNU Assembler has a powerful macro capability, with macros rather than calling a function. The Assembler creates a copy of the code in each place where it is called, substituting any parameters. Consider this alternate implementation of the uppercase program—the first file is **mainmacro.S** containing the contents of Listing 6-7.

Listing 6-7. Program to call the toupper macro

```
#
# Assembler program to convert a string to
# all upper case by calling a function.
#
# a0-a2 - parameters to linux function services
# a1 - address of output string
# a0 - address of input string
# a7 - linux function number
#

.include  "uppermacro.S"

.global _start             # Provide program starting address
                             to linker

_start:
        toupper   tststr,  buffer

# Setup the parameters to print the resulting string
# and then call Linux to do it.
        mv    a2, a0       # return is the length of the string
        li    a0, 1        # 1 = StdOut
        la    a1, buffer   # string to print
        li    a7, 64       # linux write system call
        ecall              # Call linux to output the string

# Call again to show can use twice.
        toupper   tststr2,  buffer

# Setup the parameters to print the resulting string
# and then call Linux to do it.
        mv    a2, a0       # return is the length of the string
        li    a0, 1        # 1 = StdOut
```

```
        la    a1, buffer   # string to print
        li    a7, 64       # linux write system call
        ecall              # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.
        li    a0, 0        # Use 0 return code
        li    a7, 93       # Service code 93 terminates
                             this program
        ecall              # Call linux to terminate
                             the program

.data
tststr:  .asciz  "This is our Test String that we will
convert.\n"
tststr2:  .asciz  "A second string to upper case!!\n"
buffer:   .fill  255, 1, 0
```

The macro to make the string all uppercase is in **uppermacro.S** containing Listing 6-8.

Listing 6-8. Macro version of the toupper function

```
#
# Assembly Language function to convert a string to
# all upper case.
#
# a1 - address of output string
# a0 - address of input string
# t2 - current character being processed
# t3 - temp register for comparisons
# t4 - original output string for length calc.
#
```

```
# label 1 = loop
# label 2 = cont

.MACRO  toupper      instr, outstr
        la    a0, \instr
        la    a1, \outstr
        mv    t4, a1       # save original outstr for len calc
# The loop is until null (zero) character is encountered.
1:      lb    t2, 0(a0)    # load character
        addi  a0, a0, 1    # increment buffer pointer
# If x7 > 'z' then goto cont
        li    t3, 'z'      # load 'z' for comparison
        bgt   t2, t3, 2f   # branch if letter > 'z'?
# Else if x7 < 'a' then goto end if
        li    t3, 'a'      # load 'a' for comparison
        blt   t2, t3, 2f   # goto to end if not lowercase
# If we got here then the letter is lower case, so convert it.
        addi  t2, t2, ('A'-'a')
2:      # end if
        sb    t2, 0(a1)    # store character to output str
        addi  a1, a1, 1    # increment buffer for next char
        li    t3, 0        # load 0 char for comparison
        bne   t2, t3, 1b   # loop if character isn't null

# Setup the parameters to print our hex number
# and then call Linux to do it.
        sub   a0, a1, t4   # get the len by sub'ing the
                             pointers

        .ENDM
```

Include Directive

The file **uppermacro.S** defines the macro to convert a string to uppercase. The macro does not generate any code, it just defines the macro for the Assembler to insert wherever it is called from. This file doesn't generate an object (*.o) file, rather it is included by whichever file needs to use it.

The **.include** directive

```
.include "uppermacro.S"
```

takes the contents of this file and inserts it at this point, so that the source file becomes larger. This is done before any other processing. This is like the C #**include** preprocessor directive.

Macro Definition

A macro is defined with the **.MACRO** directive. This gives the name of the macro and lists its parameters. The macro ends at the following **.ENDM** directive. The form of the directive is

```
.MACRO      macroname   parameter1, parameter2, ...
```

Within the macro, specify the parameters by preceding their name with a backslash. For instance, **parameter1** to place the value of **parameter1**. The toupper macro defines two parameters **instr** and **outstr**:

```
.MACRO      toupper     instr, outstr
```

The parameters are used in the code with **instr** and **outstr**. These are text substitutions and need to result in correct Assembly syntax or an error will be generated.

Labels

The labels "loop" and "cont" are replaced with the labels "1" and "2." This takes away from the readability of the program. The reason numeric labels are used is that otherwise an error would be generated that a label was defined more than once, if the macro is used more than once. The trick here is that the Assembler lets numeric labels be defined as many times as desired. To reference them in the code, use

```
bgt   t2, t3, 2f   # branch if letter > 'z'?
bne   t2, t3, 1b   # loop if character isn't null
```

The **f** after the **2** means the next label **2** in the forward direction. The **1b** means the next label **1** in the backwards direction.

To prove that this works, toupper is called twice in the **mainmacro.S** file, to show everything works and that this macro can be used as many times as wished.

Why Macros?

Macros substitute a copy of the code at every point they are used. This will make the executable file larger, for example, when using

```
objdump -d  uppermacro
```

two copies of code are inserted. With functions there is no extra code generated each time. This is why functions are quite appealing, even with the extra work of dealing with the stack.

The reason macros get used is performance. Most RISC-V devices have a gigabyte or more of memory—a lot of room for multiple copies of code. Remember that whenever branching, the execution pipeline must be restarted, making branching an expensive instruction. With macros, the **jal** branch is eliminated to call the function and the **ret** branch to return.

Also eliminated are any instructions to save and restore the registers used. If a macro is small and is used a lot, there can be considerable execution time savings.

Note Notice in the macro implementation of toupper that only the registers **t0–t4** and **a0–a1** were used. This avoids using any registers important to the caller. There is no standard on how to regulate register usage with macros, like there is with functions, so it is up to the programmer to avoid conflicts and strange bugs.

Macros can also be used to make the code more readable and easier to write, as described in the next section.

Using Macros to Improve Code

Using **ld**, **sd**, and **addi** to manipulate the stack is clumsy and error prone, as a lot of time is spent cutting and pasting the code from other places to try and get it correct. It would be nice if there were pseudoinstructions to **push** and **pop** the stack, with macros these can be created. Consider Listing 6-9:

Listing 6-9. Define four macros for pushing and popping the stack

```
.MACRO          PUSH1 register
                addi  sp, sp, -16
                sd    \register, 8(sp)
.ENDM
.MACRO          POP1 register
                ld    \register, 8(sp)
                addi  sp, sp, 16
.ENDM
```

```
.MACRO          PUSH2 register1, register2
                addi  sp, sp, -16
                sd    \register1, 8(sp)
                sd    \register2, 0(sp)
.ENDM
.MACRO          POP2  register1, register2
                ld    \register2, 0(sp)
                ld    \register1, 8(sp)
                addi  sp, sp, 16
.ENDM
```

This simplifies the code since these can be used to write code like in Listing 6-10:

Listing 6-10. Use push and pop macros

```
Myfunction:
        PUSH2 ra, fp
  # function body ...
        POP2  ra, fp
        ret
```

This makes writing the function prologues and epilogues easier and clearer.

Summary

In this chapter, the RISC-V stack and how it's used to help implement functions were covered. Also, how to write and call functions as a first step to creating libraries of reusable code was explained. How to manage register usage, so there aren't any conflicts between calling programs and functions instructions were given. The function calling protocol, allowing interoperating with other programming languages, instructions

were given. Also, the defining stack-based storage for local variables and how to use this memory was introduced. Finally, the GNU Assembler's macro ability as an alternative to functions in certain performance critical applications was covered.

Exercises

1. If coding for an operating system where the stack grows upwards, how are the **ld**, **sd**, and **addi** instructions coded?
2. Suppose there is a function that uses registers **x8**, **x9**, **x20**, **x23**, and **x31** and this function calls other functions. Code the prologue and epilogue of this function to store and restore the correct registers to/from the stack.
3. Write a function to convert text to all lowercase. Have this function in one file and a main program in another file. In the main program, call the function three times with different test strings.
4. Convert the lowercase program in *Exercise 3* to a macro. Have it run on the same three test strings to ensure it works properly.
5. Why does the function calling protocol have some registers that need to be saved by the caller and some by the callee? Why not make all saves by one or the other?

CHAPTER 7

Linux Operating System Services

In the sample programs so far, the ability to exit programs and display a string was needed. For Linux, the operating system services were called directly to do this. In all high-level programming languages, there is a runtime library that includes wrappers for calling the operating system. This makes it appear that these services are part of the high-level language. In this chapter, what these runtime libraries do under the covers to call Linux and what services are available will be looked at.

If only using the ESP32-C3, then skip this chapter. The contents only apply to Linux either running on a SBC like the Starfive Visionfive 2 or on the QEMU emulator. The ESP32-C3 SDK uses regular function calls as explained in Chapter 6, "Functions and the Stack."

The syntax for calling the operating system and the error codes returned will be reviewed. Help from the GNU C compiler, utilizing some C header files to get the definitions needed for the Linux service call numbers, rather than using magic numbers like 64 and 93 will give assistance.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_7

So Many Services

Linux is a powerful, full featured operating system with over 25 years of development. Linux powers devices from watches all the way up to supercomputers. One of the keys to this success is the richness and power of all the services that it offers.

There are slightly over four-hundred Linux service calls, covering all of these is beyond the scope of this book, and more the topic for a book on Linux System Programming. In this section, the mechanisms and conventions for calling these services and some examples, so how to go from the Linux documentation to writing code quickly, are covered. The Linux documentation for all these services is quite good. It is oriented entirely to C programmers, so anyone else using it must know enough C to convert the meaning to what is appropriate for the language used.

Calling Convention

Two system calls one to write ASCII data to the console and the second to exit the program have been used. The calling convention for system calls is different from that for functions, using the I-type instruction Environment Call (**ecall**) that raises a software exception invoking an exception handling routine in the Linux kernel. This exception mechanism is the same one used if the program tries to access protected memory or divide by zero. The calling convention is:

1. **a0**–**a6**: Input parameters, up to seven parameters for the system call
2. **a7**: The Linux system call number
3. Invoke the operating system with "**ecall**"
4. **a0**: The return code from the call

The software exception is a clever way to call routines in the Linux kernel without knowing where they are stored in memory. It also provides a mechanism to run at a higher security level while the call executes. Linux checks if the correct access rights to perform the requested operation are used and gives back an error code like EACCES (13) if denied.

Although it does not follow the function calling convention from Chapter 6, "Functions and the Stack," the Linux system call mechanism preserves all registers not used as parameters or the return code. When system calls require a large block of parameters, they tend to take a pointer to a block of memory as one parameter, which then holds all the data they need. Hence, most system calls don't use that many parameters and Linux recently limited the number of parameters to five.

Where to get those magic Linux system call numbers for all those useful services is described next.

Finding Linux System Call Numbers

The Linux system call number for exit is 93 and 64 is the number to write to a file. These seem rather cryptic, so where can these numbers be looked up? Can something symbolic in programs be used rather than these magic numbers? The Linux system call numbers are defined in the C include file:

```
/usr/include/asm-generic/unistd.h
```

In this file, there are define statements such as the following:

```
#define __NR_write 64
```

This defines the symbol __NR_write to represent the magic number 64 for the write Linux system call.

Next, a similar method for the service return codes is needed, so if something goes wrong, why it failed can be pinpointed.

Return Codes

The return code for these functions is usually zero or a positive number for success and a negative number for failure. The negative number is the negative of the error codes from the C include file:

```
/usr/include/errno.h
```

This file includes several other files, the main ones that contain most of the actual error codes are

```
/usr/include/asm-generic/errno.h
/usr/include/asm-generic/errno-base.h
```

We'll see how to use the constants from these files in our code when we get to a sample program.

For example, the open call, to open a file, returns a file descriptor if it is successful. A file descriptor is a small positive number, then a negative number if it fails, where it is the negative, it is one of the constants in errno.h.

If you've programmed in C, you know many of the C runtime functions take structures as parameters. The Linux service calls are the same and we'll look at dealing with these next.

Structures

Many Linux services take pointers to blocks of memory as parameters. The contents of these blocks of memory are documented with C structures, so Assembly Language programmers must reverse engineer the C and duplicate the memory structure. For instance, the nanosleep service lets the program sleep for several nanoseconds; it is defined as

```
int nanosleep(const struct timespec *req, struct
timespec *rem);
```

then the struct timespec is defined as

```
struct timespec {
            time_t tv_sec;      /* seconds */
            long   tv_nsec;     /* nanoseconds */
            };
```

Now, determine that these are two 64-bit integers, then define in Assembly Language, as follows:

```
timespecsec:   .dword   0
timespecnano:  .dword   100000000
```

To use them, load their address into the registers for the first two parameters:

```
        la         a0, timespecsec
        la         a1, timespecsec
```

The nanosleep function is used in Chapter 8, "Programming GPIO Pins," but this is typical of what it takes to directly call some Linux services.

Next, decide how to make these calls easier to use. Are they wrapped in Assembly Language functions or use another method?

About Wrappers

Rather than figure out all the registers each time to call a Linux service, a library of routines or macros to make will be developed to make the job easier. The C programming language includes function call wrappers for all the Linux services. How to use these is given in Chapter 9, "Interacting with C and Python."

Rather than duplicate the work of the C runtime library by developing wrapper functions, a library of Linux system calls using the GNU Assembler's macro functionality can be developed. However, this will not be developed for all the functions, just the functions needed. Most programmers do this, and over time their libraries become quite extensive.

A problem with macros is that often several variants with different parameter types are required. For instance, sometimes a macro could be called with a register as a parameter and other times with an immediate value.

Now that the theory of using Linux services has been explained, instructions to complete a program that uses a collection of these will be given.

Converting a File to Uppercase

In this chapter, a complete program to convert the contents of a text file to all uppercase is presented. The toupper function from Chapter 6, "Functions and the Stack" is used, and more practice coding loops and if statements are realized.

To start, a library of file I/O routines to read from the input file is needed, then write the uppercase version to another file. If you have done any C programming, these should look familiar, since the C runtime provides a thin layer over these services. Create a file: **fileio.S** containing Listing 7-1.

Note The file extension is a capital S; this is important as this allows the use of C include files as will be discussed shortly.

Listing 7-1. Macros to help read and write files

```
# Various macros to perform file I/O
#
# The fd parameter needs to be a register.
# Uses a0, a1, a2, a3, a7.
# Return code is in a0.

#include <asm/unistd.h>

.equ   O_RDONLY, 0
.equ   O_WRONLY, 1
.equ   O_CREAT,  0100
.equ   O_EXCL,   0200
.equ   S_RDWR,   0666
.equ   AT_FDCWD, -100

.macro  openFile    fileName, flags
        li    a0, AT_FDCWD
        la    a1, \fileName
        li    a2, \flags
        li    a3, S_RDWR    # RW access rights
        li    a7, __NR_openat
        ecall
.endm
.macro  readFile   fd, buffer, length
        mv    a0, \fd      # file descriptor
        la    a1, \buffer
        li    a2, \length
        li    a7, __NR_read
        ecall
.endm
.macro  writeFile   fd, buffer, length
```

```
        mv    a0, \fd      # file descriptor
        la    a1, \buffer
        mv    a2, \length
        li    a7, __NR_write
        ecall
.endm
.macro  flushClose  fd
#fsync syscall
        mv    a0, \fd
        li    a7, __NR_fsync
        ecall

#close syscall
        mv    a0, \fd
        li    a7, __NR_close
        ecall
.endm
```

A main program to orchestrate the process is needed next. Call this **main.S**, again with the capital S file extension, containing the contents of Listing 7-2.

Listing 7-2. Main program for case conversion program

```
#
# Assembler program to convert a string to
# all upper case by calling a function.
#
# a0-a2, a7 - used by macros to call linux
# s1 - input file descriptor
# s2 - output file descriptor
# t0 - number of characters read
#
```

```
#include <asm/unistd.h>
#include "fileio.S"

.equ   BUFFERLEN, 250

.global _start                    # Provide program starting
                                    address to linker

_start: openFile    inFile, O_RDONLY
        mv          s1, a0        # save file descriptor
        bgez        a0, nxtfil    # pos number file opened ok
        li          a1, 1         # stdout
        la          a2, inpErrsz  # Error msg
        lw          a2, 0(a2)
        writeFile   a1, inpErr, a2 # print the error
        j           exit

nxtfil: openFile    outFile, O_CREAT+O_WRONLY
        mv          s2, a0        # save file descriptor
        bgez        a0, loop      # pos number file opened ok
        li          a1, 1
        la          a2, outErrsz
        lw          a2, 0(a2)
        writeFile   a1, outErr, a2
        j           exit

# loop through file until done.
loop:   readFile    s1, buffer, BUFFERLEN
        mv          t0, a0        # Keep the length read
        li          t1, 0         # Null terminator for string

        # setup call to toupper and call function
        la          a0, buffer    # first param for toupper
        add         a1, a0, t0    # addr to put null,
                                    buffer + len
```

```
        sb          t1, 0(a1)     # put null at end of string.
        la          a1, outBuf
        jal         toupper

        writeFile   s2, outBuf, t0

        li          t1, BUFFERLEN
        beq         t0, t1, loop

        flushClose  s1
        flushClose  s2

# Setup the parameters to exit the program
# and then call Linux to do it.
exit:   li    a0, 0    # Use 0 return code
        li    a7, __NR_exit
        ecall          # Call Linux to terminate

.data
inFile:       .asciz  "main.S"
outFile:      .asciz  "upper.txt"
buffer:       .fill   BUFFERLEN + 1, 1, 0
outBuf:       .fill   BUFFERLEN + 1, 1, 0
inpErr:       .asciz  "Failed to open input file.\n"
inpErrsz:     .word   .-inpErr
outErr:       .asciz  "Failed to open output file.\n"
outErrsz:     .word   .-outErr
```

To build these source files, add a new rule to the **makefile**, to build .S files with **gcc** rather than **as**, as shown in the next section.

Building .S Files

The **makefile** is contained in Listing 7-3.

Listing 7-3. Makefile for our file conversion program

```
UPPEROBJS = main.o upper.o

ifdef DEBUG
DEBUGFLGS = -g
else
DEBUGFLGS =
endif

all: upper

%.o : %.S
        gcc -mno-relax $(DEBUGFLGS) -c $< -o $@

%.o : %.s
        as -mno-relax $(DEBUGFLGS) $< -o $@

upper: $(UPPEROBJS)
        ld -o upper $(UPPEROBJS)
```

This program uses the **upper.S** file from Chapter 6, "Functions and the Stack," that contains the function version of the uppercase logic.

A rule to compile the two .S files with **gcc** rather than **as** was added. Most people think of **gcc** as the GNU C Compiler, but it stands for the GNU Compiler Collection and can compile several other languages in addition to C including Assembly Language. The clever trick that gcc supports when this is done is the ability to add C preprocessor commands to the Assembly Language code.

When a .S (the capital is important) file is compiled with **gcc**, it processes all C #**include** and #**define** directives before processing the Assembly Language instructions and directives. This means standard C include files can be included for their symbols, as long as the files don't contain any C code, or conditionally excludes the C code when processed by the GNU Assembler.

Note There is an interesting conflict between RISC-V Assembly Language and the C Preprocessor, namely, RISC-V Assembly Language uses **#** to specify comments and the C preprocessor starts all commands with **#**. The C preprocessor runs first and if it encounters any comments that start with preprocessor commands, such as **#if**, **#ifdef**, **#endif**, or **#define**, it will try to process them, and this could result in errors. If the comments in **upper.S** that start with "# If …", started with "# if …" then a preprocessor error results, give it a try.

The Linux kernel consists of both C and Assembly Language code. For the definition of constants that are used by both code bases, they do not want to make the definitions in two places and risk errors from differences. Thus, all the Assembly Language code in the Linux kernel are in .S files and use various C include files including **unistd.h**.

Using this technique, our Linux function numbers are no longer magic numbers and will be correct and readable.

When a **.s** (lowercase) file with **gcc** is processed, it assumes pure Assembly Language code is desired and will not run things through the C preprocessor first.

Notice that if this program is built, it is only 3KB in size. This is one of the appeals of pure Assembly Language programming. There is nothing extra added to the program—every byte is controlled—no mysterious libraries or runtimes added.

Next, details of opening a file are given.

Opening a File

The Linux **openat** service is typical of a Linux system service. It takes four parameters:

1. **Directory File Descriptor**: File descriptor to the folder that filename is open relative to. If this is the magic number AT_FDCWD, then it means open relative to the current folder.
2. **Filename**: The file to open as a NULL terminated string.
3. **Flags**: To specify whether it is open for reading, writing, or creating the file. Some .**EQU** directives with the required values are included (using the same names as in the C runtime).
4. **Mode**: The access mode for the file when the file is created. Defines were included, but in octal these are the same as the parameters to the **chmod** Linux command.

The return code is either a file descriptor or an error code. Like many Linux services, the call fits this in a single return code by making errors negative and successful results positive.

The C runtime has both **open** and **openat** routines—the **open** routine calls the **openat** Linux service with AT_FDCWD for the first parameter as used here.

Error Checking

Most books neglect to promote good programming practices for error checking. The sample programs are kept as small as possible, so the main ideas being explained are not lost in a sea of details. This is the first program where any return codes are tested, partly because enough code had to be developed to be able to do it and because error checking code does not reveal any new concepts.

File open calls are prone to failing. The file might not exist, perhaps, because it is the wrong folder, or there are insufficient access rights to the file. Generally, check the return code to every system call, or function called, but practically speaking programmers tend to only check those return codes that are likely to fail. In this program, the two file open calls are checked. Checking every return code could make the code listings too long to include in this book, so don't take this code as an example, do the error checking in the real code.

1. Copy the file descriptor to a register that won't be overwritten, so move it to **s1**.

   ```
   mv    s1, a0        # save file descriptor
   ```

2. Test if it is positive, and if so go on to the next bit of code.

   ```
   bgez  a0, nxtfil    # pos number file opened ok
   ```

   If the branch isn't taken, then **openFile** returned a negative number.

3. To generate an error message to the console use **writeFile** routine to write an error message to **stdout**, then branch to the end of the program to exit.

```
li         a1, 1          # stdout
la         a2, inpErrsz   # Error msg string length
lw         a2, 0(a2)      # Load string length
writeFile  a1, inpErr, a2 # print the error
j          exit
```

In the .data section, the error message is defined as follows:

```
inpErr: .asciz       "Failed to open input file.\n"
inpErrsz: .word  .-inpErr
```

Then **.asciz** is standard. For **writeFile**, the length of the string is needed to write to the console. In Chapter 1, "Getting Started," we counted the characters in the string and put the hard-coded number in the code. This can be done here too, but error messages start getting long and counting the characters seems like something the computer should do. A routine can be written, like the C library's **strlen**() function to calculate the length of a NULL terminated string. Instead, a little GNU Assembler trickery can be used by adding a .word directive right after the string and initializing it with ".-inpErr".

The "." is a special Assembler variable that contains the current address the Assembler is on as it works. Hence, the current address right after the string minus the address of the start of the string is the length. Now the wording of the error message can be revised without needing to count the characters each time.

Most applications contain an error module, so if a function fails, the error module is called. Then the error module is responsible for reporting and logging the error. This way error reporting can be made quite sophisticated without cluttering up the rest of the code with error handling code. Another problem with error handling code is that it tends to be untested. Often bad things can happen when an error finally does happen, and problems with the previously untested code manifest.

Looping

In our loop, we

1. Read a block of two-hundred and fifty characters from the input file.
2. Append a NULL terminator.
3. Call toupper.
4. Write the converted characters to the output file.
5. If we aren't done, branch to the top of the loop.

Check if it is done with

```
li        t1, BUFFERLEN
beq       t0, t1, loop
```

t0 contains the number of characters returned from the read service call. If it equals the number of characters requested, then we branch to loop. If it doesn't equal exactly, then either encountered the end of file, so the number of characters returned is less (and possibly 0), or an error occurred, in which case the number is negative. Either way, we are done and fall through to the program exit.

Summary

In this chapter, we gave an overview of how to call the various Linux system services. We covered the calling convention and how to interpret the return codes. We didn't cover the purpose of each call and referred the user to the Linux documentation instead.

We presented a program to read a file, convert it to uppercase, and write it out to another file. This is our first chance to put together what we learned in Chapters 1–6 to build a full application, with loops, if statements, error messages, and file I/O.

In the next chapter, we will use Linux service calls to manipulate the GPIO pins on a RISC-V SBC.

Exercises

1. The files this program operates on are hard coded in the **.data** section. Change them, play with them, and generate some errors to see what happens. Single step through the program in **gdb** to ensure you understand how it works.
2. Modify the program to convert the file to all lowercase.
3. Convert **fileio.S** to use callable functions rather than macros. Change **main.S** to call these functions.
4. Another I-type instruction that causes an exception in **ebreak**. If running in **gdb**, executing this instruction is the same as setting a breakpoint. Adding **ebreak** instructions can be a simple way to place breakpoints in code. Add an **ebreak** statement to **main.S** and compile for debugging. Run under **gdb** to see it work. Remember to remove the **ebreak** instructions before running outside of **gdb** as this will cause the program to crash.

CHAPTER 8

Programming GPIO Pins

Most Single Board Computers have a set of General Purpose I/O (**GPIO**) pins that can be used to control homemade electronics projects. In this chapter, GPIO ports on a Starfive Visionfive 2 are looked at. Also, programming GPIO pins from Assembly Language will be shown.

An experiment with a breadboard containing several LEDs and resistors is given, so some real code can be written. The GPIO pins will be programmed in two ways: first of all, by using the Linux device driver and secondly, by accessing the GPIO controller's hardware registers directly.

GPIO Overview

The Starfive Visionfive 2 has a 40-pin GPIO header. They either provide power or are generally programmable:

- **Pins 1 and 17**: Provide +3.3V DC power.
- **Pins 2 and 4**: Provide +5V DC power.
- **Pins 6, 9, 14, 20, 25, 30, 34, and 39**: Provide electrical ground.
- Other pins are programmable for general purposes.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_8

The programmable pins can be used for output, whether they output power or not (binary 1 or 0). They can be read to see if power is provided, for instance, if they are connected to a switch.

However, this isn't all there is to GPIO; besides the functions presented so far, a number of the pins have alternate functions that can be selected programmatically. For instance, pins 3 and 5 can support the I2C standard that allows two microchips to talk to each other.

Also, there are pins that can support two serial ports to connect to radios or printers. In addition, there are pins that support Pulse Width Modulation (PWM) and Pulse-Position Modulation (PPM) that convert digital to analog and are handy for controlling electric motors.

For the first program, Linux will do the heavy lifting, which is typical for how to control hardware when there is a device driver available.

In Linux, Everything is a File

The model for controlling devices in Linux is to map each device to a file. The file appears under either **/dev** or **/sys** and can be manipulated with the same Linux service calls that operate on regular files. The GPIO pins are no different. There is a Linux device driver for them that controls the pin operations via application programs opening files, then reading and writing data to them.

The files to control the GPIO pins all appear under the **/sys/class/gpio** folder. By writing short text strings to these files, the operation of the pins can be controlled.

1. To programmatically control pin 22, the documentation for the GPIO header needs to be consulted at: `https://doc-en.rvspace.org/VisionFive2/PDF/VisionFive2_40-Pin_GPIO_Header_UG.pdf`. This contains a diagram mapping the physical pins on the Starfive board to the GPIO

functionality behind the pin. In this case, it would be determined that pin 22 is GPIO50. The physical pin is required to set up the physical wiring, then the GPIO name is used to control the pin from software.

Now the pin can be controlled from software. To tell the driver to work with GPIO50, write the string "50" to /sys/class/gpio/export. If this succeeds, then the pin can be controlled. The driver creates the following files in a gpio50 folder:

- **/sys/class/gpio/gpio50/direction**: Specifies whether the pin is for input or output.
- **/sys/class/gpio/gpio50/value**: Sets or reads the value of the pin.
- **/sys/class/gpio/gpio50/edge:** Sets an interrupt to detect value changes.
- **/sys/class/gpio/gpio50/active_low:** Inverts the meaning of 0 and 1.

2. Next, set the direction for the pin, either use it for input or for output. Write "in" or "out" to the direction file.

3. Write to the value file for an output pin or read the value file for an input pin. To turn on a pin, write "1" to value and to turn it off, we write "0". When activated, the GPIO pin provides +3.3V.

   When we are done with a pin, we should write its pin number to /sys/class/gpio/unexport. However, this will be done automatically when the program terminates.

We can do all this with the macros we created in Chapter 7, "Linux Operating System Services," in **fileio.S**. In fact, by providing this interface, the GPIO pins can be controlled via any programming language capable of reading and writing files, that is mostly every single one.

Flashing LEDs

To demonstrate programming the GPIO, connect some LEDs to a breadboard, then make them flash in sequence.

Each of the three LEDs will be connected to a GPIO pin (in this case 50, 51, and 54), then to the ground through a resistor. The resistor is needed because the GPIO is specified to keep the current under 16mA, or the circuits could be damaged.

Most electronics kits come with several 220 Ohm resistors. By Ohm's law, I = V/R, these would cause the current to be 3.3V/220Ω = 15mA, so just right. A resistor in series with the LED is required since the LED's resistance is quite low (typically around 13 Ohms and variable).

Warning LEDs have a positive and a negative side. The positive side needs to connect to the GPIO pin, reversing it could damage the LED. The longer lead is the positive lead.

Figure 8-1 shows how the LEDs and resistors are wired up on a breadboard.

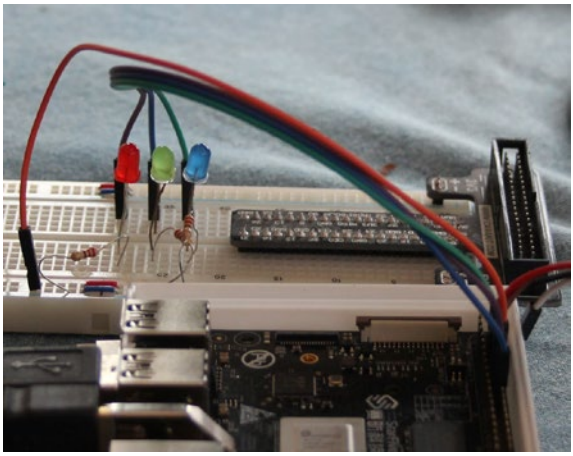

Figure 8-1. *Breadboard with LEDs and resistors installed*

Figure 8-2 shows a schematic of the flashing LEDs hardware to help with setting it up.

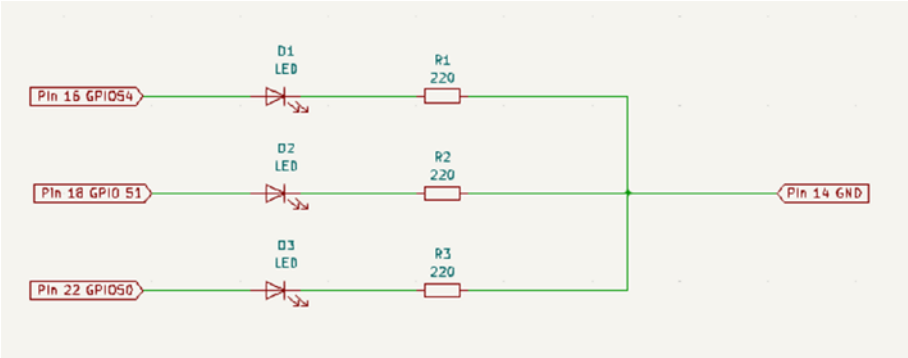

Figure 8-2. *Schematic for the flashing LEDs*

To perform the file I/O, a modified version of **fileio.S** is used. The only difference from Listing 7-1 is the addition of a **writeFileReg** routine that will accept the length parameter as a register rather than an immediate constant. The new routine is in Listing 8-1 and needs to be added to **fileio.S**.

Listing 8-1. New writeFileReg routine to add to fileio.S

```
.macro  writeFileReg   fd, buffer, length
        mv    a0, \fd      # file descriptor
        la    a1, \buffer
        mv    a2, \length
        li    a7, __NR_write
        ecall
.endm
```

Initially, define a set of macros in **gpiomacros.S**. containing Listing 8-2 that uses the updated version of **fileio.S** to perform the various GPIO functions.

Listing 8-2. Macros to control the GPIO pins

```
# Various macros to access the GPIO pins
# on the Raspberry Pi.
#
# t3 - file descriptor.
#

#include "fileio.S"

# Macro nanoSleep to sleep .1 second
# Calls Linux nanosleep entry point.
# Pass a reference to a timespec in both a0 and a1
# First is input time to sleep in seconds and nanoseconds.
# Second is time left to sleep if interrupted (which we ignore)
.macro  nanoSleep
        la    a0, timespecsec
        la    a1, timespecsec
        li    a7, __NR_nanosleep
        ecall
```

```
.endm
.macro  GPIOExport  pin
        openFile    gpioexp, O_WRONLY
        mv          t3, a0      # save the file descriptor
        writeFile   t3, \pin, 2

        flushClose  t3
.endm
.macro  GPIODirectionOut   pin
        # copy pin into filename pattern
        la          a1, \pin
        la          a2, gpiopinfile
        addi        a2, a2, 20
        lb          a3, 0(a1)   # load pin digit
        addi        a1, a1, 1   # increment for second digit
        sb          a3, 0(a2)   # store to filename
        addi        a2, a2, 1   # increment for second digit
        lb          a3, 0(a1)
        sb          a3, 0(a2)
        openFile    gpiopinfile, O_WRONLY
        mv          t3, a0      # save the file descriptor
        writeFile   t3, outstr, 3
        flushClose  t3
.endm
.macro  GPIOWrite   pin, value
        # copy pin into filename pattern
        la          a1, \pin
        la          a2, gpiovaluefile
        addi        a2, a2, 20
        lb          a3, 0(a1)    # load pin digit
        addi        a1, a1, 1    # increment for second digit
        sb          a3, 0(a2)    # store to filename
```

```
        addi        a2, a2, 1    # increment for second digit
        lb          a3, 0(a1)
        sb          a3, 0(a2)
        openFile    gpiovaluefile, O_WRONLY
        mv          t3, a0       # save the file descriptor
        writeFile   t3, \value, 1
        flushClose  t3
.endm

.data
timespecsec:   .dword   0
timespecnano:  .dword   100000000
gpioexp:    .asciz  "/sys/class/gpio/export"
gpiopinfile: .asciz "/sys/class/gpio/gpioxx/direction"
gpiovaluefile: .asciz "/sys/class/gpio/gpioxx/value"
outstr:     .asciz  "out"
            .align  4 # save users having to do this.
.text
```

Now add a controlling program **main.S** containing Listing 8-3 to orchestrate the process.

Listing 8-3. Main program to flash the LEDs

```
#
# Assembler program to flash three LEDs connected to the
# Starfive Visionfive 2 GPIO port.
#
# t0 - loop variable to flash lights 10 times
#

#include "gpiomacros.S"
```

```
.global _start                # Provide program starting
address to linker
_start: GPIOExport  pin50
        GPIOExport  pin54
        GPIOExport  pin51
        nanoSleep

        GPIODirectionOut pin50
        GPIODirectionOut pin54
        GPIODirectionOut pin51
        # setup a loop counter for 10 iterations
        li    t0, 10

loop:   GPIOWrite   pin50, high
        nanoSleep
        GPIOWrite   pin50, low
        GPIOWrite   pin54, high
        nanoSleep
        GPIOWrite   pin54, low
        GPIOWrite   pin51, high
        nanoSleep
        GPIOWrite   pin51, low
        #decrement loop counter and see if we loop
        addi        t0, t0, -1      # Subtract 1 from loop
        bnez        t0, loop        # If not 0 then loop

_end:   li    a0, 0 # Use 0 return code
        li    a7, __NR_exit
        ecall       # Call Linux to terminate

pin50:      .asciz  "50"
pin54:      .asciz  "54"
pin51:      .asciz  "51"
```

```
low:        .asciz  "0"
high:       .asciz  "1"
```

This program is a straightforward application of the Linux system service calls learned in Chapter 7, "Linux Operating System Services."

Note The **/sys/class/gpio** files have restricted access, so the program must be run using **sudo**.

Moving Closer to the Metal

For Assembly Language programmers, the previous example is not satisfying. When programming in Assembly Language, usually devices are directly manipulated for performance reasons or to perform operations that simply cannot be done in high-level programming languages. In this section, the GPIO controller will be interacted with directly.

Warning Backup work before running the program, since there may be a need to power off and power back on again. In the previous section, the device driver provided a level of protection, so damage could not easily be caused. Now that the code is written directly to the hardware registers, there is no such protection. If a mistake is made and the wrong registers are manipulated, the Visionfive's operation may be interfered with causing it to crash or lock up.

Virtual Memory

In Chapter 5, "Thanks for the Memories," how to access memory was explained. These memory addresses aren't physical memory addresses, rather they're virtual memory addresses. As a Linux process, the program is given a large virtual address space that can expand well beyond the amount of physical memory.

Within this address space, some of it is mapped to physical memory to store the Assembly Language instructions, **.data** sections, and 8 MB stack. Furthermore, Linux may swap some of this memory to secondary storage like the SD Card as it needs more physical memory for other processes. There is a lot of complexity in the memory management process to allow dozens of processes to run independently of each other, with each thinking it has the whole system to itself.

In the next section, access to specific physical memory addresses is desired, but when that access is requested, Linux returns a virtual memory pointer that is different from the physical address asked for. This is okay, as behind the scenes the memory management hardware in the RISC-V CPU will do the memory translations between virtual and physical memory.

In Devices, Everything is Memory

The GPIO controller has 16 enable registers and 16 write registers; however, these cannot be read or written to like the RISC-V CPU's registers. The RISC-V instruction set does not know anything about the GPIO controller and there are no special instructions to support it. The way to access these registers is by reading and writing to specific memory locations. There is circuitry in the Visionfive's System on a Chip (SoC) that will see these memory reads and writes, and redirect them to the GPIO's registers. This is how most hardware communicates.

The memory address for the GPIO registers is 0x13040000. This address is configurable by the operating system, so you need to check what it is for what you are doing. The easiest way to confirm the true value is to use the command:

```
dmesg
```

In its output is something like:

```
 2.741289] starfive_jh7110-pinctrl 13040000.gpio: StarFive GPIO
chip registered 64 GPIOs
```

Note The output of **dmesg** could be quite long. Use:

```
dmesg | grep gpio
```

---

to scan for this entry.

This is a kernel message from initializing the GPIO controller chip, which gives useful information of where the registers are.

Sounds easy—load addresses into registers, then reference the memory stored there. Not so fast, if this were tried our program would crash with a memory access error. This is because these memory addresses are outside those assigned to the program. The first job then is to get access by mapping this memory into the process's address space.

This leads us back to everything being a file in Linux. There is a file that will give a pointer, which can be used to access these memory locations, as follows:

1. Open the file **/dev/mem**.
2. Then ask **/dev/mem** to map the registers for GPIO into the memory space. We do this with the Linux **mmap** service. **Mmap** takes the following parameters:

- **a0**: Hint for the virtual address we would like. We don't really care and will use NULL, which gives Linux complete freedom to choose.
- **a1**: Length of region. Should be a multiple of 4096, the memory page size.
- **a2**: Memory protection required.
- **a3**: File descriptor to access **/dev/mem**.
- **a4**: Offset into physical memory. In our case 0x13040000.

This call will return a virtual address in **a0** that maps to the physical address we asked for. This function returns a small negative number if it fails, that can be looked up in **errno.h**.

Registers in Bits

Although these registers have been mapped to memory locations, they don't always act like memory. These aren't like CPU registers or real memory. The circuitry is intercepting memory reads and writes to these locations, but only acting on things that it understands. In the previous sections, the Linux device driver for GPIO hid all these details.

The GPIO registers are 32-bits in size. Data can only be transferred to/from these registers using 32-bit **lw**/**sw** instructions. The address must be aligned with the register exactly or a bus error will result. For instance, if **a2** contains the address to a GPIO address, is read it with

```
ld   a1, 0(a2)
```

A bus error results when the program is run, because the GPIO controller cannot provide 64-bits of data. The following must be used:

```
lw   a1, 0(a2)
```

GPIO Enable Registers

The first thing to do is configure the pins used for output. There is a bank of sixteen registers to enable the GPIO pins. Each register controls four GPIO pins allowing for a total of 64 GPIOs. Each pin gets eight bits in one of these registers to configure it. These are read-write registers.

To use these registers, the protocol is to

1. Read the register
2. Set the bits for what is wanted
3. Write the value back

Note We must be careful not to affect other bits in the register.

Although each pin has room for eight bits in the register, the enable register only uses six of those eight bits. To enable the pin, these bits need to be set to zero. Hardware registers are wired to minimize circuitry costs, so often how they are set is counter-intuitive to what programmers expect.

To find the correct register

Register address = base address + (gpio number / 4) * 4

This might look counter-intuitive but suppose the address of GPIO 50 is required then using integer arithmetic 50 / 4 = 12, the remainder is discarded, then 12 * 4 = 48 which is then the address of the correct register.

The position within the register of the bits is determined by the remainder on dividing by four. What is desired is a shift amount so that data can be shifted into the correct position for bit operations. The calculation is

shift = 8 * (gpio pin remainder on dividing by 4)

This gives the number of bits to shift things into position.

A define

```
.equ    DOEN_MASK, 0x3f
```

is used to clear the six bits. The calculation is

Register value and (not (DOEN_MASK << shift))

Try running through this calculation by hand to ensure it clears the desired six bits, while leaving all other bits untouched.

Remember Although multiplication and division have not been covered, shifting bits left is equivalent to multiplying by powers of two and shifting bits right is equivalent to dividing by powers of two, so multiplying and dividing by four and eight can be handled with shift operations.

GPIO Output Set Registers

There are 16 registers for setting/clearing pins. These registers are 0x40 bytes above the base GPIO memory address. An **.equ** for this is created:

```
.equ    setregoffset, 0x40
```

This is added to the GPIO memory address before adding the register address. Seven bits are used to control each GPIO pin; hence, that mask is defined as

```
.equ    DOUT_MASK, 0x7f
```

To set the GPIO signal high, light the LED, the six high-order bits are cleared and a one is set in the low order bit. To clear the GPIO signal, turn off the LED, all seven bits are set to zero. Thus, the formula to set a bit is:

Register value and (not (DOUT_MASK << shift)) + (1 << shift)

With the registers in hand, a program is presented to flash the LEDs, accessing the GPIO hardware registers directly.

More Flashing LEDs

In this section, the flashing LEDs program will be repeated, but this time using mapped memory and by accessing the GPIO's registers directly. First, the macros that do the nitty-gritty work from Listing 8-4 go in **gpiomem.S**.

Listing 8-4. GPIO support macros using mapped memory

```
# Various macros to access the GPIO pins
# on the Starfive Visionfive 2.
#
# t6 - memory map address.
# Macros use registers: a0, a1, a2, a3, a4, a5, a7, t5
#

#include "fileio.S"

.equ   pagelen, 512
.equ   setregoffset, 0x40
.equ   PROT_READ, 1
.equ   PROT_WRITE, 2
.equ   MAP_SHARED, 1
.equ   DOUT_MASK, 0x7f
.equ   DOEN_MASK, 0x3f
```

```
# Macro to map memory for GPIO Registers
.macro mapMem
        openFile     devmem, O_RDWR    # open /dev/mem
        mv           a4, a0            # fd for memmap
        # check for error and print error msg if necessary
        bgez         a4, 1f            # pos number file opened ok
        li           a1, 1             # stdout
        lw           a2, memOpnsz      # Error msg
        writeFileReg a1, memOpnErr, a2 # print the error
        j            _end

# Setup can call the mmap2 Linux service
1:      ld           a5, gpioaddr   # address we want / 4096
        li           a1, pagelen    # size of mem we want
        li           a2, (PROT_READ + PROT_WRITE) # mem prot
        li           a3, MAP_SHARED # mem share options
        li           a0, 0          # let linux choose a address
        li           a7, __NR_mmap  # mmap service num
        ecall                       # call service
        mv           t6, a0         # keep the returned address
        # check for error and print error msg if necessary
        bgez         t6, 2f         # pos number file opened ok
        li           a1, 1          # stdout
        lw           a2, memMapsz   # Error msg
        writeFileReg a1, memMapErr, a2 # print the error
        j            _end
2:
.endm

# Macro nanoSleep to sleep .1 second
# Calls Linux nanosleep entry point
# which is function __NR_nanosleep.
```

```
# Pass a reference to a timespec in both a0 and a1
# First is input time to sleep in seconds and nanoseconds.
# Second is time left to sleep if interrupted (which we ignore)
.macro nanoSleep
        la    a0, timespecsec
        la    a1, timespecsec
        li    a7, __NR_nanosleep
        ecall
.endm
.macro  GPIODirectionOut    pin
        li    a0, \pin      # pin to turn on
        srli  a1, a0, 2     # pin offset div by 4
        slli  a1, a1, 2     # mult by 4, now multiple of 4
        sub   a2, a0, a1    # shift value start with remainder
        slli  a2, a2, 3     # multiply by 8 (bits per gpio)
        li    a3, DOEN_MASK # mask
        sll   a3, a3, a2    # shift into position
        not   a3, a3        # ones complement bits for anding

        add   a4, t6, a1    # add to base address
        lwu   a5, 0(a4)     # load register value
        and   a0, a3, a5    # and value to reg value

        sw    a0, 0(a4)     # write to the register
.endm
.macro  GPIOTurnOn   pin
        li    a0, \pin      # pin to turn on
        srli  a1, a0, 2     # pin offset div by 4
        slli  a1, a1, 2     # mult by 4, now multiple of 4
        sub   a2, a0, a1    # shift value start with remainder
        slli  a2, a2, 3     # multiply by 8 (bits per gpio)
        li    a3, DOUT_MASK # mask
```

```
        sll   a3, a3, a2    # shift into position
        not   a3, a3        # ones complement bits for anding

        add   a4, t6, a1    # add to base address
        addi  a4, a4, setregoffset # add offset to write regs
        lwu   a5, 0(a4)     # load register value
        and   a0, a3, a5    # and value to reg value

        li    a3, 1         # load 1 to set on
        sll   a3, a3, a2    # shift into place
        add   a0, a0, a3    # add to register value
        sw    a0, 0(a4)     # write to the register
.endm
.macro  GPIOTurnOff   pin
        li    a0, \pin      # pin to turn on
        srli  a1, a0, 2     # pin offset div by 4
        slli  a1, a1, 2     # mult by 4, now multiple of 4
        sub   a2, a0, a1    # shift value start with remainder
        slli  a2, a2, 3     # multiply by 8 (bits per gpio)
        li    a3, DOUT_MASK # mask
        sll   a3, a3, a2    # shift into position
        not   a3, a3        # ones complement bits for anding

        add   a4, t6, a1    # add to base address
        addi  a4, a4, setregoffset # add offset to write regs
        lwu   a5, 0(a4)     # load register value
        and   a0, a3, a5    # and value to reg value

        sw    a0, 0(a4)     # write to the register
.endm

.data
timespecsec:   .dword   0
timespecnano:  .dword   100000000
```

```
devmem:         .asciz  "/dev/mem"
memOpnErr:      .asciz  "Failed to open /dev/mem\n"
memOpnsz:       .word  .-memOpnErr
memMapErr:      .asciz  "Failed to map memory\n"
memMapsz:       .word  .-memMapErr
                .align  4 # relign after strings
gpioaddr:       .dword   0x13040000    # mem address of gpio
registers
.text
```

Now the driving program **mainmem.S** contains Listing 8-5, which is quite similar to the last one. The main differences are in the macros.

Listing 8-5. Main program for the memory mapped flashing lights

```
#
# Assembler program to flash three LEDs connected to the
# Raspberry Pi GPIO port using direct memory access.
#
# s11 - loop variable to flash lights 10 times
#

#include "gpiomem.S"

.global _start                  # Provide program
starting address
_start:
        mapMem
        nanoSleep

        GPIODirectionOut 50
        GPIODirectionOut 54
        GPIODirectionOut 51
        # setup a loop counter for 10 iterations
        li           s11, 10
```

```
loop:
        GPIOTurnOn   50
        nanoSleep
        GPIOTurnOff  50
        GPIOTurnOn   54
        nanoSleep
        GPIOTurnOff  54
        GPIOTurnOn   51
        nanoSleep
        GPIOTurnOff  51
        #decrement loop counter and see if we loop
        addi  s11, s11, -1   # Subtract 1 from loop reg
        bnez  s11, loop      # If not 0 then loop

_end:   li   a0, 0  # Use 0 return code
        li   a7, __NR_exit
        ecall       # Call Linux to terminate
```

The main program is the same as the first example, except that it includes a different set of macros.

The first thing needed is to call the **mapMem** macro. This opens /**dev**/**mem** and sets up and calls the **mmap** service as we described in the section *In Devices, Everything is Memory*. We store the returned address into **t6**, so that it is easily accessible from the rest of the macros. There is error checking on the file open and **mmap** calls since these can fail.

GPIOTurnOn in Detail

In this section, the GPIOTurnOn macro will be examined in detail, using GPIO 50 as an example.

1. The first instruction loads 50 into register **a0**.

```
li    a0, \pin      # pin to turn on
```

The value 50 is divided by 4 by shifting the value right two bits yielding 12, then shifted left two bits to multiply by 4, yielding 48 which is the offset for the GPIO output register that contains the bits for GPIO 50. This value is kept in register **a1**.

```
srli  a1, a0, 2     # pin offset div by 4
slli  a1, a1, 2     # mult by 4, now multiple of 4
```

2. To get the shift value, the remainder is required. This is obtained by subtracting the calculated offset from the original pin number, which yields two, stored in register **a2**.

```
sub   a2, a0, a1    # shift value start with rem
```

3. This value is multiplied by eight to get the number of bits to shift values over. This is accomplished by shifting register **a2** left three bits.

```
slli  a2, a2, 3     # multiply by 8 (bits per gpio)
```

4. Now the DOUT_MASK is loaded into register **a3** and shifted into position, using the shift value in **a2**.

```
li    a3, DOUT_MASK # mask
sll   a3, a3, a2    # shift into position
```

5. To get a mask to and with the register, the ones complement is taken using the **not** pseudoinstruction.

```
not   a3, a3        # ones complement bits for anding
```

6. To calculate the GPIO address, add the base GPIO address to the offset calculated previously, placing the result in **a4**.

```
add   a4, t6, a1    # add to base address
```

7. Add the offset of the write registers to **a4**.

```
addi  a4, a4, setregoffset # add offset to write
registers
```

8. Load the current value of the register into **a5**.

```
lwu   a5, 0(a4)     # load register value
```

9. And the mask value with the current register value and place the result in **a0**.

```
and   a0, a3, a5    # and value to reg value
```

10. Load the value of one to turn the LED on into register **a3**.

```
li    a3, 1         # load 1 to set on
```

11. Shift the one value into position.

```
sll   a3, a3, a2    # shift into place
```

12. Add the shifted one value to the current register value.

```
add   a0, a0, a3    # add to register value
```

13. Store new register value back into the register.

```
sw    a0, 0(a4)     # write to the register
```

The other routines are similar in how they work.

Root Access

To access **/dev/mem**, root access is needed, so run this program with root access via

```
sudo ./flashmem
```

If this is not done, then the file open will fail. Accessing **/dev/mem** is powerful and gives access to all memory and all hardware devices.

This is a restricted operation, so we need to be root. Programs that directly access memory are usually implemented as Linux device drivers or kernel loadable modules, but then installing these also requires root access. A virus or other malware would love to have access to all physical memory.

Summary

In this chapter, we built on everything we've learned so far, to write a program to flash a series of LEDs attached to the GPIO ports on the Starfive Visionfive 2. This was done in two ways:

1. Using the GPIO device driver by accessing the files under **/sys/class/gpio**
2. Using direct memory access by asking the device driver for **/dev/mem** to give us a virtual block of memory corresponding to the GPIO's control registers

Controlling devices are a key use case for Assembly Language programming. Hopefully, this chapter gave you a flavor for what is involved.

In Chapter 9, “Interacting with C and Python,” how to interact with high-level programming languages like C and Python will be taught

Exercises

1. Not all device interactions can be abstracted by reading or writing files. Linux allows a general function, **ioctl**, to define special operations. Consider a network interface, what are some functions needed to control with **ioctl**?
2. Why does the GPIO controller pack so much functionality into each register? Why not have a separate register for each pin? What are the pros and cons of each approach?
3. Why does Linux consider access to the GPIO controller dangerous and restrict usage to root?

CHAPTER 9

Interacting with C and Python

In the early days of microcomputers, like the Apple II, complete applications were coded in Assembly Language, such as the first spreadsheet program VisiCalc. Many video games were also written in Assembly Language to squeeze every bit of performance possible out of the hardware. However, modern compilers, like the GNU C compiler, generate good code and microprocessors are much faster, as a result most applications are written in a collection of programming languages, where each excels at a specific function. For example, video games today are commonly written in C, C++ or even C#, and Assembly Language for performance, or to access parts of the video hardware not exposed through the graphics library used.

In this chapter, using components written in other languages from the Assembly Language code will be explained, as well as how other programming languages can make use of the fast-efficient code when writing in Assembly Language.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_9

Calling C Routines

On the ESP32-C3, the programs have already called the Espressif SDK's **puts** function, which is written in C, so for the ESP32-C3, everything is ready to go.

Under Linux, to call C functions, the program must be restructured. The C runtime has a **_start** label that must be called first to initialize itself before calling the program, which it does by calling a **main** function. If the **_start** label is left in, an error that **_start** is defined more than once is created. Similarly, the Linux terminate program service will not be called anymore, instead we will return from main and let the C runtime do that along with any other cleanup it performs.

Note In this situation, the **-mno-relax** assembler command line argument is not required. This is because the C runtime will initialize the global pointer, allowing this optimization.

To include the C runtime, it could be added to the command line arguments in the **ld** command in the **makefile**. However, it is easier to compile the program with the GNU C compiler (which includes the GNU Assembler), then it will link in the C runtime automatically. To compile the program, use the following:

```
gcc -o myprogram myprogram.S
```

This will call **as** on **myprogram.S** and then perform the **ld** command including the C runtime.

The C runtime gives a lot of capabilities including wrappers for most of the Linux System Services. There is an extensive library for manipulating NULL-terminated strings, routines for memory management, and routines to convert between all the data types.

Printing Debug Information

One handy use of the C runtime is to print out data to trace what the program is doing. We wrote a routine to output the contents of a register in hexadecimal, and we could write more Assembly Language code to extend this, or we could just get the C runtime to do it. After all, if we are printing out trace or debugging information, it doesn't need to be performant, rather easy to add to the code.

For this example, we'll use the C runtime's **printf** function to print out the contents of a register in both decimal and hexadecimal format. This routine will be packaged as a macro, and all the registers that might be corrupted will be preserved. This way the macro can be called without worrying about register conflicts. Also, a macro can be provided to print a string for either logging or formatting purposes.

The C **printf** function is mighty, as it takes a variable number of arguments depending on the contents of a format string. There is extensive online documentation on **printf**, so for a fuller understanding, please have a look. We will call the collection of macros **debug.S**. and it contains the code from Listing 9-1. This macro assumes all the registers are 64-bits. For a 32-bit implementation, see Exercise 9-8.

Listing 9-1. Debug macros that use the C runtime's printf function

```
# Various macros to help with debugging

# These macros preserve[1]  all registers.
# Beware they will change the condition flags.

.altmacro

.macro  saveReg regNum
        sd   x\regNum, \regNum * 8 (sp)
.endm
```

```
.macro  loadReg regNum
        ld   x\regNum, \regNum * 8 (sp)
.endm

.macro  saveRegs
        .set    i, 1
        .rept   31
                saveReg %i
                .set   i, i+1
        .endr
.endm

.macro  restoreRegs
        .set    i, 1
        .rept   31
                loadReg   %i
                .set      i, i+1
        .endr
.endm

.macro  printReg    reg
        addi   sp, sp, -256
        saveRegs

        mv     a2, x\reg  # for the %d
        mv     a3, x\reg  # for the %x
        li     a1, \reg
        la     a0, ptfStr # printf format str
        call   printf     # call printf
        restoreRegs
        addi   sp, sp, 256
.endm
```

```
.macro  printStr    str
        addi   sp, sp, -256
        saveRegs

        la     a0, 1f     # load print str
        call   printf     # call printf

        restoreRegs
        addi   sp, sp, 256
        j      2f         # branch around str
1:      .ascii        \str
        .asciz        "\n"
        .align        4
2:
.endm

.data
ptfStr: .asciz "x%d = %32ld, 0x%016lx\n"
.align 8
.text
```

Preserving State

First, push registers **x1–x31**, either use these registers or **printf** might change them. They are not saved as part of the function calling protocol. At the end, restore these. This makes calling the macros as minimally disruptive to the calling code as possible.

It is unfortunate that each instruction can only save or restore one register at a time and since there are 31 registers; this means 32 instructions to push all these registers and another 32 to pop all of them off of the stack.

Remember The stack must always grow/shrink in multiples of 16-bytes.

Rather than writing out all these instructions, the GNU Assembler's macro facility is used to generate this repetitive code. In this case, the **.altmacro** syntax is used that allows the use of variables and create loops. One macro performs the loop and calls another macro to generate the code. This is necessary to change the macro variable i into a macro parameter that can be embedded in the Assembly Language code.

Calling Printf

The C function is called with these arguments:

```
printf("x%c = %32ld, 0x%016lx\n", reg, xreg, xreg);
```

Since there are four parameters, they are set into **a0**-**a3**. In **printf,** each string, starting with a percentage sign ("%"), takes the next parameter and formats it according to the next letter:

- **c** for character
- **d** for decimal
- **x** for hex
- **0** means 0 pad
- **l** for long meaning 64-bits
- A number specifies the length of the field to print

Note It is important to move the value of the register to **a2** and **a3** first since populating the other registers might wipe out the passed in value if printing **a0** or **a1**. If the register is **a2** or **a3**, one of the mv instructions does nothing. Luckily, we don't get an error or warning, so we don't need a special case.

Now the details of how to pass this format string to **printf** will be shown.

Passing a String

In the **printStr** macro, a string is passed to print. Assembly Language does not handle strings, so the string is embedded in the code with an **.ascii** directive, then branched around it.

There is an **.align** directive right after the string, since Assembly Language instructions must be word aligned. It is good practice to add an **.align** directive after strings, since other data types will load faster when word aligned.

Generally, adding data to the code section is not a best practice, but for the macro, this is the easiest way. The assumption is that the debug calls will be removed from the final code. If too many strings are added, this could make **pc** relative offsets too large to be resolved. If this happens, strings may need to be shortened, or some may need to be removed.

Next, a program is needed to test the **printf** macro.

Register Masking Revisited

In Chapter 8, "Programming GPIO Pins," sample code to mask a GPIO pins output register was provided. Besides seeing the LEDs flash, the code could be run under **gdb** to see the internals in operation. Now, a minimal

example.S will be created and code will be added to build the GPIO output register mask and call the new debug macros to see the result in Listing 9-2.

Listing 9-2. example.S to demonstrate the debug routines

```
#
# Example of some calculations to show printReg and printStr.
#

.include "debug.S"

.global main              # Provide program starting address

.equ   DOUT_MASK, 0x7f

# Calculates the mask value needed to use a GPIO
# output register.
main:
        li     a0, 50        # pin to turn on - GPIO50
        srli   a1, a0, 2     # pin offset div by 4
        slli   a1, a1, 2     # mult by 4, now multiple of 4
        sub    a2, a0, a1    # shift value start with remainder
        slli   a2, a2, 3     # multiply by 8 (bits per gpio)
        li     a3, DOUT_MASK # mask
        sll    a3, a3, a2    # shift into position
        not    a3, a3        # ones complement bits for anding
        printStr "Register a0 GPIO register"
        printReg 10          # x10 = a0
        printStr "Register a1 GPIO register offset"
        printReg 11          # x11 = a1
        printStr "Register a2 bit shift value"
        printReg 12          # x12 = a2
```

```
        printStr "Register a3 mask value for right bits"
        printReg 13          # x13 = a3

        li     a0, 0         # return code
        ret
```

The **makefile**, in Listing 9-3, for this is quite simple.

Listing 9-3. Makefile for updated example.S

```
example: example.S debug.S
        gcc -o example example.S
```

If we compile and run the program, we will see

```
user@starfive:~/Chapter9$ make
gcc -o example example.S
user@starfive:~/Chapter9$ ./example
Register a0 GPIO register
x10 =                                50, 0x0000000000000032
Register a1 GPIO register offset
x11 =                                48, 0x0000000000000030
Register a2 bit shift value
x12 =                                16, 0x0000000000000010
Register a3 mask value for right bits
x13 =                          -8323073, 0xffffffffff80ffff
```

Besides adding the debug statements, notice how the program is restructured as a function. The entry point is **main**.

By just adding the C runtime, a powerful tool-chest to save time is available as the full Assembly Language application is developed. On the downside, however, notice the executable has grown to over 9 KB.

Knowing how to call C routines from the Assembly Language code, next the reverse will be done and instructions for calling Assembly Language from C will be given.

Calling Assembly Routines from C

A typical scenario is to write most of the application in C, then call Assembly Language routines in specific use cases. If the function calling protocol from Chapter 6, "Functions and the Stack," is followed, C will not be able to tell the difference between the Assembly Language functions and any functions written in C.

As an example, let's call the **toupper** function from Chapter 6, "Functions and the Stack," from C. Listing 9-4 contains the C code for **uppertst.c** to call the Assembly function.

Listing 9-4. Main program to show calling the toupper function from C

```
//
// C program to call our Assembly
// toupper routine.
//

#include <stdio.h>

extern int mytoupper( char *, char * );

#define MAX_BUFFSIZE 255
int main()
{
        char *str = "This is a test.";
        char outBuf[MAX_BUFFSIZE];
        int len;

        len = mytoupper( str, outBuf );
        printf("Before str: %s\n", str);
```

```
        printf("After str: %s\n", outBuf);
        printf("Str len = %d\n", len);
        return(0);
}
```

The **makefile** is in *Listing 9-5*.

Listing 9-5. Makefile for C and the toupper function

```
uppertst: uppertst.c upper.S
        gcc -o uppertst uppertst.c upper.S
```

The name of the **toupper** function needed to be changed to **mytoupper** since there is already a **toupper** function in the C runtime, and this led to a multiple definition error. This had to be done in both the C and the Assembly Language code. Otherwise, the function is the same as in Chapter 6, "Functions and the Stack."

The parameters and return code for the function to the C compiler must be defined with the following code:

```
extern int mytoupper( char *, char * );
```

This should be familiar to all C programmers, as this must be done for C functions as well. Usually, all these definitions would be gathered and put in a header (**.h**) file.

As far as the C code is concerned, there is no difference between using this Assembly Language function versus if it was written in C. When the program is compiled and run, the program looks like the following:

```
user@starfive:~/Chapter9$ make
gcc -o uppertst uppertst.c upper.S
user@starfive:~/Chapter9$ ./uppertst
Before str: This is a test.
After str: THIS IS A TEST.
Str len = 16
```

The string is uppercase as expected, but the string length appears one greater than expected. That is because the length includes the NULL character that is not the C standard. To use this a lot with C, subtract 1, so that the length is consistent with other C runtime routines.

Packaging the Code

The Assembly Language code could be left in individual object (**.o**) files, but it is more convenient for programmers using the object files to package them together into a library. This way the user of the Assembly Language routines just needs to add one library to get all of the code, rather than possibly dozens of .o files. In Linux there are two ways to do this, the first way is to package the code together into a static library that is linked into the program. The second method is to package the code as a shared library that lives outside the calling program that can be shared by several applications.

Static Library

To package the code as a static library, use the Linux **ar** command. This command will take a number of **.o** files and combine them into a single file, by convention lib<ourname>.a, that can then be included into a **gcc** or **ld** command. To do this, we modify the **makefile** to build this way as demonstrated in Listing 9-6.

Listing 9-6. **`Makefile`** to build **`upper.s`** into a statically linked library

```
LIBOBJS = upper.o

all: uppertst2

%.o : %.S
        as $(DEBUGFLGS) $< -o $@
```

```
libupper.a: $(LIBOBJS)
        ar -cvq libupper.a upper.o

uppertst2: uppertst.c libupper.a
        gcc -o uppertst2 uppertst.c libupper.a
```

If we build and run this program, we get:

```
user@starfive:~/Chapter9$ make
as  upper.S -o upper.o
ar -cvq libupper.a upper.o
a - upper.o
gcc -o uppertst2 uppertst.c libupper.a
user@starfive:~/Chapter9$ ./uppertst2
Before str: This is a test.
After str: THIS IS A TEST.
Str len = 16
```

The only difference to the last example is that **as** is used first to compile **upper.S** into **upper.o**, and then use **ar** to build a library containing the routine. To distribute the library, include **libupper.a**, a header file with the C function definitions and some documentation. Even if not selling, or otherwise distributing the code, building libraries internally can help organizationally to share code among programmers and reduce duplicated work. In the next section, shared libraries are explored, which is another Linux facility for sharing code.

Shared Library

Shared libraries are much more technical than statically linked libraries, because they place the code in a separate file from the executable and are dynamically loaded by Linux as needed. There are several issues that will be touched on, such as versioning and library placement in the file system.

If packaging the code as a shared library, this section provides a starting point and demonstrates that it applies to Assembly Language code as much as C code.

The shared library is created with the **gcc** command, giving it the **-shared** command line parameter to indicate what shared library will be created and then the **-soname** parameter to name it.

To use a shared library, it must be in a specific place in the filesystem.

1. New places can be added on, but for this example a place created by the C runtime, namely, **/usr/local/lib**, will be used.
2. After the library is built, it will be copied here, and a couple of links to it will be created. These steps are all required as part of the shared library versioning control system. To use the shared library **libup.so.1**, **-lup** is included on the **gcc** command to compile **uppertst3**.
3. The **makefile** is presented in Listing 9-7.

Listing 9-7. Makefile for building and using a shared library

```
LIBOBJS = upper.o

all: uppertst3

%.o : %.S
        as $(DEBUGFLGS) $< -o $@

libup.so.1.0: $(LIBOBJS)
        gcc -shared -Wl,-soname,libup.so.1 -o libup.so.1.0:
        $(LIBOBJS)
        gcc -shared -Wl,-soname,libup.so.1 -o libup.so.1.0
        $(LIBOBJS)
        mv libup.so.1.0 /usr/local/lib
```

```
        ln -sf /usr/local/lib/libup.so.1.0 /usr/local/lib/
        libup.so.1
        ln -sf /usr/local/lib/libup.so.1.0 /usr/local/lib/
        libup.so
ldconfig

uppertst3: libup.so.1.0
        gcc -o uppertst3 uppertst.c -lup
```

4. If run, several commands will fail. To copy the files to **/usr/local/lib**, root access is needed, so use the **sudo** command. This is done to the **make** command:

   ```
   sudo make
   ```

   This causes Linux to search all the folders that hold shared libraries and update its master list. Run this once after successfully compiling the library, or Linux will not know it exists.

Note Placing **-lup** on the end of the command to build **uppertst3**, after the file that uses it, is important, or unresolved externals will result when it is built.

The following is the sequence of commands to build and run the program:

```
user@starfive:~/Chapter9$ sudo make -B
as  upper.S -o upper.o
gcc -shared -Wl,-soname,libup.so.1 -o libup.so.1.0: upper.o
gcc -shared -Wl,-soname,libup.so.1 -o libup.so.1.0 upper.o
mv libup.so.1.0 /usr/local/lib
```

```
ln -sf /usr/local/lib/libup.so.1.0 /usr/local/lib/libup.so.1
ln -sf /usr/local/lib/libup.so.1.0 /usr/local/lib/libup.so
ldconfig
gcc -o uppertst3 uppertst.c -lup
user@starfive:~/Chapter9$ ./uppertst3
Before str: This is a test.
After str: THIS IS A TEST.
Str len = 16
```

If **objdump** is used to look inside **uppertst3**, the code for the **mytoupper** routine will not be found, instead, in the **main** code the following code will be found:

```
 6e6:  f1bff0ef            jal    600 <mytoupper@plt>
```

which calls:

```
0000000000000600 <mytoupper@plt>:
 600:  00002e17            auipc  t3,0x2
 604:  a28e3e03            ld     t3,-1496(t3) # 2028
                                  <mytoupper@Base>
 608:  000e0367            jalr   t1,t3
 60c:  00000013            nop
```

Gcc inserted this indirection into the code, so the loader can fix up the address when it dynamically loads the shared library.

As a final technique, mixing Assembly Language and C code in the same source code file will be shown.

Embedding Assembly Language Code inside C Code

The GNU C Compiler allows Assembly code to be embedded right in the middle of C code. It contains features to interact with C variables and labels and cooperate with the C compiler for register usage.

Listing 9-8 is a simple example, where the core algorithm for the **toupper** function is embedded inside the C main program, name it **uppertst4.c**.

Listing 9-8. Embedding the Assembly routine directly in C code

```
//
// C program to embed our Assembly Language
// toupper routine inline.
//

#include <stdio.h>

#define MAX_BUFFSIZE 255
int main()
{
        char *str = "This is a test.";
        char outBuf[MAX_BUFFSIZE];
        int len;

        asm
        (
                "mv t4, %2\n"
                "loop:  lb  t2, 0(%1)\n"
                "addi   %1, %1, 1\n"
                "li     t3, 'z'\n"
                "bgt    t2, t3, cont\n"
```

```
                "li     t3, 'a'\n"
                "blt    t2, t3, cont\n"
                "addi   t2, t2, ('A'-'a')\n"
                "cont:  sb  t2, 0(%2)\n"
                "addi   %2, %2, 1\n"
                "li     t3, 0    \n"
                "bne    t2, t3, loop\n"
                "sub    %0, %2, t4\n"
                : "=r" (len)
                : "r" (str), "r" (outBuf)
                : "t2", "t3", "t4", "a0", "a1"
        );

        printf("Before str: %s\n", str);
        printf("After str: %s\n", outBuf);
        printf("Str len = %d\n", len);
        return(0);
}
```

The **asm** statement allows Assembly Language code to be embedded directly into the C code. By doing this, an arbitrary mixture of C and Assembly could be written. The comments are stripped out from the Assembly Language code, so the structure of the C and Assembly Language is easier to read. The general form of the **asm** statement is

```
asm asm-qualifiers ( AssemblerTemplate
                        : OutputOperands
                [ : InputOperands]
                [ : Clobbers ] ]
                [ : GotoLabels])
```

The parameters are

- **AssemblerTemplate**: A C String containing the Assembly Language code. There are macro substitutions that start with **%** to let the C compiler insert the inputs and outputs.
- **OutputOperands**: A list of variables or registers returned from the code. This is required, since it is expected that the routine does something. In this case, "**=r**" (len) where the **=r** means an output register that is wanted to go into the C variable len.
- **InputOperands**: List of input variables or registers used by the routine. In this case, **"r" (str)**, **"r" (outBuf)** meaning two registers are wanted: one holding **str** and one holding **outBuf**. It is fortunate that C string variables hold the address of the string, which is what is wanted in the register.
- **Clobbers**: A list of registers that we use and will be clobbered when the code runs. In this case, "**t2**," "**t3**," "**t4**," "**a0**," and "**a1**."
- **GotoLabels**: A list of C program labels that the code might want to jump to. Usually, this is an error exit. If a jump to a C label is done, warn the compiler with a **goto asm-qualifier**.

Label the input and output operands, otherwise the compiler will assign them names %0, %1, … as was used in the Assembly Language code.

Since this is a single C file, it is easy to compile with the following code:

```
gcc -o uppertst4 uppertst4.c
```

Running the program produces the same output as the last section.

If the program is disassembled, the C compiler avoids using the clobber registers entirely. It will load up the input registers from the variables on the stack, before the code executes and then copies the return value from the assigned register to the variable len on the stack. It does not give the same registers originally used, but that is not a problem.

This routine is straightforward and does not have any side effects. If the Assembly Language code is modifying things behind the scenes, add a volatile keyword to the **asm** statement to make the C compiler be more conservative on any assumptions it makes about the code.

In the next section, calling the Assembly Language code from the popular Python programming language is discussed.

Calling Assembly from Python

If the functions that follow the Linux function calling protocol from Chapter 6, "Functions and the Stack," are written, the documentation on how to call C functions for any given programming language can be followed. For example, Python has a good capability to call C functions in its **ctypes** module. This module requires packaging the routines into a shared library.

Since Python is an interpreted language, linking static libraries cannot be done, but dynamically loading and calling shared libraries can be accomplished. The techniques shown here for Python have matching components in many other interpreted languages.

The hard part is already done, which is building the shared library version of the uppercase function, so all that must be done is to call it from Python. Listing 9-9 is the Python code for **uppertst5.py**.

Listing 9-9. Python code to call **mytoupper**

```
from ctypes import *

libupper = CDLL("/usr/local/lib/libup.so")

libupper.mytoupper.argtypes = [c_char_p, c_char_p]
libupper.mytoupper.restype = c_int

inStr = create_string_buffer(b"This is a test!")
outStr = create_string_buffer(250)

len = libupper.mytoupper(inStr, outStr)

print(inStr.value.decode())
print(outStr.value.decode())
print(len)
```

The code is fairly simple as follows:

1. Import the **ctypes** module to use it.
2. Load the shared library with the **CDLL** function. This is an unfortunate name since it refers to Windows DLLs, rather than something more operating system neutral.
3. The next two lines are optional, but good practice. They define the function parameters and return type to Python to do extra error checking.

In Python, strings are immutable, meaning they cannot be changed, and they are in Unicode, meaning each character takes up more than one byte. The strings need to be in regular buffers, that can be changed, and be in ASCII rather than Unicode.

To make a string ASCII in Python:

1. Put a "**b**" in front of the string, which means to make it a byte array using ASCII characters.
2. The **create_string_buffer** function in the **ctypes** module creates a string buffer that is compatible with C (and hence Assembly Language) to use.

We then call the function and print the inputs and outputs. There are quite a few good Python IDEs for Linux. The Thonny Python IDE as shown in Figure 9-1 will be used to test the program.

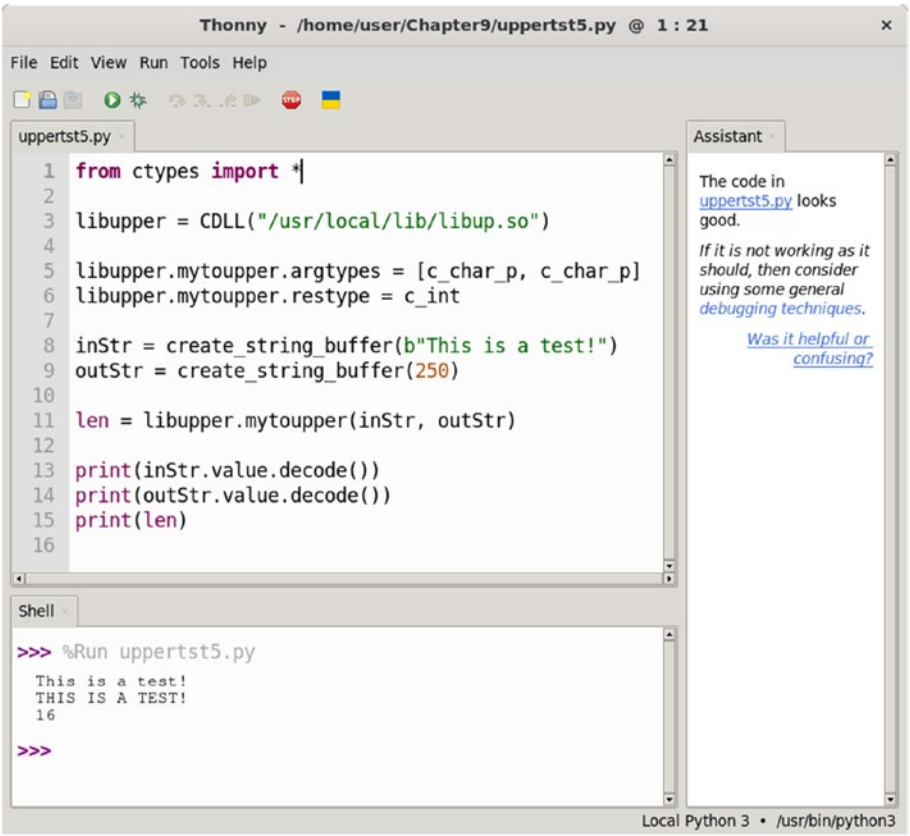

Figure 9-1. *The Python program running in the Thonny IDE*

Summary

In this chapter, calling C functions from Assembly Language code was looked at. The standard C runtime was used to develop some debug helper functions to make developing Assembly Language code easier. Then the reverse was done, and the Assembly Language uppercase function was called from a C **main** program.

How to package the code as both static and shared libraries as well as how to package the code for consumption was taught. How to call the uppercase function from Python was shown, which is typical of high-level languages with the ability to call shared libraries.

In the next chapter, Chapter 10, "Multiply and Divide," multiplication and division will be added to the programming repertoire.

Exercises

1. Add a macro to **debug.S** to print a string given a register as a parameter that contains a pointer to the string to print.
2. Add a macro to **debug.S** to print a register, if it contains a single ASCII character.
3. In the **printReg** macro, set **t0**–**t6** and **a0**–**a7** to known unusual values before the call to **printf**. Then step through the call to **printf** to see how many of these registers are clobbered.
4. Create a C program to call the lowercase routine from Chapter 6, "Functions and the Stack," Exercise 3 and print out some test cases.

5. Create static and shared library packages for the lowercase routine from Chapter 6, “Functions and the Stack,” Exercise 3.
6. Take the lower-case routine from Chapter 6, “Functions and the Stack,” Exercise 3 and embed it in C code using an **asm** statement.
7. Create a Python program to call the shared library from Exercise 5.
8. Modify **debug.S** to work on the ESP32-C3. The registers are all 32-bit. Then write a test program to show they work.
9. Create an ESP32-C3 project with the main program in C and call the **upper.S** routine, similar to Listing 9-4.
10. Run **objdump -d** on a program that uses **debug.S**. Check the code generated by the macros to ensure the code correctly saves and restores all the registers.

CHAPTER 10

Multiply and Divide

In this chapter, we return to using mathematics. Addition, subtraction, and a collection of bit operations on the CPU registers have already been covered, so now, instructions for multiplication and division will be given in this chapter.

These instructions are part of the RISC-V M extension that adds integer multiplication and division to the standard base integer instructions studied so far. This means that not all RISC-V CPUs contain these instructions. If missing, they can be simulated by writing loops in the base instruction set. All the RISC-V CPUs covered in this book contain the M extension instructions, but these are likely missing from the most inexpensive microcontrollers. All the instructions in the M extension are R-type instructions.

All the instructions covered so far, typically execute in one clock cycle. Multiplication typically takes three to ten clock cycles, while division can take up to thirty clock cycles. These times vary widely depending on the cost and implementation efficiency of the given RISC-V CPU. The key point is that multiplication and especially division are time-consuming operations, and it pays to look for shortcuts, such as using shift operations wherever possible.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_10

Multiplication

The complexity introduced by multiplication is that the product is twice as large as the individual operands. Hence, if two 32-bit integers are multiplied, the product could be 64-bits in size, and similarly multiplying two 64-bit integers could result in a 128-bit result. Some architectures, such as ARM, allow two result registers to hold the product. RISC-V uses two instructions to produce the result, with rules so that only one multiplication takes place.

The rules for signed versus unsigned multiplications are different, so unlike addition, there are both signed and unsigned versions of the instructions that calculate the high order bits.

The basic multiply instructions are

```
mul  rd, rs1, rs2   # signed multiplication
mulw rd, rs1, rs2   # 32-bit mult (64-bit CPUs only)
```

Here are some notes on this instruction:

- **rd** is the lower 64-bits of the product for 64-bit CPUs and the lower 32-bits for 32-bit CPUs.
- All the operands are registers, but immediate operands are not allowed. Remember left shift multiplies by powers of two, such as two, four, and eight.
- There is no unsigned version of this instruction. See Exercise 10-1 for why it is not needed.
- The **mulw** instruction for 32-bit multiplication on a 64-bit CPU only produces a 32-bit product, even though there is room for the full product in the results register.

To receive the higher-order bits of the product, there are

```
mulh      rd, rs1, rs2    # signed multiplication
mulhu     rd, rs1, rs2    # unsigned multiplication
```

with **rd** being the upper 64-bits of the product for 64-bit CPUs and the upper 32-bits for 32-bit CPUs.

It appears that to get the complete product, multiplication needs to be performed twice. However, the RISC-V specification provides some rules, that if followed, then the processor can perform an optimization and only perform the multiplication once. For signed arithmetic, use the following:

```
mulh  rdh, rs1, rs2
mul   rdl, rs1, rs2
```

For unsigned multiplication, use

```
mulhu rdh, rs1, rs2
mul   rdl, rs1, rs2
```

If this sequence is performed, then only one multiplication will be performed, and the second instruction will take one clock cycle. There are four rules that need to be adhered to

1. The **mulh** or **mulhu** instruction must occur first.
2. The **mul** instruction must be the next instruction.
3. The registers **rs1** and **rs2** must be the same and in the same order in both instructions.
4. The destination register **rdh** cannot be one of **rs1** or **rs2**.

Examples

Listing 10-1 has some code to demonstrate all the various multiply instructions. Use the **debug.S** file from Chapter 9, "Interacting with C and Python," meaning the program must be organized with the C runtime in mind.

Listing 10-1. Examples of the various multiply instructions

```
#
# Examples of Multiplication
#

.include "debug.S"

.global main # Provide program starting address

# Load the registers with some data
# Use small positive numbers that will work for all
# multiply instructions.
main:
        li    x5, 25
        li    x6, 4

        printStr "Inputs:"
        printReg 5
        printReg 6

        mul   x7, x5, x6
        printStr "mul x7=x5*x6:"
        printReg 7

        mulw  x7, x5, x6
        printStr "mulw x7=x5*x6:"
        printReg 7

        ld    x5, A
        ld    x6, B
        printStr "Inputs:"
        printReg 5
        printReg 6
        mulh  x7, x5, x6
```

```
        mul   x28, x5, x6
        printStr "mulh x7 = top 64 bits of x5*x6 (signed):"
        printReg 7
        printStr "mul x28 = bottom 64 bits of x5*x6:"
        printReg 28

        mulhu x7, x5, x6
        mul   x28, x5, x6
        printStr "mulhu x7 = top 64 bits of x5*x6 (unsigned):"
        printReg 7
        printStr "mul x28 = bottom 64 bits of x5*x6:"
        printReg 28

        li    a0, 0       # return code
        ret

.data
.align 16
A:     .dword       0x7812345678
B:     .dword       0xFABCD12345678901
```

The **makefile** is as expected. The output is as follows:

```
user@starfive:~/Chapter10$ make
gcc -o multiply multiply.S
user@starfive:~/Chapter10$ ./multiply
Inputs:
x5 =                                25, 0x0000000000000019
x6 =                                 4, 0x0000000000000004
mul x7=x5*x6:
x7 =                               100, 0x0000000000000064
mulw x7=x5*x6:
x7 =                               100, 0x0000000000000064
```

```
Inputs:
x5 =                        515701495416, 0x0000007812345678
x6 =                 -379198319187490559, 0xfabcd12345678901
mulh x7 = top 64 bits of x5*x6 (signed):
x7 =                        -10600956976, 0xffffffd88223bd0
mul x28 = bottom 64 bits of x5*x6:
x28 =                845536204478549567 2, 0x75577afb36c28e78
mulhu x7 = top 64 bits of x5*x6 (unsigned):
x7 =                        505100538440, 0x000000759a569248
mul x28 = bottom 64 bits of x5*x6:
x28 =                845536204478549567 2, 0x75577afb36c28e78
```

To demonstrate **mulh**/**mul** and **mulhu**/**mul**, large numbers were loaded that overflowed a 64-bit result, so non-zero values in the upper 64-bits were seen. Notice the difference between the signed and unsigned computation.

Multiply is straight forward, now division is explained.

Division

Integer division is part of the M extension. For division, both the quotient and remainder may be required.

The division instructions are as follows:

```
div    rd, rs1, rs2  # signed rd = rs1 / rs2
divu   rd, rs1, rs2  # unsigned rd = rs1 / rs2
rem    rd, rs1, rs2  # signed rd = remainder ( rs1 / rs2 )
remu   rd, rs1, rs2  # unsigned rd = remainder( rs1 / rs2 )
```

where

- **rd**: is the destination register.
- **rs1**: is the register holding the numerator.
- **rs2**: is a register holding the denominator.

Note These instructions aren't the inverses of **mulh**/**mul** or **mulhu**/**mul**. For this, **xs1** needs to be a register pair, so the value to be divided can be 128-bits, or 64-bits for 32-bit CPUs. To perform this division, we need to either go to the optional floating-point processor or create the code.

If both the quotient and remainder are required, then RISC-V supports an optimization, like multiplication, where one division is performed, and the second instruction retrieves the result from the previous division in one clock cycle. To get this, for signed division, use the following code:

```
div   rdq, rs1, rs2
rem   rdr, rs1, rs2
```

For unsigned division, use the following code:

```
divu  rdq, rs1, rs2
remu  rdr, rs1, rs2
```

Follow the four rules:

1. The **div** or **divu** instruction must appear first.
2. The **rem** or **remu** instruction must be the next instruction.
3. The two registers **rs1** and **rs2** must be the same and in the same order.
4. The destination register **rdq** cannot be the same as either **rs1** or **rs2**.

Division by Zero and Overflow

On many CPU architectures, dividing by zero or having an overflow occur results in an exception. This is not the case with RISC-V. The designer of RISC-V felt that the extra complexity in the hardware circuitry required is not worth it, given that checking for zero before dividing is easy. Instead, RISC-V specification defines fixed results in these cases and proceeds as if nothing bad happened. For instance, a division by zero results in -1. All the error cases are included in the example in the next section. There is only one overflow case, namely, if the most negative integer is divided by -1, this is since there is one more negative integer than positive integers, due to zero.

Example

The code to execute the divide instructions is simple. Listing 10-2 is an example as was done for multiplication, name the file **division.S**.

Listing 10-2. Examples of the div, divu, rem, and remu instructions

```
#
# Examples of Integer Division
#

.include "debug.S"

.global main # Provide program starting address

# Load the registers with some data
# Perform various division instructions
main:
        li    x5, 100
        li    x6, 6
```

```
printStr "Inputs:"
printReg 5
printReg 6

div   x7, x5, x6
rem   x28, x5, x6
printStr "x5 / x6 (signed):"
printReg 7
printStr "x5 %% x6 (signed):"
printReg 28

divu  x7, x5, x6
remu  x28, x5, x6
printStr "x5 / x6 (unsigned):"
printReg 7
printStr "x5 %% x6 (unsigned):"
printReg 28

# Signed division by zero
li    x6, 0
div   x7, x5, x6
rem   x28, x5, x6
printStr "Signed division by zero:"
printReg 7
printStr "Signed remainder on division by zero:"
printReg 28

# Unsigned division by zero
li    x6, 0
divu  x7, x5, x6
remu  x28, x5, x6
printStr "Unsigned division by zero:"
printReg 7
```

```
        printStr "Unsigned remainder on division by zero:"
        printReg 28

        # Overflow
        li    x5, -9223372036854775808
        li    x6, -1
        div   x7, x5, x6
        rem   x28, x5, x6
        printStr "Overflow division:"
        printReg 7
        printStr "Overflow remainder:"
        printReg 28

        li    a0, 0        # return code
        ret
```

The **makefile** is as expected. When the program is built and run, this is the result:

```
user@starfive:~/Chapter10$ make
gcc -o division division.S
user@starfive:~/Chapter10$ ./division
Inputs:
x5 =                              100, 0x0000000000000064
x6 =                                6, 0x0000000000000006
x5 / x6 (signed):
x7 =                               16, 0x0000000000000010
x5 % x6 (signed):
x28 =                               4, 0x0000000000000004
x5 / x6 (unsigned):
x7 =                               16, 0x0000000000000010
x5 % x6 (unsigned):
x28 =                               4, 0x0000000000000004
```

```
Signed division by zero:
x7 =                              -1, 0xffffffffffffffff
Signed remainder on division by zero:
x28 =                            100, 0x0000000000000064
Unsigned division by zero:
x7 =                              -1, 0xffffffffffffffff
Unsigned remainder on division by zero:
x28 =                            100, 0x0000000000000064
Overflow division:
x7 =             -9223372036854775808, 0x8000000000000000
Overflow remainder:
x28 =                              0, 0x0000000000000000
```

Next, combining multiplication and addition to perform matrix multiplication is shown.

Example: Matrix Multiplication

As a slightly more sophisticated example, a return to a first-year university Math course on Linear Algebra is required. Most science students are forced to take this course to learn to work with vectors and matrices, then they hope to never see these concepts again. Unfortunately, they form the foundation for both computer graphics and machine learning. Before delving into the program, a review of linear algebra follows.

Vectors and Matrices

A vector is an ordered list of numbers. For instance, in 3D graphics it might represent a location in 3D space where [x, y, z] are the coordinates. Vectors have a dimension which is the number of elements they contain.

It turns out that a useful computation with vectors is something called a dot product. If $A = [a_1, a_2, \ldots, a_n]$ is one vector, and $B = [b_1, b_2, \ldots, b_n]$ is another vector, then their dot product is defined as

$$A \bullet B = a_1 * b_1 + a_2 * b_1 + \ldots + a_n * b_n$$

To calculate this dot product, then a loop performing multiplications and additions needs to be quite efficient.

A matrix is a two-dimensional table of numbers such as

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

Matrix multiplication is a complicated process that drives first year Linear Algebra students nuts. When multiplying matrix A times matrix B, then each element on the resulting matrix is the dot product of a row of matrix A with a column of matrix B.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} = \begin{vmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{vmatrix}$$

If these were 3x3 matrices, then there would be nine dot products each with nine terms. A matrix can also be multiplied by a matrix by a vector the same way.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} b_1 \\ b_2 \end{vmatrix} = \begin{vmatrix} a_{11}b_1+a_{12}b_2 \\ a_{21}b_1+a_{22}b_2 \end{vmatrix}$$

In 3D graphics, if a point is represented as a 4D vector [x, y, z, 1], then the affine transformations of scale, rotate, shear, and reflection can be represented as 4x4 matrices. Any number of these transformations can be

combined into a single matrix. Thus, to transform an object into a scene requires a matrix multiplication applied to each of the object's vertex points. The faster this is done, the faster a frame renders in a video game.

In neural networks, the calculation for each layer of neurons is calculated by a matrix multiplication followed by the application of a nonlinear function. The bulk of the work is matrix multiplication. Most neural networks have many layers of neurons, each requiring matrix multiplication. The matrix size corresponds to the number of variables and the number of neurons; consequently, the matrices' dimensions are often in the thousands. How quickly an object recognition or speech translation performs is dependent on how fast matrices can multiply, and that is dependent upon how fast the dot product multiplication/addition loop performs.

These important applications are why the RISC-V has the proposed V extensions, which allow many of these operations to be performed in parallel. None of the processors covered in this book implement the V extension.

Multiplying 3x3 Integer Matrices

To practice loops, addition and multiplication, multiply two 3x3 matrices. The algorithm being implemented is shown in Listing 10-3.

Listing 10-3. Pseudo-code for our matrix multiplication program

```
FOR row = 1 to 3
        FOR col = 1 to 3
                acum = 0
                FOR i = 1 to 3
                        acum = acum + A[row, i]*B[i, col]
                NEXT I
                C[row, col] = acum
        NEXT col
NEXT row
```

The row and column loops go through each cell of the output matrix and calculate the correct dot product for that cell in the innermost loop. Listing 10-4 shows the implementation in Assembly Language, name the file **matmul.S**.

Listing 10-4. 3x3 Matrix multiplication in Assembly Language

```
#
# Multiply 2 3x3 integer matrices
#
# Registers:
#      t0 - Row index
#      t1 - Column index
#      t2 - Address of row
#      t3 - Address of column
#      t4 - 64 bit accumulated sum
#      t5 - Cell of A
#      t6 - Cell of B
#      s1 - Position in C
#      s2 - Loop counter for printing
#      s3 - row in dotloop
#      s4 - col in dotloop

.global main # Provide program starting address

        .equ   N, 3   # Matrix dimensions
        .equ   WDSIZE, 4 # Size of element
main:
        addi   sp, sp, -48
        sd     ra, 32(sp)  # push the return address to
                             the stack
        sd     s1, 24(sp)
        sd     s2, 16(sp)
```

```
        sd      s3, 8(sp)
        sd      s4, 0(sp)

        li      t0, N       # Row index
        la      t2, A       # Address of current row
        la      s1, C       # Address of results matrix
rowloop:
        la      t3, B       # first column in B
        li      t1, N       # Column index (will count
                              down to 0)

colloop:
        # Zero accumulator registers
        li      t4, 0

        li      a0, N       # dot product loop counter
        mv      s3, t2      # row for dot product
        mv      s4, t3      # column for dot product
dotloop:
        # Do dot product of a row of A with column of B
        lw      t5, 0(s3)   # load A[row, i]
        addi    s3, s3, WDSIZE   # increment row counter
        lw      t6, 0(s4)   # load B[i, col]
        addi    s4, s4, N*WDSIZE # increment col counter
        mul     t5, t5, t6  # A[row, i] * B[i, col]
        add     t4, t4, t5  # Add to sum
        addi    a0, a0, -1  # Dec loop counter
        bnez    a0, dotloop      # If not zero loop

        sw      t4, 0(s1)   # C[row, col] = dotprod
        add     s1, s1, WDSIZE
        add     t3, t3, WDSIZE   # Inc current col
        addi    t1, t1, -1  # Dec col loop counter
        bnez    t1, colloop # If not zero loop
```

```
        add    t2, t2, (N*WDSIZE) # Increment to next row
        addi   t0, t0, -1  # Dec row loop counter
        bnez   t0, rowloop # If not zero loop

# Print out matrix C
# Loop through 3 rows printing 3 cols each time.
        li     s2, N       # Print N rows
        la     s1, C       # Addr of results matrix
printloop:

        la     a0, prtstr  # printf format string
        lw     a1, 0(s1)   # first element in current row
        lw     a2, WDSIZE(s1)  # second element in current row
        lw     a3, 2*WDSIZE(s1)  # third element in current row
        add    s1, s1, WDSIZE * N  # increment N cells
        call   printf      # Call printf
        addi   s2, s2, -1  # Dec loop counter
        bnez   s2, printloop       # If not zero loop

        li     a0, 0       # return code
        ld     ra, 32(sp)  # pop the return address from
                             the stack
        ld     s1, 24(sp)
        ld     s2, 16(sp)
        ld     s3, 8(sp)
        ld     s4, 0(sp)
        addi   sp, sp, 48

        ret

.data
# First matrix
A:      .word  1, 2, 3
        .word  4, 5, 6
        .word  7, 8, 9
```

```
# Second matrix
B:      .word  9, 8, 7
        .word  6, 5, 4
        .word  3, 2, 1
# Result matix
C:      .fill  9, 4, 0

prtstr: .asciz  "%3d  %3d  %3d\n"
```

After compiling and running this program, the result is

```
user@starfive:~/Chapter10$ make
gcc -o matmul matmul.S
user@starfive:~/Chapter10$ ./matmul
 30   24   18
 84   69   54
138  114   90
```

Accessing Matrix Elements

The three matrices in memory are stored in row order. They are arranged in the **.word** directives so that the matrix structure is shown. In the pseudo-code, the matrix elements are referred to using two-dimensional arrays. There are no instructions or operand formats to specify two-dimensional array access, so it must be created. To Assembly Language, each array is just a nine-word sequence of memory. To translate the one-dimensional array to two dimensions, the following could be done as follows:

```
A[i, j] = A[i*N + j]
```

where N is the dimension of the array. However, do not do this, as in Assembly Language it pays to notice that the array elements are accessed in order and can go from one element in a row to the next by adding the size of an element—the size of a word, or four bytes. An element in a

column can go to the next one by adding the size of a row. Therefore, the constant N * WDSIZE is used often in the code. This way the array can be gone through incrementally and never have to multiply array indexes. Generally, multiplication and division are expensive operations and should be avoided as much as possible.

Register Usage

Quite a few registers are used, so luckily all the loop indexes and pointers can be placed in registers, without having to move them in and out of memory. If this was needed, space would have to be allocated on the stack to hold any needed variables.

Notice that registers **s1** and **s2** are used in the loop that does the printing. That is because the **printf** function will change any of the registers **t0**–**t6**. Registers **t0**–**t6** are chiefly used otherwise since these do not need to be preserved for the caller. However, **s1**–**s4** do need to be preserved, so these are pushed and popped to and from the stack along with **ra**.

Summary

Various forms of the multiply and division instructions supported in the RISC-V instruction set were introduced.

Variations of these instructions were reviewed and then an example matrix multiplication program was presented to show them in action. Matrix multiplication is heavily used in computer graphics and machine learning applications.

In Chapter 11, “Floating-Point Operations,” more math will be shown, but in scientific notation allowing fractions and exponents, going beyond integers for the first time.

Exercises

1. To multiply two 64-bit numbers resulting in a 128-bit product, the **mul** instruction was used to obtain the lower 64-bits of the product for both the signed and unsigned integer cases. To prove that this works, work on a small example:
   - Multiply two 4-bit numbers to get an 8-bit product.
   - Multiply 0xf by 2. In this signed case, 0xf is −1 and the product is −2, in the unsigned case 0xf is 15 and the product is 30.
   - Manually perform the calculation to ensure the correct result is obtained in both cases.
2. Write a signed 64-bit integer division routine that checks if the denominator is zero before performing the division. Print an error if zero is encountered.
3. Write a routine to compute a dot product of dimension six. Put the numbers to calculate in the **.data** section and print the result.
4. Change the matrices calculated in the example and check that the result is correct.
5. Create an ESP32-C3 project for matrix multiplication. Most of the code remains unchanged since the matrices are already 32-bit integers. The size of the registers pushed/popped to/from the stack needs to be adjusted and **main** changed to **app_main**.

CHAPTER 11

Floating-Point Operations

In this chapter, the RISC-V floating-point extensions will be studied. The floating-point extensions provide useful instructions for working with numbers in scientific notation, which add a decimal point along with exponents. Floating-point numbers will be defined, as well as how they are represented in memory and how to insert them into Assembly Language programs. How to transfer data between the floating-point registers and the regular RISC-V integer registers and memory is explained. Also, how to perform basic arithmetic operations, comparisons, and conversions is presented.

Floating point support is part of the F, D, Q, and V extensions. Floating-point support also requires the **Zicsr** extension which adds control and status registers. The ESP32-C3 does not support any of these; therefore, it has no native support for floating-point arithmetic, although the **GCC** library does contain an integer-based implementation of most floating point operations. To use the instructions in this chapter, a Starfive Visionfive 2 or the QEMU emulator is required. The Starfive Visionfive 2 supports the F and D extensions, but not the Q or V extension.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_11

About Floating Point Numbers

Floating point numbers are a way to represent numbers in scientific notation on the computer and are formatted as follows:

$$1.456354 \times 10^{16}$$

There is a fractional part and an exponent that allows movement of the decimal place to the left if it is positive and to the right if it is negative. The RISC-V CPU extension functions are as follows:

- **F**: Deals with single precision floating-point numbers that are 32-bits in size.
- **D**: Supports double precision floating-point numbers that are 64-bits in size.
- **Q**: Supports quad precision floating-point numbers that are 128-bits in size.
- **V**: Adds half precision 16-bit floating-point numbers along with vector operations.

The RISC-V CPU uses the IEEE 754 standard for floating point numbers. Each number contains a sign bit to indicate if it is positive (+) or negative (-), a field of bits for the exponent and a string of digits for the fractional part. Table 11-1 lists the number of bits for the parts of each format, along with which RISC-V extension supports it.

Table 11-1. *Bits of a Floating-Point Number*

Name	Extension	Precision	Sign	Fractional	Exponent	Decimal Digits
Half	V	16-bits	1	10	5	3
Single	F	32-bits	1	23	8	7
Double	D	64-bits	1	52	11	16
Quad	Q	128-bits	1	113	14	34

The decimal digits column of Table 11-1 is the approximate number of decimal digits that the format can represent, or the decimal precision.

There are special representations for positive and negative infinity ($\pm\infty$). If a computation results in a larger number than can be represented, the result becomes infinity. Single-precision positive infinity is represented by 0x7F800000.

About Normalization and NaNs

In the integers seen so far, all combinations of the bits provide a valid unique number. No two different patterns of bits produce the same number; however, this is not the case in floating-point. First is the concept of Not a Number (**NaN**), that are produced from illegal operations like dividing by zero or taking the square root of a negative number. These allow the error to quietly propagate through the calculation without crashing a program.

In the IEEE 754 specification, a NaN is represented by an exponent of all one bits, for example, 11111, depending on the size of the exponent, and all other bits are zero. The following operations will result in a NaN:

$$\infty - \infty, -\infty + \infty, 0 \times \infty, 0 \div 0, \infty \div \infty$$

The IEEE 754 specification defines two types of NaNs: a quiet NaN and a signaling NaN; however, the RISC-V instructions only produce quiet NaNs, as RISC-V does not throw (signal) exceptions because of computations.

A normalized floating-point number means the first digit in the fractional part is nonzero. A problem with floating point numbers is that numbers can often be represented in multiple ways. For instance, a fractional part of 0 with either sign bit and any exponent is zero. Consider a representation of 1:

$$1E0 = 0.1E1 = 0.01E2 = 0.001E3$$

All of these represent 1, but the first one can be called with no leading zeros in the normalized form. The RISC-V CPU tries to keep floating-point numbers in normal form but will break this rule for small numbers, where the exponent is already as negative as it can go, then to try to avoid underflow errors, the floating-point extensions will give up on normalization to represent numbers a bit smaller than it could otherwise.

Recognizing Rounding Errors

If a number like $1/3 = 0.33333...$, is represented in floating point, then only seven or so digits are kept for single precision. This introduces rounding errors. If these are a problem, usually going to double precision solves the problems, but some calculations are prone to magnifying rounding errors, such as subtracting two numbers that have a minute difference.

Note Floating point numbers are represented in base two, so the decimal expansions leading to repeating patterns of digits are different from that of base ten. It can be a surprise that 0.1 is a repeating binary fraction: 0.00011001100110011… Meaning that

adding dollars and cents in floating point will introduce a rounding error. For financial calculations, most applications use fixed point arithmetic that is built on integer arithmetic to avoid rounding errors in addition and subtraction.

Defining Floating Point Numbers

The GNU Assembler has directives for defining storage for both single and double precision floating point numbers, for example, **.single** and **.double**:

```
.single      1.343, 4.343e20, -0.4343, -0.4444e-10
.double      -4.24322322332e-10, 3.141592653589793
```

These directives always take base 10 numbers.

Note The GNU Assembler does not have a directive for 16-bit half-precision floating point numbers, so they must be loaded one of these and then do a conversion. Similarly, there is no support for 128-bit floating point numbers.

About Floating Point Registers

There are 32 floating point registers referred to as **f0**, …, **f31**. The size of these registers is dependent on which extensions the RISC-V CPU supports. If the CPU only supports the F extension, then they are 32-bits in size, if the CPU supports the D extension then they are 64-bits and if the CPU supports the Q extension then they are 128-bits in size.

Note The register **f0** is a regular register and not a special zero register like **x0**.

The Status and Control Register

The reason the floating-point extensions require the **Zicsr** extension is for a floating-point control and status register **fcsr**. This lets the default rounding mode be set and allows checking for exceptions. The exceptions accumulate and need to be manually cleared. Like integer division, there are no interrupts and it is up to the programmer to either check the inputs to a calculation thoroughly and/or check the exception flags after the calculation. Figure 11-1 shows which bits of the **fcsr** control which functions.

Bits	31-8	7-5	4-0
Function	Reserved	Rounding Mode (frm)	Accrued exceptions (fflags)
			NV DZ OF UF NX

Figure 11-1. *Format of the bits in the 32-bit floating point control and status register (**fcsr**)*

For the rounding mode, a standard rounding mode can be selected, or it can be set to **dyn** and the rounding mode must be specified with each instruction. Table 11-2 lists the supported rounding modes along with their value and mnemonic. The standard rounding mode is **rne**.

Table 11-2. *Rounding Modes and Their Encodings*

Rounding Mode	Mnemonic	Description
000	rne	Round to nearest, ties to even
001	rtz	Round towards zero
010	rdn	Round down (towards $-\infty$)
011	rup	Round up (towards $+\infty$)
100	rmm	Round to nearest, ties to max magnitude
101		Invalid. Reserved for future use.
110		Invalid. Reserved for future use.
111	dyn	Dynamic rounding mode, use the **rm** field in each instruction.

Each exception type has a bit in the **fcsr** and the values are shown in Table 11-3.

Table 11-3. *Exception Mnemonics and Their Meaning*

Flag Mnemonic	Description
nv	Invalid operation
dz	Divide by zero
of	Overflow
uf	Underflow
nx	Inexact

These mnemonics are not defined in **gcc**, so the **.equ** directive needs to be used to define them. The fields in the **fcsr** can be read and set using the instructions added via the RISC-V **Zicsr** extension. However, there are a set of pseudoinstructions for accessing the fields in the **fscr** which are presented here.

```
frcsr       rd      # read the fcsr into rd
fscsr       rd, rs  # rd = old value fcsr, fcsr = rs
fscsr       rs      # write fcsr = rs
frrm        rd      # read rounding mode into rd
fsrm        rd, rs  # swap rounding mode with rs
fsrm        rs      # write rounding mode = rs
fsrmi       rd, imm # swap rounding mode with immediate
fsrmi       imm     # write rounding mode with immediate
frflags     rd      # read exception flags
fsflags     rd, rs  # swap exception flags
fsflags     rs      # write exception flags
fsflagsi    rd, imm # swap exception flags with immediate
fsflagsi    imm     # write exception flags with immediate
```

In Listing 11-1, the routine to calculate the distance between two points shows how to set the rounding mode and monitor operations for exceptions.

Notice how it restores the rounding mode to the previous value before returning. This avoids unexpected side effects affecting the calling program.

Defining the Function Call Protocol

In Chapter 6, “Functions and the Stack,” the protocol for the calling function or function that saves which registers was given. With these floating-point registers, they must be added to the protocol. Like the integer registers, the floating-point registers have ABI aliases to help make code more readable.

- **Callee saved**: The function is responsible for saving registers **f8–f9** and **f18–f27**. They need to be saved by a function if the function uses them. Their ABI aliases are **fs0–fs11**.
- **Caller saved**: All other registers do not need to be saved by a function, so they must be saved by the caller if they are required to be preserved. This includes **f10–f17** which are used to pass parameters and have aliases **fa0–fa7**. **f0–f7** and **f28–f31** have aliases **ft0–ft11**.

Loading and Saving FPU Registers

In Chapter 5, "Thanks for the Memories," load and store instructions to load registers from memory, then store them back to memory, were given. The floating-point registers have a matching set of instructions, for example,

```
        la    x5, fp1      # load address of fp1
        flw   f0, 0(x5)    # load single precision fp1
        fld   f1, 4(x5)    # load double precision fp2
        fsw   f0, 12(x5)   # store single precision fp3
        fsd   f1, 16(x5)   # store double precision fp4
        ...
.data
fp1: .single  3.14159
fp2: .double  4.3341
fp3: .single  0.0
fp4: .double  0.0
```

Data can also be moved between the CPU's integer registers and the floating-point registers with the **fmv** instruction, for example,

```
fmv.x.w     f0, x5  # move 32-bits of x5 to f0
fmv.w.x     x5, f0  # move 32-bits of f0 to x5
fmv.x.d     f0, x5  # move 64-bits of x5 to f0
fmv.d.x     x5, f0  # move 64-bits of f0 to x5
```

Note The **fmv** instruction copies the bits unmodified. It does not perform any sort of conversion.

To move data between floating-point registers, there is an **fmv** pseudoinstruction based on the sign injection **fsgnj** instruction.

```
fmv.s       f0, f1  # f0 = f1 single precision
fmv.d       f0, f2  # f0 = f1 double precision
```

Performing Basic Arithmetic

The floating-point instructions include the four basic Arithmetic operations, along with a few extensions like multiply and accumulate. There are some specialty functions like square root and min/max.

Each of these functions can operate on either **h**, **s**, **d**, or **q** values. Here is a selection of the instructions. The four forms of the **fadd** instruction with each floating-point type are listed, then the rest are listed with just the **d** versions to save space:

- fadd.h fd, fn, fm # fd = fn + fm half precision
- fadd.s fd, fn, fm # fd = fn + fm single precision

- fadd.d fd, fn, fm # fd = fn + fm double precision
- fadd.q fd, fn, fm # fd = fn + fm quad precision
- fsub.d fd, fn, fm # fd = fn - fm
- fmul.d fd, fn, fm # fd = fn * fm
- fdiv.d fd, fn, fm # fd = fn / fm
- fmadd.d fd, fn, fm, fa # fd = fa + fm * fn
- fmsub.d fd, fn, fm, fa # fd = fm *fn - fa
- fnmsub.d fd, fn, fm, fa # fd = fa - fm *fn
- fnmadd.d fd, fn, fm, fa # fd = -fa - fm * fn
- fmax.d fd, fn, fm # fd = Max(fn, fm)
- fmin.d fd, fn, fm # fd = Min(fn, fm)
- fsqrt.d fd, fn # fd = Square Root(fn)

Plus, there are a couple of useful pseudo instructions:

- fneg.d fd, fn # fd = - fn
- fabs.d fd, fn # fd = Absolute Value(fn)

Each of these instructions can also have a rounding mode modifier to specify the rounding mode used in the computation.

```
.equ        rup, 3
fadd.s      f0, f1, f2, rup  # add with
                               rounding mode up
```

These functions are all simple, so let us move on to an example using floating-point functions.

Calculating Distance Between Points

Given two points (x_1, y_1) and (x_2, y_2), the distance between them is determined by the following formula:

$$d = \text{sqrt}((y_2-y_1)^2 + (x_2-x_1)^2)$$

Write a function to calculate this for any two single precision floating point pair of coordinates. Use the C runtime's **printf** function to print out the results. First of all, copy the distance function from Listing 11-1 to the file **distance.S**.

Listing 11-1. Function to calculate the distance between two points

```
#
# Example function to calculate the distance
# between two points in double precision
# floating point.
#
# Inputs:
#      a0 - pointer to the 4 FP numbers
#             they are x1, y1, x2, y2
# Outputs:
#      a0 - the length (as double precision FP)

.global distance # Allow function to be called by others

        .equ  rne, 0
#
distance:
        # save return address incase printf is called for error
        addi   sp, sp, -16  # allocate 16 bytes on stack
        sd     ra, 0(sp)    # push return address
```

```
        # save and set the fcsr
        frcsr  t0    # t0 = original fcsr
        fsrmi  rne   # set rounding mode
        fsflagsi 0   # clear flags

        # load the 4 numbers
        fld    ft0, 0(a0)
        fld    ft1, 8(a0)
        fld    ft2, 16(a0)
        fld    ft3, 24(a0)

        # calc ft4 = x2 - x1
        fsub.d ft4, ft2, ft0
        # calc ft5 = y2 - y1
        fsub.d ft5, ft3, ft1
        # calc ft4 = ft4 * ft4 (x2-X1)^2
        fmul.d ft4, ft4, ft4
        # calc ft5 = ft5 * ft5 (Y2-Y1)^2
        fmul.d ft5, ft5, ft5
        # calc ft4 = ft4 + ft5
        fadd.d ft4, ft4, ft5
        # calc sqrt(ft4)
        fsqrt.d ft4, ft4

        # did a floating point error occur?
        frflags    t1     # read exceptions flags
        beqz   t1, nerror # no error then go to exit
        la     a0, errormsg
        mv     a1, t1
        call   printf
nerror:
        # move result to fa0 to be returned
        fmv.x.d   a0, ft4
```

```
        fscsr  t0          # restore fcsr
        ld     ra, 0(sp)   # pop ra
        addi   sp, sp, 16  # deallocate stack space
        ret

.data
errormsg: .asciz "Floating-point error = %x\n"
```

Place the code from Listing 11-2 in **main.S** that calls distance three times with three different points and prints out the distance for each one.

Listing 11-2. Main program to call the distance function three times

```
#
# Main program to test our distance function
#
# s1 - loop counter
# s0 - address to current set of points

.global main # Provide program starting address to linker

#

        .equ   N, 4   # Number of points.

main:
        addi   sp, sp, -32  # allocate 32 bytes on stack
        sd     ra, 16(sp)   # push return address
        sd     s0, 8(sp)    # push s0 register
        sd     s1, 0(sp)    # push s1 register

        la     s0, points   # pointer to current points

        li     s1, N        # number of loop iterations

loop:   mv     a0, s0       # move pointer to parameter 1 (a0)

        call   distance     # call distance function
```

```
        mv      a1, a0        # return double to a1 to print
        la      a0, prtstr    # load print string
        call    printf        # print the distance

        addi    s0, s0, (4*8)   # 4 points each 8 bytes
        addi    s1, s1, -1    # decrement loop counter
        bnez    s1, loop      # loop if more points

        li      a0, 0         # return code

        ld      s1, 0(sp)     # pop s1
        ld      s0, 8(sp)     # pop s0
        ld      ra, 16(sp)    # pop ra
        addi    sp, sp, 32    # deallocate stack space
        ret

.data
points:         .double     0.0, 0.0, 3.0, 4.0
                .double     1.3, 5.4, 3.1, -1.5
                .double     1.323e10, -1.2e-4, 34.55, 5454.234
                .double     9.42e250, 4.44e120, 4.4, 8.8
prtstr:         .asciz "Distance = %f\n"
```

The **makefile** is in Listing 11-3.

Listing 11-3. Makefile for the distance program

```
distance: main.S distance.S
        gcc -o distance main.S distance.S
```

If the program is built and run, the result is as follows:

```
user@starfive:~/Chapter11$ make
gcc -o distance main.S distance.S
user@starfive:~/Chapter11$ ./distance
Distance = 5.000000
```

```
Floating-point error = 1
Distance = 7.130919
Floating-point error = 1
Distance = 13229999965.451126
Floating-point error = 5
Distance = inf
```

The data was constructed so the first set of points comprise a 3–4–5 triangle, that is why the exact answer of **5** is the result for the first distance. Notice that all the other calculations print a floating-point error, where this is one, which means the calculation is inexact as is expected from most square root calculations.

This distance function saves the **fcsr**, sets a known rounding method, and clears the error flags. Near the end, it checks the error flags, prints out a message if any are set, and then restores the **fcsr** before returning. The last calculation is set up to generate an overflow error and the result ends up being infinity.

The distance function is straight forward. It loads the four numbers in four **fld** instructions, then calls the various floating-point arithmetic functions to perform the calculation. This function operates on double-precision 64-bit floating-point numbers using the **.d** versions of the instructions.

The part of the main routine that loops and calls the distance routine is straight forward.

When **printf** is called, the first parameter, the **printf** format string, goes in **a0**, then the next parameter, the double to print goes in **a1**. This is why the result is moved from a floating-point register to the integer return register **a0** before returning.

Note **printf** only supports printing double-precision floating-point numbers, so to print other formats requires a conversion which is covered in the next section.

If debugging the program with **gdb**, to see the contents of the FPU registers at any point, use the "**info all-registers**" command, which will exhaustively list all the registers.

Performing Floating-Point Conversions

To convert between floating-point numbers and integers, there is the **fcvt** instruction. To specify the format of the floating-point number, the same abbreviations are used as in the computation instructions. If the CPU supports multiple floating-point formats, then this instruction supports converting between them. To specify the integer format, use a value from Table 11-4.

Table 11-4. *Abbreviations of Integer Type for the* ***fcvt*** *Instruction*

Mnemonic	Description
w	32-bit signed word
wu	32-bit unsigned word
l	64-bit signed long integer
lu	64-bit unsigned long integer

Also, like the computational instructions, any **fcvt** can take an optional rounding mode operand. A couple of examples of the **fcvt** function are as follows:

```
.equ          rup, 3
fcvt.s.w      f0, x5        # f0 = float(x5)
fcvt.lu.d     x5, f7, rup   # x5 = int(f7), rounding mode up
fcvt.d.s      f4, f5        # f4 (double) = f5 (single)
fcvt.s.d      f4, f5        # f4 (single) = f5 (double)
fcvt.s.w      f5, x0        # f5 = 0.0 single precision
fcvt.d.w      f5, x0        # f5 = 0.0 double precision
```

Notice the last two examples which provide a useful method to initialize floating-point registers to floating-point zero.

Conversions between integers and floating-point numbers have a lot of room for errors. If the conversion cannot be performed, then the nearest value is provided and the **nv** flag is set in the control and status register **fcsr**.

Floating-Point Sign Injection

The **fsgnj** instruction is the basis for several useful pseudoinstructions. It allows the sign of one floating-point number to be set to another, the reverse or an **xor** of both.

```
fsgnj.s     f5, f6, f7    # x5 = abs(x6) with sign x7
fsgnjn.d    f5, f6, f7    # x5 = abs(x6) with sign -x7
fsgnjx.d    f5, f6, f7    # x5 = abs(x6) with sign(x6)
                          # xor sign(x7)
```

This function is then the basis for the **fmv**, **fneg**, and **fabs** pseudoinstructions, for instance,

```
fmv.s  rd, rs             # translates to fsgnj.s
                            rd, rs, rs
```

Comparing Floating-Point Numbers

There are three instructions to compare floating-point numbers. Here are its forms:

```
feq.s  x5, f4, f5   # if f4 = f5 then x5 = 1 else x5 = 0
flt.d  x5, f4, f5   # if f4 < f5 then x5 = 1 else x5 = 0
fle.d  x5, f4, f5   # if f4 <= f5 then x5 = 1 else x5 = 0
```

If either operand is a NaN value, then will return zero and set the invalid operation (**nv**) flag in the **fscr**. To test if a floating-point number is valid, there is the **fclass** instruction that will classify a floating-point number, or instance:

```
        fclass.d  x5, f5 # classify double f5 putting
result in x5
```

The possible classification results are listed in Table 11-5.

Table 11-5. *Floating-Point Classifications Returned via the* ***fclass*** *Instruction*

Classification	Description
0	Negative infinity (−∞)
1	A negative normal number
2	A negative subnormal number
3	Negative zero (-0)
4	Positive zero (+0)
5	A positive subnormal number
6	A positive normal number
7	Positive infinity (+∞)
8	A signaling NaN
9	A quiet NaN

Testing for equality of floating-point numbers is problematic, due to rounding error numbers are often close, but not exactly equal. The solution is to decide on a tolerance, then consider numbers equal if they are within the tolerance from each other. For instance, the tolerance **e** may be defined as equal to 0.000001, then consider two registers equal if:

abs(S1 - S2) < e

where **abs**() is a function to calculate the absolute value.

Example

Create a routine to test if two floating-point numbers are equal using this technique. First, add 100 cents, then test if they exactly equal $1.00 (spoiler alert, they will not). Secondly, compare the sum using the **fpcomp** routine that tests them within a supplied tolerance, usually referred to as epsilon.

Start with the floating-point comparison routine, placing the contents of Listing 11-4 into **fpcomp.S**.

Listing 11-4. Routine to compare two floating point numbers within a tolerance

```
#
# Function to compare two floating point numbers
# the parameters are a pointer to the two numbers
# and an error epsilon.
#
# Inputs:
#       a0 - pointer to the 3 FP numbers
#             they are x1, x2, e
# Outputs:
#       a0 - 1 if they are equal, else 0

.global fpcomp # Allow function to be called by others
```

```
fpcomp:         # load the 3 numbers
        flw     ft0, 0(a0)
        flw     ft1, 4(a0)
        flw     ft2, 8(a0)

        # calc ft3 = ft2 - ft1
        fsub.s ft3, ft1, ft0
        fabs.s ft3, ft3
        fle.s  t0, ft3, ft2
        beqz   t0, notequal
        li     a0, 1
        j      done

notequal:li     a0, 0

done:  ret
```

The main program **maincomp.S** contains Listing 11-5.

Listing 11-5. Main program to add up 100 cents and compare to $1.00

```
#
# Main program to test our distance function
#
# s0 W19 - loop counter
# s1 X20 - address to current set of points

.global main # Provide program starting address

        .equ   N, 100 # Number of additions.

main:
        addi   sp, sp, -32  # allocate 32 bytes on stack
        sd     ra, 16(sp)   # push return address
        sd     s0, 8(sp)    # push s0 register
        sd     s1, 0(sp)    # push s1 register
```

```
# Add up one hundred cents and test if they equal $1.00

        li     s0, N        # number of loop iterations

# load cents, running sum and real sum to FPU
        la     a0, cent
        flw    fs0, 0(a0)
        flw    fs1, 4(a0)
        flw    fs2, 8(a0)
loop:
        # add cent to running sum
        fadd.s fs1, fs1, fs0
        addi   s0, s0, -1   # decrement loop counter
        bnez   s0, loop     # loop if more points

        # store computed sum
brk1:   fsw    fs1, 4(a0)
        # compare running sum to real sum
        feq.s  a1, fs1, fs2
        # print if the numbers are equal or not
        bnez   a1, equal
        la     a0, notequalstr
        call   printf
        j      next
equal:  la     a0, equalstr
        call   printf

next:
# load pointer to running sum, real sum and epsilon
        la     a0, runsum

# call comparison function
        call   fpcomp       # call comparison function
# compare return code to 1 and print if the numbers
```

```
# are equal or not (within epsilon).
        bnez   a0, equal2
        la     a0, notequalstr
        call   printf
        j      done
equal2: la     a0, equalstr
        call   printf

done:   li     a0, 0        # return code
        ld     s1, 0(sp)    # pop s1
        ld     s0, 8(sp)    # pop s0
        ld     ra, 16(sp)   # pop ra
        addi   sp, sp, 32   # deallocate stack space
        ret

.data
cent:          .single 0.01
runsum:        .single 0.0
sum:           .single 1.00
epsilon:       .single 0.00001
equalstr:      .asciz "equal\n"
notequalstr:   .asciz "not equal\n"
```

The **makefile**, in *Listing 11-6*, is as expected.

Listing 11-6. The makefile for the floating-point comparison example

```
fpcomp: fpcomp.S maincomp.S
        gcc -g -o fpcomp fpcomp.S maincomp.S
```

If the program is built and run the program, the following is the result:

```
user@starfive:~/Chapter11$ make -B
gcc -o distance main.S distance.S
gcc -g -o fpcomp fpcomp.S maincomp.S
user@starfive:~/Chapter11$ ./fpcomp
not equal
equal
```

If the program is run under **gdb**, set a breakpoint to done and then **info all-registers**, the sum of 100 cents can be examined and will look as follows:

```
fs0            {float = 0.00999999978,
  double = -nan(0xfffff3c23d70a)} (raw 0xffffffff3c23d70a)
fs1            {float = 0.999999344,
  double = -nan(0xfffff3f7ffff5)} (raw 0xffffffff3f7ffff5)
fs2            {float = 1,
  double = -nan(0xfffff3f800000)} (raw 0xffffffff3f800000)
```

The **fs0** contains a cent, $0.01, and it can be seen from **gdb** that this has not been represented exactly and this is where rounding error will come in. The sum of 100 cents ends up being in register **fs1** as 0.999999344 that does not equal the expected sum of 1 contained in register **fs2**. **gdb** does not know whether the number contained in the register is single or double precision, so it prints out both along with the raw bits.

Then the **fpcomp** routine is called, which determines if the numbers are within the provided tolerance and hence considers them equal.

It did not take that many additions to start introducing rounding errors into the sums, so be careful when using floating-point for this reason.

Summary

In this chapter, the following was learned:

- What floating-point numbers are and how they are represented.
- Normalization, NaNs, and rounding errors.
- How to create floating point numbers in the **.data** section.
- The bank of floating-point registers and how half-, single-, double-, and quad-precision values are contained in them.
- How to load data into the floating-point registers.
- How to perform mathematical operations and save them back to memory.
- How to convert between floating-point types and integers and compare floating-point numbers, and the effect rounding errors have on these comparisons.

In Chapter 12, "Optimizing Code," several techniques to write highly efficient code will be shown.

Exercises

1. Create a program to load and add the following numbers:

   2.343 + 5.3

   3.5343425445 + 1.534443455

   3.14e12 + 5.55e-10

   How accurate are the results?

2. Integer division by 0 resulted in the incorrect answer of -1. Create a program to perform a floating-point division by 0 and see what the result is.

3. Create a program to generate each of the floating-point exceptions found in the status and control register.

4. The RISC-V floating-point instructions have a square root function, but no trigonometric functions. Write a function to calculate the sine of an angle in radians using the approximate formula:

   sin x = x – x3/3! + x5/5! – x7/7!

   where ! stands for factorial and is calculated as: 3! = 3 * 2 *1. Write a main program to call this function with several test values.

CHAPTER 12

Optimizing Code

In this chapter, how to make the uppercase routine more efficient is discussed. This includes design patterns for more efficient conditional statements. Optimizing code often involves thinking outside the box and going beyond finding ways to remove one or two instructions in a loop, often involving a major algorithmic change. Then several further common optimization techniques are listed, but first, a trick to simplify the main **if** statement will be given.

Optimizing the Uppercase Routine

The original uppercase routine implements the pseudocode

```
IF (t2 >= 'a') AND (t2 <= 'z') THEN
        t2 = t2 - ('a'-'A')
END IF
```

with the following Assembly Language code:

```
# If t2 > 'z' then goto cont
        li   t3, 'z'       # load 'z' for comparison
        bgt  t2, t3, cont  # branch if letter > 'z'?
# Else if t2 < 'a' then goto end if
        li   t3, 'a'       # load 'a' for comparison
        blt  t2, t3, cont  # goto to end if not lowercase
```


S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_12

```
# if we got here then the letter is lower case, so convert it.
        addi t2, t2, ('A'-'a')
cont:   # end if
```

This code implements the reverse logic of branching around the **addi** instruction if **t2**< 'a' or **t2**> 'z'. This was fine for a chapter on teaching branch instructions since it demonstrated two of them. However, in this chapter, the goal is to eliminate branches entirely; therefore, instructions are on how to improve this code are presented one step at a time.

Simplifying the Range Comparison

A common way to simplify range comparisons is to shift the range, so a lower bound comparison is not needed. If '**a**' is subtracted from everything, then the pseudocode becomes

```
t4 = t2 - 'a'
IF (t4 >= 0) AND t4 <= ('z'-'a') THEN
        t2 = t2 - ('a'-'A')
END IF
```

If **t4** is treated as an unsigned integer, then the first comparison does nothing, since all unsigned integers are greater than zero. In this case, we simplified our range from two comparisons to one comparison that is **t4** <= ('z'-'a'). To understand why two registers are used here, see Exercise 12-1.

This leads to the first improved version of our **toupper** function. This new **upper2.S** is shown in Listing 12-1.

Listing 12-1. Uppercase routine with simplified range comparison

```
#
# Assembly Language function to convert a string to
# all upper case.
#
```

```
# a1 - address of output string
# a0 - address of input string
# t2 - current character being processed
# t3 - temp register for comparisons
# t4 - minus 'a' to compare < 26.
# t5 - original output string for length calc.
#

.global toupper            # Allow other files to call
                             this routine


toupper:
        mv    t5, a1       # save original outstr for len calc
# The loop is until null (zero) character is encountered.
loop:   lb    t2, 0(a0)    # load character
        addi  a0, a0, 1    # increment buffer pointer
# Want to know if 'a' <= W5 <= 'z'
# First subtract 'a'
        addi  t4, t2, -'a'
# Else if x7 < 26 then goto end if
        li    t3, 26       # load 'a' for comparison
        bltu  t3, t4, cont # goto to end if not lowercase
# if we got here then the letter is lower case, so convert it.
        addi  t2, t2, ('A'-'a')
cont:   # end if
        sb    t2, 0(a1)    # store character to output str
        addi  a1, a1, 1    # increment buffer for next char
        li    t3, 0        # load 0 char for comparison
        bne   t2, t3, loop # loop if character isn't null
```

```
# Setup the parameters to print our hex number
# and then call Linux to do it.
        sub   a0, a1, t5   # get the len by sub'ing the
pointers

        ret                # Return to caller
```

This example uses the same **main.S** from Listing 6-3. Listing 12-2 is a **makefile** for all the code in this chapter. Comment out any programs that have not been reached yet, or a compile error will occur.

Listing 12-2. **`Makefile`** for the uppercase routine version in this chapter

```
UPPER2OBJS = main.o upper2.o
UPPER3OBJS = upper3.o
ifdef DEBUG
DEBUGFLGS = -g
else
DEBUGFLGS =
endif
LSTFLGS =

all: upper2 upper3

%.o : %.S
        as $(DEBUGFLGS) -mno-relax $(LSTFLGS) $< -o $@

upper2: $(UPPER2OBJS)
        ld -o upper2 $(UPPER2OBJS)

upper3: $(UPPER3OBJS)
        ld -o upper3 $(UPPER3OBJS)
```

This is an improvement and a great optimization to use when range comparisons are needed. Next, another branch is removed by restricting the problem domain.

Restricting the Problem Domain

The best optimizations of code arise from restricting the problem domain. If only alphabetic characters are needed, the range comparison can be removed entirely. In Appendix D, "ASCII Character Set," the only difference between upper- and lowercase letters is that lowercase letters have the 0x20 bit set, whereas uppercase letters do not. This means it is possible to convert a lowercase letter to an uppercase one by performing an **and** operation to clear that one bit. If this is done to special characters, it will corrupt the bits of quite a few characters.

Often in computing, text is treated as case-insensitive, meaning that text can be entered in any combination of case. The Assembler does this, so it doesn't care if **MV** or **mv** is entered. Similarly, many computer languages are case-insensitive, so variable names can be entered in any combination of upper and lowercase and it means the same thing. Machine learning algorithms that process text always convert the text into a standard form, usually throwing away all punctuation and converting all the text to one case. Forcing this standardization saves a lot of extra processing later—an implementation of this is shown in Listing 12-3 that goes in **upper3.S**.

Listing 12-3. Uppercase routine as a macro, using **and** for alphabetic characters only

```
#
# Assembly Language function to convert a string to
# all upper case.
#
```

```
# a1 - address of output string
# a0 - address of input string
# t2 - current character being processed
# t5 - original output string for length calc.
#

.global _start             # Entry point for linker.

.MACRO toupper inputstr, outputstr
        la    a0, \inputstr
        la    a1, \outputstr
        mv    t5, a1        # save original outstr for len calc
# The loop is until null (zero) character is encountered.
loop:   lb    t2, 0(a0)     # load character
        addi  a0, a0, 1     # increment buffer pointer
        andi  t2, t2, 0xdf # clear the bit at 0x20
        sb    t2, 0(a1)     # store character to output str
        addi  a1, a1, 1     # increment buffer for next char
        li    t3, 0         # load 0 char for comparison
        bne   t2, t3, loop # loop if character isn't null

# Setup the parameters to print our hex number
# and then call Linux to do it.
        sub   a0, a1, t5    # get the len by sub'ing the
pointers
.ENDM

_start:
        toupper     instr, outstr

# Setup the parameters to print the resulting string
# and then call Linux to do it.
        mv    a2, a0        # return code is the length
```

```
        li    a0, 1        # 1 = StdOut
        la    a1, outstr   # string to print
        li    a7, 64       # linux write system call
        ecall              # Call linux to output the string

# Setup the parameters to exit the program
# and then call Linux to do it.
        li    a0, 0        # Use 0 return code
        li    a7, 93       # Service command code 93 terminates
        ecall              # Call linux to terminate
                             the program

.data
instr:  .asciz  "ThisIsRatherALargeVariableNameAaZz//[`{\n"
outstr: .fill  255, 1, 0
```

This file contains the **_start** entry point and **print** Linux calls, so no **main.S** is needed. Here is the output of building and running this version:

```
ubuntu@ubuntu:~$ make
as  -mno-relax  upper3.S -o upper3.o
ld -o upper3 upper3.o
ubuntu@ubuntu:~$ ./upper3
THISISRATHERALARGEVARIABLENAMEAAZZ[@[
```

There are special characters at the end of the string showing how some are converted correctly and some are not.

Besides using the **and** instruction to eliminate all conditional processing, the **toupper** routine is implemented as a macro to eliminate the overhead of calling a function. This is typical of many optimizations. This is how to save instructions by narrowing the problem domain, in this case to work on alphabetic characters rather than on all ASCII characters.

Note The space character has a hex value of 0x20, so if it is **and**'ed with 0xdf, the result will be zero, a null character, terminating the loop.

Tips for Optimizing Code

The first rule of optimizing code is to time and test everything. The designers of the RISC-V processor continually incorporate improvements into the hardware designs. Each year, the RISC-V processors get faster and more optimized. Since RISC-V processors are designed by several companies, the performance characteristics from processor to processor can be quite different. Improving performance through optimizing Assembly Language code is not always intuitive. The processor can be quite smart at some things and quite dumb at others. If tests are not set up to measure the results of changes, functions can become much worse.

With that said, general Assembly Language optimization techniques follow in the next section.

Avoiding Branch Instructions

Most RISC-V CPUs work on several instructions at once via an instruction pipeline, and if the instructions do not involve a branch, then everything works great. If the CPU hits a branch instruction, it must do one of the three things:

1. Throw away any work it did on instructions after the branch instruction.
2. Make an educated guess as to which way the branch is likely to go and proceed in that direction, then it only needs to discard the work if it guessed wrong.

3. Start processing instructions in both directions of the branch at once, perhaps it cannot do as much work, but it accomplishes something until the direction of the conditional branch is decided.

CPUs were getting quite good at predicting branches and keeping their pipelines busy, until the Spectre and Meltdown security exploits figured out how to access this work and exploit it. This caused CPU designers, including RISC-V, to reduce some of this functionality.

As a result, conditional branch instructions can be expensive and also lead to hard to maintain spaghetti code that should be avoided. Therefore, reducing conditional branches helps performance and leads to more maintainable code.

Moving Code Out of Loops

The most common optimization is to move any code inside a loop to outside the loop. Examine every instruction in a loop to see if it can be moved either before or after the loop. In the case of loops executed millions of times, saving even one clock cycle in loop execution can make a big difference.

Avoiding Expensive Instructions

Instructions like multiplication and division take multiple clock cycles to execute, so accomplishing them through additions or subtractions in an existing loop can help. Also, consider using bit manipulation instructions, like shifting left to multiply by two. If these instructions are necessary for the algorithm, then there is not much that can be done.

Similarly, floating-point instructions are much slower than integer instructions. Minimize reliance on floating-point arithmetic as much as possible.

Use Macros

Calling a function can be costly if a lot of registers need to be saved to the stack, then restored before returning. Do not be afraid of using macros to eliminate the function call and return instructions along with register saving/restoring. Also, eliminating the branches is often effective since it keeps the instruction pipeline full.

Loop Unrolling

This is repeating the code the same number of times of the loop, saving the overhead of the instructions that do the looping. For instance, if there is a loop that executes ten times, instead of a loop having ten copies of this code. Macros are an easy way to accomplish this.

Delay Preserving Registers in Functions

If a register that is callee saved in a function is only used under specific conditions, such as in the **then** clause of an **if** statement, then delay saving that register to the stack until that code path is taken. Similarly restore the value before leaving that code path. This is beneficial if that route through the code is infrequently taken.

Keeping Data Small

Even though RISC-V processors can mostly process instructions involving the full 64-bit **x** registers in the same time as using the 32-bit version, it puts a strain on the memory bus moving all that data. Remember that the memory bus is moving data, along with loading instructions to execute and doing all that for all the processing cores. Reducing the quantity of data moved to and from memory can help speed things up.

Beware of Overheating

A single RISC-V SBC CPU typically has four or more processing cores. All of these units can work at once with clever programming, theoretically processing a huge amount of data in parallel. The gotcha is that the more circuitry involved in processing, then the more heat produced.

If this is attempted, beware that a single board computer, like the Starfive Visionfive 2, can overheat. While the processor will not be damaged, it will detect the overheating and slow itself down, undoing all the great work done.

Summary

In this chapter, two optimizations were performed on the uppercase function. The following operations were discussed:

1. Simplifying range comparisons
2. Simplifying the domain and using bit manipulations

Next several hints to consider when optimizing code were provided.

In Chapter 13, "Reading and Understanding Code," we will examine how the C compiler generates code and talk about understanding compiled programs.

Exercises

1. In the first optimization, consider the alternate pseudocode:

```
x5 = x5 - 'a'
IF (x5 >= 0) AND x5 <= ('z'-'a') THEN
x5 = x5 + 'A'
END IF
```

Why is this incorrect?

2. Each generation of RISC-V CPUs, new instruction set extensions are added. List the pros and cons of utilizing newer instruction set extensions to optimize the code.

3. Set up a way to run each of the programs in this chapter in a large loop, and time how long each one takes. Which technique is fastest and why? Consider using the Linux **gettimeofday** service.

CHAPTER 13

Reading and Understanding Code

Now that the core of RISC-V Assembly Language has been covered, this chapter provides the fundamentals to read another programmer's code. Reading other programmer's code is a great way to not only add to a toolkit of tips and tricks but also improves coding. Also, where Assembly Language source code for the RISC-V processor can be found will be reviewed. One of the Assembly Language routines from the Linux kernel will be examined to learn new optimization techniques. Then how the GNU C compiler writes Assembly Language code will be examined and analyzed. The NSA's Ghidra hacking tool that converts Assembly Language code back into C code, at least approximately, will be studied.

The uppercase program will be used to see how the C compiler writes Assembly Language code and then examines how Ghidra can take that code and reconstitute the C code.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_13

Browsing Linux & GCC Code

One of the many nice things about working with Linux and the GNU Compiler Collection is that they are open source. That means browsing through the source code and perusing the Assembly Language parts contained there is allowed. They are available in the following Github repositories:

- Linux Kernel: `https://github.com/torvalds/linux`
- GCC Source Code: `https://github.com/gcc-mirror/gcc`

Clicking the "Clone or download" button and choosing "Download ZIP" is the easiest way to get them. Within all this source code, a couple of good folders to review RISC-V Assembly Language source code are as follows:

- Linux Kernel:
  - arch/riscv/include/asm
  - arch/riscv/lib
  - arch/riscv/kernel
  - arch/riscv/kvm
- GCC:
  - libgcc/config/riscv

Note The Assembly Language source code for these are in *.S files (note the uppercase S). This is so the C header files can be included, and C preprocessor directives can be used.

There are no separate folders for 32- and 64-bit. For RISC-V they are similar enough that the difference can be handled using conditional C preprocessor statements such as

```
#if __riscv_xlen == 32#endif
#endif
```

A lot can be learned by studying this code, for example, the Linux kernel version of **strcmp** will be studied next.

Comparing Strings

The Linux kernel contains machine-specific code to handle operations like the initialization of the CPU, handling interrupts, and performing multi-tasking. It also contains Assembly Language versions of many C runtime functions and other specialty functions that optimize the Linux kernel's performance.

The Linux Kernel wants to use the C runtime library. However, it must be initialized after Linux is up and running. Instead the Linux kernel has copies of some key runtime functions. Furthermore, special machine-specific, highly optimized versions are contained in the **arch/riscv/lib** folder.

There is a lot we can learn from these functions as they are written by top programmers and reviewed by many people.

The Linux kernel is written mostly in C and makes extensive use of C style, null-terminated strings. As a result, utilizing highly optimized versions of functions that operate on these strings is crucial. Often strings and other data structures in the Linux kernel are word aligned allowing further optimizations.

Listing 13-1 is the source code from the Linux 6.7 kernel currently under development, the file is **arch/riscv/lib/strcmp.S**. Linux kernel source code uses both C and Assembler macros, this routine contains fewer macros than most, so this code should be largely familiar. Before reading the following code, think of how it might implement an Assembly Language function to compare two null-terminated C strings.

Listing 13-1. The Linux kernel's strcmp function

```
/* SPDX-License-Identifier: GPL-2.0-only */

#include <linux/linkage.h>
#include <asm/asm.h>
#include <asm/alternative-macros.h>
#include <asm/hwcap.h>

/* int strcmp(const char *cs, const char *ct) */
SYM_FUNC_START(strcmp)

        ALTERNATIVE("nop", "j strcmp_zbb", 0, RISCV_ISA_EXT_
        ZBB, CONFIG_RISCV_ISA_ZBB)

        /*
         * Returns
         *   a0 - comparison result, value like strcmp
         *
         * Parameters
         *   a0 - string1
         *   a1 - string2
         *
         * Clobbers
         *   t0, t1
         */
```

```
1:
        lbu   t0, 0(a0)
        lbu   t1, 0(a1)
        addi  a0, a0, 1
        addi  a1, a1, 1
        bne   t0, t1, 2f
        bnez  t0, 1b
        li    a0, 0
        ret
2:
        /*
         * strcmp only needs to return (< 0, 0, > 0) values
         * not necessarily -1, 0, +1
         */
        sub   a0, t0, t1
        ret

/*
 * Variant of strcmp using the ZBB extension if available.
 * The code was published as part of the bitmanip manual
 * in Appendix A.
 */
#ifdef CONFIG_RISCV_ISA_ZBB
strcmp_zbb:

.option push
.option arch,+zbb

        /*
         * Returns
         *   a0 - comparison result, value like strcmp
         *
         * Parameters
```

```
         *   a0 - string1
         *   a1 - string2
         *
         * Clobbers
         *   t0, t1, t2, t3, t4
         */

        or    t2, a0, a1
        li    t4, -1
        and   t2, t2, SZREG-1
        bnez  t2, 3f

        /* Main loop for aligned string.  */
        .p2align 3
1:
        REG_L t0, 0(a0)
        REG_L t1, 0(a1)
        orc.b t3, t0
        bne   t3, t4, 2f
        addi  a0, a0, SZREG
        addi  a1, a1, SZREG
        beq   t0, t1, 1b

        /*
         * Words don't match, and no null byte in the first
         * word. Get bytes in big-endian order and compare.
         */
#ifndef CONFIG_CPU_BIG_ENDIAN
        rev8  t0, t0
        rev8  t1, t1
#endif

        /* Synthesize (t0 >= t1) ? 1 : -1 in a branchless
           sequence. */
```

```
        sltu  a0, t0, t1
        neg   a0, a0
        ori   a0, a0, 1
        ret

2:
        /*
         * Found a null byte.
         * If words don't match, fall back to simple loop.
         */
        bne   t0, t1, 3f

        /* Otherwise, strings are equal. */
        li    a0, 0
        ret

        /* Simple loop for misaligned strings. */
        .p2align 3
3:
        lbu   t0, 0(a0)
        lbu   t1, 0(a1)
        addi  a0, a0, 1
        addi  a1, a1, 1
        bne   t0, t1, 4f
        bnez  t0, 3b

4:
        sub   a0, t0, t1
        ret

.option pop
#endif
SYM_FUNC_END(strcmp)
```

The function is implemented twice, depending on whether the RISC-V **Zbb** bitwise operations extension is present. If **Zbb** is present, the routine can use a word-by-word comparison; then if it isn't present, a simple byte-by-byte comparison is used. The source code for this book contains an example where this code is separated and free of Linux dependencies, making it easier to play with. Both the QEMU emulator and Starfive Visionfive 2 support the **Zbb** extension, so they can run the quicker version. The following subsections explain how this function works and why it is implemented the way it is.

About the Algorithm

The first version of the routine copies byte-by-byte and only uses instructions from the core integer instruction set. It is a simple, straightforward implementation and easy to follow. Since it uses **lbu** instructions to load byte-by-byte, this works equally well in both 32- and 64-bits.

The second version processes via loading the size of a register at a time. This means for 64-bits, eight bytes are processed at each step of the loop, potentially speeding things up eight times. This algorithm only kicks in if both strings are aligned, saving extra code to handle the unaligned bytes at the start of the strings. Most C data structures are aligned, so this will be the case for most strings being processed by the Linux kernel. If the strings aren't aligned, then a copy of the byte-by-byte algorithm from the first case is jumped to. The checks for alignment by **or**'ing the two string addresses together and then **and**'ing them against the bit-size minus one. If this is non-zero, then one of the strings is not aligned. See Exercise 13-2.

The trick to this algorithm is to check for the null byte string terminator efficiently. In the case of 64-bit, this means checking each of the eight bytes for null one-by-one. Doing this with the instructions studied so far would

be quite complex, involving comparisons and branches, defeating the optimization that is being attempted. The answer is an obscure instruction in the RISC-V **Zbb** extension:

```
orc.b rd, rs
```

Here is the description from the *RISC-V Bit-Manipulation ISA-extensions* specification:

> *Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result **rd** to all zeros if no bit within the respective byte of **rs** is set, or to all ones if any bit within the respective byte of **rs** is set.*

The effect is that if any bit in a byte of **rs** is non-zero, then all the bits of the corresponding byte of **rd** will be set to all ones, and if the null byte is encountered, then the corresponding bits will be set to zero. This means if **rd** is compared to -1 (all ones) then equality means that there is no null byte present. If there is a null byte, then the equality will fail. Before the loop **t4** is loaded with -1 and then the comparison

```
bne   t3, t4, 2f
```

is used to determine if the loop is complete. This branches to the byte-by-byte loop to complete the comparison up to the null. The usual comparison can't be used since comparing any bytes after the null is incorrect.

Note The option:

```
.option arch,+zbb
```

is used to enable the **Zbb** extension to the assembler. Otherwise, an unknown opcode error may result.

Macros and Kernel Options

Reading Linux kernel source code is made difficult due to all the processor variants supported. A key part to building Linux kernels is setting all the hundreds of build parameters that can dramatically change the performance, compatibility, and size of the kernel. In this code, it supports both 32- and 64-bit RISC-V processors. The core routine is written for the core integer instruction set RV32I or RV64I but then includes a version of the routine that utilizes an instruction **orc.b** from the **Zbb** bit manipulation extension.

The ALTERNATIVE macro, defined in arch/riscv/include/asm/alternative-macros.h, controls which routine is included, depending on whether the **Zbb** extension is configured.

The main difference between 32- and 64- bits is that to load a full register a selection between using **lw**/**ld** and **sw**/**sd** instructions is required to load the appropriate size of integer. These differences are handled by macros in arch/riscv/include/asm.h. They allow to switch between two different instructions based on whether they are 32- or 64- bits. The **REG_L** macro is used in the aligned version with the **Zbb** extension to select the correct load function based on bitness.

```
#define __ASM_STR(x)        x
#if __riscv_xlen == 64
#define __REG_SEL(a, b)    __ASM_STR(a)
#elif __riscv_xlen == 32
#define __REG_SEL(a, b)    __ASM_STR(b)
#else
#error "Unexpected __riscv_xlen"
#endif

#define REG_L              __REG_SEL(ld, lw)
#define REG_S              __REG_SEL(sd, sw)
```

The macros SYM_FUNC_START and SYM_FUNC_END are defined in include/linux/linkage.h. They contain the GNU Assembler directives to ensure the routine is aligned properly and the function name is global.

Quite a bit of time has been spent on writing Assembly Language, now have a look at how the GNU C compiler writes Assembly code.

Code Created by GCC

In this section, the uppercase routine is coded in C and compares the generated code to what was written. For this example, **gcc** needs to do as good a job as possible, so the **-O3** option is used for maximal optimization.

Create **upper.c** from Listing 13-2.

Listing 13-2. C implementation of the **`mytoupper`** routine

```
#include <stdio.h>

int mytoupper(char *instr, char *outstr)
{
        char cur;
        char *orig_outstr = outstr;

        do
        {
                cur = *instr;
                if ((cur >= 'a') && (cur <='z'))
                {
                        cur = cur - ('a'-'A');
                }
                *outstr++ = cur;
                instr++;
```

```
        } while (cur != '\0');
        return( outstr - orig_outstr );
}

#define BUFFERSIZE 250

char *tstStr = "This is a test!";
char outStr[BUFFERSIZE];

int main()
{
        mytoupper(tstStr, outStr);
        printf("Input: %s\nOutput: %s\n", tstStr, outStr);

        return(0);
}
```

Compile this with the following command:

```
gcc -O3 -o upper upper.c
```

then run **objdump** to see the generated code:

```
objdump -d upper >od.txt
```

Listing 13-3 is the result.

Listing 13-3. Assembly code generated by the C compiler for the uppercase function.

```
00000000000005d0 <main>:
 5d0:  1141                add    sp,sp,-16
 5d2:  00002597            auipc  a1,0x2
 5d6:  a365b583            ld     a1,-1482(a1) # 2008 <tstStr>
 5da:  e406                sd     ra,8(sp)
 5dc:  862e                mv     a2,a1
 5de:  00002697            auipc  a3,0x2
```

```
5e2:  a8268693        add    a3,a3,-1406 # 2060 <outStr>
5e6:  4865            li     a6,25
5e8:  00064783        lbu    a5,0(a2)
5ec:  f9f7871b        addw   a4,a5,-97
5f0:  0ff77713        zext.b a4,a4
5f4:  fe07851b        addw   a0,a5,-32
5f8:  02e86063        bltu   a6,a4,618 <main+0x48>
5fc:  00a68023        sb     a0,0(a3)
600:  00164783        lbu    a5,1(a2)
604:  0685            add    a3,a3,1
606:  0605            add    a2,a2,1
608:  f9f7871b        addw   a4,a5,-97
60c:  0ff77713        zext.b a4,a4
610:  fe07851b        addw   a0,a5,-32
614:  fee874e3        bgeu   a6,a4,5fc <main+0x2c>
618:  00f68023        sb     a5,0(a3)
61c:  0685            add    a3,a3,1
61e:  c399            beqz   a5,624 <main+0x54>
620:  0605            add    a2,a2,1
622:  b7d9            j      5e8 <main+0x18>
624:  00002617        auipc  a2,0x2
628:  a3c60613        add    a2,a2,-1476 # 2060 <outStr>
62c:  00000517        auipc  a0,0x0
630:  11c50513        add    a0,a0,284 # 748 <_IO_stdin_
                             used+0x8>
634:  f8dff0ef        jal    5c0 <printf@plt>
638:  60a2            ld     ra,8(sp)
63a:  4501            li     a0,0
63c:  0141            add    sp,sp,16
63e:  8082            ret
```

A few things to notice about this listing are as follows:

- The compiler automatically inlined the **mytoupper** function like the macro version. The **mytoupper** function is elsewhere in the listing in case it is called from another file.
- The compiler knows about the range optimization and shifts the range, so it only makes one comparison. The shift is performed by

  ```
  addw   a4,a5,-97
  ```

- All the variables fit in the corruptible registers. As a result, it only saves and restores the **ra** register, since **printf** is called.
- There are a few occurrences of:

  ```
  zext.b a4,a4
  ```

  This is a pseudoinstruction for:

  ```
  andi a4, a4, 255
  ```

  This is to maintain type correctness in C. A C char data type is an unsigned 8-bit number. When it is subtracted, it could go negative, resulting in the upper bits of **a4** being set to 1. This corrects it back to an unsigned quantity. This is not in our routine, because it is only ever saved using the 8-bit **sb** instruction; therefore, the upper bits would be ignored whatever they are.

- The compiler always performs the case conversion with

  ```
  addw   a0,a5,-32
  ```

  then based on the comparison, it either saves **a0** or **a5** depending upon whether the conversion is required or not.

Overall, the compiler did a reasonable job of compiling our code, but there are a few instructions that can be removed. It can certainly be seen how some hand optimization will help.

This is why many Assembly Language programmers start with C code, then remove any extra instructions. The C code becomes less efficient once it can't fit all the variables in registers and must start swapping data to and from the stack. This usually happens when the complexity is higher, and the need for speed is greater.

In Chapter 8, "Programming GPIO Pins," programming the GPIO pins using the GPIO controller's memory registers was introduced. This sort of code confuses the optimizer. Often it needs to be turned off, or it optimizes away the code that accesses these locations. This is because memory locations are written to and are never read, and also memory is read that has not been set. There are keywords to help the optimizer; however, Assembly Language can result in much improved code, due to working against the C optimizer, which does not know what the GPIO controller is doing with this memory.

Reverse Engineering and Ghidra

In the Linux world, most of the programs encountered are open source, where the source code can easily be downloaded and studied. There is documentation on how it works, and people are actively encouraged to contribute to the program, perhaps fix bugs or add a new feature.

Suppose a program is encountered where the source code is not available, and we want to know how it works. Perhaps, to study it, to see if it contains malware. It might be the case that we are worried about privacy concerns and want to know what information the program sends on the Internet. Maybe, it's a game, and we want to know if there is a secret code, we can enter to go into God mode. What is the best way to go about this?

The Assembly Language code of any Linux executable can be examined using **objdump** or **gdb**. Having learned enough about Assembly Language from studying this book, that sense can be made of instructions encountered. However, this does not help form a big picture of how the program is structured and it is time-consuming to examine the raw Assembly language code, but there are tools to help with this.

Until recently there were only expensive commercial products available to parse raw Assembly Language code; however, the National Security Agency (NSA), yes, that NSA, released a version of the tool that their own hackers use to analyze code, called Ghidra, after the three-headed monster that Godzilla fights. This tool analyzes compiled programs and includes the ability to decompile a program back into C code, tools to show graphs of function calls, and the ability to make annotations as the program structure is discovered.

Ghidra can be downloaded from `https://ghidra-sre.org/`. To install it, unzip it, then run the **ghidraRun** script if using Linux or **ghidraRun.bat** on Windows. Ghidra requires the Java runtime, if it is not already installed, install it on the target operating system.

Note Ghidra requires the 64-bit version of Oracle Java 17 or later. The easiest way to run this is on an Intel- or AMD-based computer. Then copy the executable or object file from the RISC-V system over to this computer and Ghidra will happily analyze it.

Decompiling an optimized C program is difficult. As shown in the last section, the **GCC** optimizer does major rewriting of the original code as part of converting it to Assembly Language. Take the upper program that was compiled from C in the last section, give it to Ghidra to decompile, and see whether the result is like the starting source code.

1. Create a new project in Ghidra, usually non-shared, and with the name **upper**.
2. Select "File - Import File..." and select the upper program. An information dialog appears as shown in Figure 13-1.

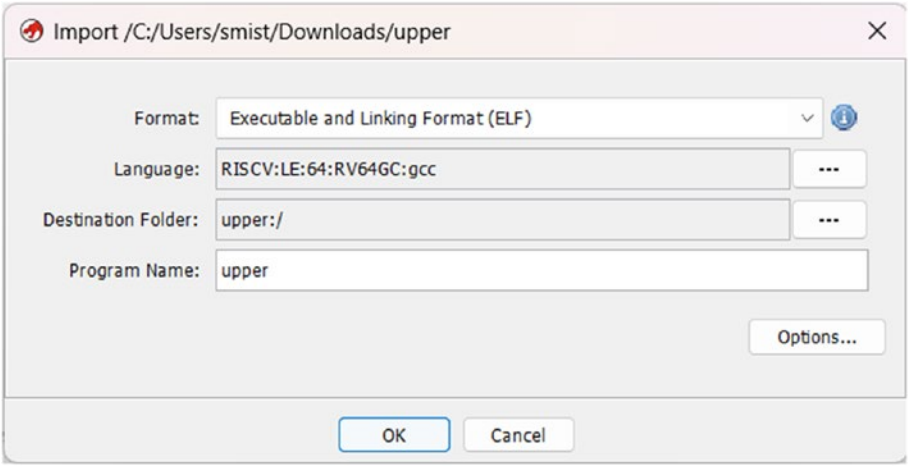

Figure 13-1. *High-level information on the upper executable*

3. The Import Results Summary window pops up with more detailed data. Click OK to get the main window.

4. Right click the upper executable and select: "Open in default tool." This opens the code analysis window. Click ***Yes*** when asked "upper has not been analyzed. Would you like to analyze it now?" and click ***Analyze*** in the following options dialog. Figure 13-2 is the resulting code analysis window.

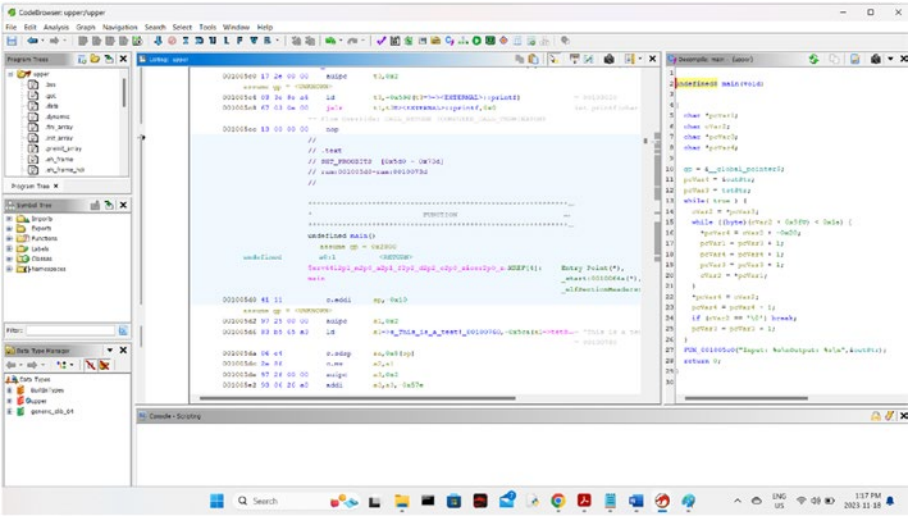

Figure 13-2. *Ghidra analyze window for the upper program*

Listing 13-4 is the C code that Ghidra generates. The lines above the definition of the main routine were added, and the **tstStr** parameter was added to the **printf** function so the program will compile and run. Ghidra also tried to include code to initialize the global pointer, which is commented out.

Listing 13-4. C code created by Ghidra for the **`upper`** C program

```
#include <stdio.h>

#define BUFFERSIZE 250

char *tstStr = "This is a test!";
char outStr[BUFFERSIZE];

typedef unsigned char byte;
typedef int undefined8;

#define true 1
#define FUN_001005c0 printf

undefined8 main(void)

{
  char *pcVar1;
  char cVar2;
  char *pcVar3;
  char *pcVar4;

  // gp = &__global_pointer$;
  pcVar4 = &outStr;
  pcVar3 = tstStr;
  while( true ) {
    cVar2 = *pcVar3;
    while ((byte)(cVar2 + 0x9fU) < 0x1a) {
      *pcVar4 = cVar2 + -0x20;
      pcVar1 = pcVar3 + 1;
      pcVar4 = pcVar4 + 1;
      pcVar3 = pcVar3 + 1;
      cVar2 = *pcVar1;
    }
```

```
    *pcVar4 = cVar2;
    pcVar4 = pcVar4 + 1;
    if (cVar2 == '\0') break;
    pcVar3 = pcVar3 + 1;
  }
  FUN_001005c0("Input: %s\nOutput: %s\n",tstStr, &outStr);
  return 0;
}
```

Run the program. The expected output is as follows:

```
ubuntu@ubuntu:~$ gcc upperghid.c
upperghid.c: In function 'main':
upperghid.c:23:10: warning: assignment to 'char *' from incompatible
pointer type 'char (*)[250]' [-Wincompatible-pointer-types]
   23 |   pcVar4 = &outStr;
      |          ^
upperghid.c:39:37: warning: format '%s' expects argument of type
'char *', but argument 3 has type 'char (*)[250]' [-Wformat=]
   39 |   FUN_001005c0("Input: %s\nOutput: %s\n",tstStr, &outStr);
      |                                    ~^            ~~~~~~~
      |                                     |            |
      |                                     char *       char (*)[250]
ubuntu@ubuntu:~$ ./a.out
Input: This is a test!
Output: THIS IS A TEST!
```

The code produced is not pretty. The variable names are generated. It knows about **tstStr** and **outStr** because these are global variables. The logic is in smaller steps, often each C statement is the equivalent of a single Assembly Language instruction. Notice how the **if** statement to determine if a character is lowercase became a **while** loop. Similarly, notice how the main loop was changed from a **do**/**while** to a **while** with an **if**/**break** near

the end. When trying to figure out a program without the source code, having different viewpoints is a great help. As compiler code optimizers get better and better, it becomes a real challenge for tools like Ghidra to unravel what **gcc** optimizers have done.

Note This technique only works for true compiled languages like C, Fortran, or C++. It does not work for interpreted languages like Python or JavaScript, nor for partially compiled languages that use a virtual machine architecture like Java or C#. There are other tools for these and often these are much more effective since the compile step doesn't do as much.

Summary

In this chapter, some sample Assembly Language source code in the Linux kernel and the **GCC** runtime library is located was examined. The Linux kernel's **strcmp** function was studied to see how it works. A C version of the uppercase program was written, so the Assembly Language code that the C compiler produces was studied and compared to what was written by hand.

The sophisticated Ghidra program for decompiling programs was looked at and used to see what it produces. Although it produces working C code from Assembly Language code, it is not that easy for programmers to read.

In Chapter 14, "Hacking Code," how hackers use Assembly Language knowledge to hack code and take control of computers will be explained.

Exercises

1. Manually execute the instructions in Listing 13-1 that perform the loop to ensure it is understood how it works and that it performs the correct number of iterations. This book's sample code includes a version that can be run directly.

2. Consider the code that checks for word alignment.

```
or     t2, a0, a1
and    t2, t2, SZREG-1
bnez   t2, 3f
```

   Manually try a few cases with **SZREG** set to two, to ensure this code works properly.

3. Look at the Linux kernel library function **memset.S** located in arch/riscv/lib. Can this code be easily read?

4. Compile the C code generated by the Ghidra disassembler in Listing 13-4, then run **objdump** on the output and compare it to the original Assembly Code in Listing 13-3. Is this result what was expected?

5. Examine one of the smaller executables from /usr/bin, such as **head**, in Ghidra. How does it work? What is the main block of code?

CHAPTER 14

Hacking Code

For this chapter, hacking means gaining illicit access to a computer or network by various tricky means. The goal is to give programmers the tools and knowledge to prevent code being hacked by criminal hackers. This chapter offers techniques to hack programs by providing them with bad data. Another form of hacking is social engineering where people are scammed into revealing their passwords, or other personal data by sophisticated methods over the phone, social media, or email; however, that is a topic for a different book.

Every programmer should know about hacking. If a programmer does not know how hackers exploit security weaknesses, then that programmer will unknowingly provide access for hackers to wreak havoc on their code.

Buffer Overrun Hack

As an example, the classic buffer overrun problem will be examined, including how it happens, how to exploit it, then how to protect against it. Anyone with security experience will notice that our uppercase routine is error-prone and will likely lead to buffer overrun vulnerabilities in the program. Next, what buffer overrun is and how it gets exploited is covered.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2_14

Causes of Buffer Overrun

The uppercase routine happily converts text to uppercase until it hits a NULL (0) character. If the provided input text is bigger than the output buffer the caller provides, then this routine overwrites whatever is in memory after the output buffer. Depending on where the buffer is located, this affects the type of attack that is possible. This buffer being located on the stack will be looked at. The weakness of the stack is that this is where function return addresses get stored when function calls are nested. If the attack is arranged precisely, the function's return address can be overwritten and cause the function to return to a selected place.

There are other forms of buffer overrun attacks if the data is stored in the C runtime heap, or in the program's data segment. These attacks are similar to the one explored for the stack. If too much data is entered into such a text field, the program typically crashes, since important program data is overwritten, for instance, corrupting pointers. Even though the hacker will not get proprietary data this way, this is still a good foundation for a **Denial of Service** (DoS) attack. If this is a web server and a hacker causes it to crash, then it needs to be restarted and re-initialized. This typically takes several seconds. This means a hacker can send a message to the web server every few seconds to keep it offline.

Stealing Credit Card Numbers

Imagine a credit card company's web server running a web application that uses the uppercase program, because it needs to convert names to uppercase super-fast, so that its web pages are exceptionally responsive. Suppose there is a page on the website where a customer enters their name, and the web application converts it to uppercase; however, the web page is not error checking for the length of data and passes it to our uppercase routine as is. Furthermore, for convenience, this web

application provides several administrative utilities, such as a facility to download all the credit card data, so it can be backed up. These utilities are only available to administrative users with special clearance and require a digital certificate to access. A hacker wants to dupe the customer facing part of the website into giving them access to the administrative part without requiring extra user authentication.

In Chapter 6, "Functions and the Stack," the function call protocol specified that if a function calls another function, it must store the **ra** register to the stack, so that it will not be lost. The uppercase main program and uppercase routine will be modified to have an intermediate routine, so **ra** is stored to the stack and the output buffer will be allocated on the stack, rather than stored in the data segment.

Listing 14-1 contains three routines: the skeleton of the credit card company's web application. It has the usual **_start** entry point that calls the routine **calltoupper**. This routine pushes **ra** to the stack and allocates 16-bytes for the output buffer. The **DownloadCreditCardNumbers** routine should not be accessible to regular users. The specially constructed input data that a hacker enters in a text box will cause nefarious things to happen is constructed in the data segment.

Listing 14-1. Main web application for the credit card company

```
#
# Assembler program to demonstrate a buffer
# overrun hacking attack.
#
# a0-a2 - parameters to Linux function services
# a1 - address of output string
# a0 - address of input string
# a7 - Linux function number
#

.global _start            # Provide program starting address
```

```
DownloadCreditCardNumbers:
# Setup the parameters to print hello world
# and then call Linux to do it.
        li    a0, 1     # 1 = StdOut
        la    a1, getcreditcards # string to print
        li    a2, 30    # length of our string
        li    a7, 64    # linux write system call
        ecall           # Call linux to output the string
        ret

calltoupper:
        addi  sp, sp, -16  # allocate 16 bytes on stack
        sd    ra, 0(sp)    # push return address
        addi  sp, sp, -16  # 16 bytes for outstr
        la    a0, instr    # start of input string
        mv    a1, sp       # address of output string

        call  toupper

aftertoupper:       # convenient label to use as a breakpoint
        addi  sp, sp, 16   # Free outstr
        ld    ra, 0(sp)    # pop ra
        addi  sp, sp, 16   # deallocate stack space
        ret

_start:

        call  calltoupper

# Setup the parameters to exit the program
# and then call Linux to do it.
        li    a0, 0        # Use 0 return code
        li    a7, 93       # Service command code 93 terminates
        ecall              # Call Linux to terminate
                             the program
```

```
.data
instr:  .ascii  "This is our Test" # Correct length string
        .dword 0x00000000000100e8  # overwrite for ra
getcreditcards:       .asciz  "Downloading Credit Card Data!\n"
        .align 4
```

For this example, the first example of the uppercase routine from Chapter 6, "Functions and the Stack," is used. When this program is compiled and run, the result is

```
Downloading Credit Card Data!
```

repeated over and over until Control+C is pressed. This is in spite of the routine **DownloadCreditCardNumbers** never being called within the program. Why the program is put in an infinite loop will be explained shortly.

The code for the user interface is not included, rather the data is provided in the .data section, to keep things simple and easy to follow.

Next, what happens to the stack through the process as this function runs is examined.

Stepping Through the Stack

The stack is set up in the **calltoupper** function. Figure 14-1 shows the values of **sp** and what is stored in each 16-byte block. Remember that **sp** must always be 16-byte aligned.

0x3fffff030			original sp
0x3fffff020	8 bytes for ra	8 bytes zero	addi sp, sp, -16; sd ra, 0(sp)
0x3fffff010	16 byte buffer for outstr		addi sp, sp, -16

Figure 14-1. *The contents of the stack inside the* ***calltoupper*** *function*

Remember that the stack grows downwards, so when something is pushed onto the stack, **sp** is decremented. The pointer passed for outstr will be 0x3fffffff010, and since the loop in the uppercase routine increments, if it overflows its buffer, it overwrites the stored value for **ra** located at memory address 0x3fffffff020. The hacker's strategy is to overwrite **ra** with an address causing the program to do their bidding.

Listing 14-2 shows the memory addresses of the key instructions considered. The hacker wants to overwrite the **ra** register with 0x100e8, that's the address of the **DownloadCreditCardNumbers** routine.

Listing 14-2. Excerpts of the objdump output of the program in Listing 14-1

00000000000100e8 <DownloadCreditCardNumbers>:

```
   100e8:     00100513            li     a0,1
...
0000000000010104 <calltoupper>:
   10104:     ff010113            add    sp,sp,-16
   10108:     00113023            sd     ra,0(sp)
...
0000000000010134 <_start>:
   10134:     00000097            auipc  ra,0x0
   10138:     fd0080e7            jalr   -48(ra) # 10104
<calltoupper>
   1013c:     00000513            li     a0,0
```

1. In **_start** the **jalr** is run to call the **calltoupper** routine. This places the address of the next instruction into **ra** and jumps to **calltoupper**. This means **ra** has the value 0x1013c at this point.

2. On entering **calltoupper**, **sp** contains 0x3fffff030. Executes the:

```
add   sp,sp,-16
sd    ra,0(sp)
```

instructions which decrements **sp** by 16 and copies **ra** to this memory location. This makes **sp** 0x3fffff020 and the 16-bytes there contain:

```
0x3fffff020: 0x0000000000010126 0x0000000000000000
```

Showing that **ra** was pushed to the stack.

3. Execute

```
add   sp,sp,-16
```

This allocates 16 bytes for our output buffer. This reduces the stack pointer to 0x3fffff010 and the contents of the stack are

```
0x3fffff010: 0x0000000000000000 0x0000000000000000
0x3fffff020: 0x0000000000010126 0x0000000000000000
```

4. The function **toupper** converts our string to uppercase. It does this correctly for the first part of the string "This is our Test" (16-bytes). Since there is no NULL (0) terminator, it will also process the next byte 0xe8 that is not lowercase, so will be copied as is. The next byte is a NULL (0), so it stops, but it does

copy the NULL byte which is necessary to change the address to the one that is desired. The register **sp** is not affected by this series of operations, but upon returning from **toupper**, the stack contains

```
0x3ffffff010: 0x2053492053494854 0x545345542052554f
0x3ffffff020: 0x00000000000100e8 0x0000000000000000
```

The first line is the new string, converted to uppercase. But notice the return address at 0x3ffffff020 has changed from 0x00010126 to 0x000100e8. This means the return address is the address of the **DownloadCreditCardNumbers** routine.

5. The **calltoupper** cleans up the stack and returns:

```
addi  sp, sp, 16   # Free outstr
ld    ra, 0(sp)    # pop ra
addi  sp, sp, 16   # deallocate stack space
ret
```

The key point is that the **ld** instruction loads the address of **DownloadCreditCardNumbers** into **ra**, then the **ret** instruction branches to that routine causing a major data breach.

In performing this hack, the hacker is lucky on a couple of points:

1. Only one byte of data needs to be copied and the NULL byte to get the address changed to the desired address.
2. The byte needed to copy was not one for a lowercase letter, so it was left alone by the **toupper** routine.

A successful hack usually requires luck and fortuitous circumstances. If this was not the case, there are still options. For example, it is possible to jump into the middle of the **DownloadCreditCardNumbers** routine. The start of a function usually contains a function prologue, that if it were not intended to successfully return from can be skipped. After all, if it is not important for the program to continue working correctly, only that credit card numbers are downloaded.

The reason the program goes into an infinite loop is because a **jalr** is not used to call **DownloadCreditCardNumbers**, instead a **ret** instruction is used. So, nothing updates **ra** to a new value, therefore the **ret** at the end of **DownloadCreditCardNumbers** jumps to the same address again.

This was an example of one particular buffer overrun exploit; however, hackers have many ways to exploit buffer overruns, whether the data is on the stack, in the C memory heap, or in our data segment. The following are several ways to avoid buffer overrun problems.

Mitigating Buffer Overrun Vulnerabilities

To combat buffer overrun problems, there are techniques to use in the code and those tools can provide help. In this section, both will be examined. First, consider the bad design of the function parameters to the uppercase routine. Before considering a solution, look at the root cause of many buffer overrun problems, the C runtime's **strcpy** function, and the various solutions proposed to fix this design. Since all compiled programming languages use the RISC-V function calling protocol, these issues are the same for both Assembly Language and C programmers.

Do Not Use strcpy

The C runtime's **strcpy** routine has the following prototype:

```
char * strcpy ( char * destination, const char * source );
```

It copies characters from source to destination, until a NULL (0) character is encountered. This results in buffer overrun vulnerabilities like the one just encountered. The original suggested solution, was to replace all occurrences of **strcpy** with **strncpy**:

```
char * strncpy ( char * destination, const char * source,
        size_t num );
```

Place the size of the destination in **num**, and it stops copying at that point. That stops the buffer overrun at this point, but now the destination string is not NULL (0) terminated, and this could lead to a buffer overrun later in the code. One suggestion is to always do the following:

```
strncpy( dest, source, num );
dest[num-1] = '\0';
```

This NULL terminates the string, but it requires the programmer to remember to always do this. Perhaps, under deadline pressure, this may be forgotten.

A new function was then introduced to the BSD C runtime, **strlcpy** that always NULL terminates the destination string.

```
size_t strlcpy(char *destination, const char *source,
        size_t size);
```

This function eliminates that problem, as the destination is always NULL (0) terminated, but this function is non-standard and not part of the GNU C library.

A criticism of both **strncpy** and **strlcpy** type functions is that they eliminate the ability to nest these functions to quickly build larger more complicated strings. This is because the remaining buffer length is not easily determined if concatenating strings together. Another suggested solution is the following:

```
char * strecpy ( char * destination, const char * source,
        char * end );
```

This **strecpy** passes in a pointer to the end of the destination buffer. This is handy when calls are nested, since **end** stays constant, unlike a remaining length that shrinks as the nest is built. Again, this is a nonstandard function and not part of the C runtime.

These functions all stop overwriting the destination buffer and prevent data corruption. However, they all have a problem that they could allow the leakage of sensitive data. Suppose the source is not NULL (0) terminated and the source buffer is smaller than the destination buffer, then the function will copy data until the destination buffer is full. This means possibly sensitive data was copied from past the end of the source buffer into the destination buffer. If this is displayed later, it might give away sensitive or helpful information to hackers. This leads to another form:

```
errno_t strncpy_s(char * destination, size_t destmax,
        const char * source, size_t srcmax);
```

In **strncpy_s**, the size of both buffers is provided, and the function returns an error code to indicate what happened.

This discussion was to point out that there are a lot of tradeoffs in fixing API designs. When making the uppercase routine more secure, there are quite a few pros and cons to consider. A list of recommendations will be presented towards the end of this chapter, but first what the operating system and GNU compiler can do to help will be discussed.

PIE Is Good

The exploit performed earlier relied upon knowing the address of the **DownloadCreditCardNumbers** routine. The assumption is that the hacker learned this from somewhere else, perhaps by obtaining an illicit copy of the application's source code, or the build map file from the dark web.

With modern virtual memory systems, the operating system can give a process any memory addresses it likes, they do not need to have any relation to real memory addresses. This gave rise to a feature called Position Independent Executables (**PIE**) introduced to Linux around 2005. With this feature an executable is loaded with a different base address each time it is run. This is a special case of Address Space Layout Randomization (**ASLR**) and this feature is often referred to by either name.

This sounds good, so why did the exploit performed earlier work? Why could not PIE defeat if? The reason is that PIE needs to be turned on in the command line for the **ld** command. This is a conservative approach, whereby turning it on it is acknowledging that there is no code that cannot be relocated. Furthermore, none of the shared libraries use code that cannot be relocated. To turn on PIE, add **-pie** to the list of options for the **ld** command.

Note At the time of this writing, there is a problem with the installation of the GNU toolchain on RISC-V, and for this to work, the following commands need to be run first:

```
user@starfive:/lib$ cd /lib
user@starfive:/lib$ sudo cp ld-linux-riscv64-lp64d.so.1 ld.so.1
```

When this option is chosen, a routine in **ld.so** configures parameters before calling the **_start** entry point.

If this is done, the following is

```
user@starfive:~/Chapter14$ make -B
gcc -mno-relax  -c creditcard.S -o creditcard.o
gcc -mno-relax  -c upper.S -o upper.o
ld -o creditcard creditcard.o upper.o
```

```
ld -pie -o creditcardpie creditcard.o upper.o
user@starfive:~/Chapter14$ ./creditcardpie
Illegal instruction
```

The error may be "Segmentation fault" depending on the memory values. If the program is debugged with **gdb**, all the addresses change and can be examined. Often when debugging PIE is turned off, it is only enabled for release to make decoding what is going on easier. This still is not ideal, it is better since the credit card numbers did not get stolen, but the program still crashed. This can lead to an easy **DoS** attack for hackers to make our application unavailable.

The program needs to be relocatable. What stops a program being relocatable? Mostly hard-coding memory addresses in the data section that the linker does not know about. For example, when the **la** pseudoinstruction is used, it translates to **auipc** and **add** instructions to create the address in memory to use, but it also creates a relocation record so the loader can fix up the address.

In Chapter 2, "Loading and Adding," how to load a register with a **li** pseudoinstruction was shown, which would translate to a number of **slli** and **addi** instructions. If this technique is used to load a memory address, then the program will not be relocatable as the loader has no idea what the code is doing and cannot fix up the address.

It is a good practice to enable PIE for any C or Assembly Language programs. PIE is not perfect; therefore, hackers have found ways around it. But it introduces a second step, hackers usually require a second vulnerability in addition to the buffer overrun to hack the program.

Poor Stack Canaries Are the First to Go

The GNU C compiler has a feature to detect buffer overruns. The idea is to add extra code to place a secret random value next to the stored function return address. Then this value is tested before the function returns and

if corrupted, then a buffer overrun has occurred and the program is terminated. These stack canaries are like the proverbial canaries in a coal mine, because when something goes wrong they are the first to go and warn that something bad is happening.

The source code that accompanies this book has a version of **upper.c** from Chapter 13, "Reading and Understanding Code," that introduces a buffer overrun. Like PIE, this is an optional feature and needs to be enabled with a **gcc** command line option. Here **-fstack-protector-all** is used, which is the most aggressive form of this feature. If this is added, compiled, and run, the result is as follows:

```
user@starfive:~/Chapter14$ make
gcc -o uppercanary -fstack-protector-all -O3 upper.c
user@starfive:~/Chapter14$ ./uppercanary
Input: This is a test!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyy
andevenlongerandlongerandlonger
Output: THIS IS A TEST!XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXYYYYYY
ANDEVENLONGERANDLONGERANDLONGER
*** stack smashing detected ***: terminated
Aborted
user@starfive:~/Chapter14$
```

This is great, as it prevents the buffer overrun, but it is quite expensive processing and timewise, since it adds quite a few instructions to every function. Next look at the code that is generated inside the **main** function. The following is extracted from an **objdump** of this program.

```
0000000000000690 <main>:
 690:   7179                    add    sp,sp,-48
 692:   f022                    sd     s0,32(sp)
 694:   0030                    add    a2,sp,8
 696:   00002417                auipc  s0,0x2
```

```
 69a:  9aa43403          ld     s0,-1622(s0) # 2040
                                <__stack_chk_guard@
                                GLIBC_2.27>
 69e:  f406              sd     ra,40(sp)
 6a0:  601c              ld     a5,0(s0)
 6a2:  ec3e              sd     a5,24(sp)
 6a4:  4781              li     a5,0

// body of routine ...

 6fc:  6762              ld     a4,24(sp)
 6fe:  601c              ld     a5,0(s0)
 700:  8fb9              xor    a5,a5,a4
 702:  4701              li     a4,0
 704:  e791              bnez   a5,710 <main+0x80>
 706:  70a2              ld     ra,40(sp)
 708:  7402              ld     s0,32(sp)
 70a:  4501              li     a0,0
 70c:  6145              add    sp,sp,48
 70e:  8082              ret
 710:  f61ff0ef          jal    670 <__stack_chk_fail@plt>
```

Six instructions were added to the function prologue and seven instructions to the function epilogue.

In detail, the instructions in the function prologue are gone through one-by-one:

1. **add**: Allocates 48 bytes on the stack, which is room for what the program needs plus the stack canary.
2. **sd**: Saves register **s0** to the stack. This register is used as a pointer to where the stack canary value is stored in the data segment.
3. **add**: Configures **a2** as the C stack frame.

4. **auipc**: First part of forming the address of the stack canary. The **s0** register is used as a pointer to this data combined with the offset in the **ld**/**sd** instructions.
5. **ld**: Offset -1622 is where the stack canary is stored. This loads the actual address of the stack canary, allowing it to be anywhere in memory.
6. **sd**: Store the return address, **ra**, to the stack.
7. **ld**: Load the value of the stack canary into register **a5**.
8. **sd**: Store the stack canary to the correct place on the stack to guard the function return pointer (pushed **ra**).
9. **li**: Overwrite the register with the stack canary, **a5**, with zero, so it is not left lying around. This is to try and prevent data leakage.

Next, the instructions in the function epilogue are as follows:

1. **ld**: Load the stack canary from the stack into register **a4**.
2. **ld**: Load the original stack canary value from the C runtime's data segment. In this case, **s0** still contains the pointer, so it does not need to be rebuilt.
3. **xor**: Compare the two values. Exclusive OR'ing two registers has the same effect as subtracting them, in that the result is zero if they are the same (see Exercise 14-1).
4. **li**: Overwrite the stack canary in **a4** with zero, so it is not left lying around.

5. **bnez**: If the values are not equal, then a problem is detected and jump to the **jal** instruction after the **ret** instruction.
6. **ld**: Load **ra** back from the stack. If execution got this far, then **ra** has not been overwritten because the stack canary survived.
7. **ld**: Restore **s0** back from the stack.
8. **li**: Set the return value to zero in **a0**.
9. **add**: Release the stack space by adding 48 to the stack pointer, **sp**.
10. **ret**: Normal subroutine return.
11. **jal**: Call to error reporting routine. This routine terminates the program rather than returning. The call is done this way, since the routine is in the C runtime and the offset in **bnez** is insufficient to jump this far.

Stack canaries are quite effective, but if a hacker discovers the value used in a running process, they can construct a buffer overrun exploit. Plus, the fact that having the process terminate like this is never a good thing.

Preventing Code Running on the Stack

Originally stack overflow exploits would copy a hacker's Assembly Language program as a regular part of the buffer, then overwrite the function's return address to cause this code to execute. Full featured RISC-V CPUs such as the Visionfive and QEMU emulator contain hardware security that mark pages of memory as readable, writable, and/or executable. To prevent code running from the stack Linux removed the

bit allowing code to execute there, making the stack read/write only. With a simple example like this one, it is hard to do without adding a lot of extra compile and link switches to enable stack code execution, since it is firmly off by default.

This does not make executing code on the stack impossible, but it makes it much more difficult, requiring an extra exploit to disable this feature. The other danger is that a shared library disables this feature unknowingly.

This stack memory protection is not available on most microcontrollers such as the ESP32-C3. These lower cost processors simply implement the core integer instruction set with perhaps one or two extensions for things like multiplication and division. This means they are susceptible to code running on the stack. Typically, these processors do not drive user interface programs; however, they do communicate with other devices using serial hardware interfaces. If a hacker gets access to one of these devices, they can replace it with their own device which will insert malicious data into the communications protocol, possibly triggering stack overflow problems and allowing hackers to execute their own code on the stack. This is a reminder that any code that accesses external data must check all external data carefully and be aware of hacking exploits. There have been many successful hacks of IoT devices developed along these lines.

Tradeoffs of Buffer Overflow Mitigation Techniques

Care needs to be taken when designing our APIs to prevent security vulnerabilities. Only use routines that provide some protection against buffer overrun, for example, using **strncpy** over **strcpy**. Enforce this by adding checks to the code check-in process in the source control system. But as pointed out earlier, there are still tradeoffs and weaknesses in these

approaches. Ultimately, the best protection from buffer overruns is to not have them in the first place but beware that no matter how careful a programmer is, mistakes and bugs happen.

Beware of data leakage. If a memory address is included in an error message, then a hacker can use this to determine what the PIE offset is. This might sound unlikely, but there are cases where programmers have a general error reporting mechanism, that includes the contents of all the registers—some of these likely contain memory addresses.

CPU exploits like Spectre and Meltdown show how to access bits of memory contained in the CPU cache. It is unlikely a hacker will find a password this way, but highly likely they will find a memory address or a stack canary.

If every buffer overflow protection technique is turned on and incorporated, then chances are that the code will run as much as fifty percent slower. This might be acceptable in some applications, or parts of applications; however, there are going to be parts of an application that need high performance to be competitive or even usable.

If a section of code needs to be heavily optimized, a layer or module outside of this code is needed that sanitizes and ensures the correctness of the data that is passed to the optimized routine. It needs to be ensured that this data checking cannot be bypassed and that it ensures that the data passes any assumptions in the optimized routines. Code and security reviews can help with this to ensure several sets of eyes have looked for potential problems. The reviewers must have security and hacking expertise, so they know what to look out for.

Note Placing this code in the user interface module is often a mistake. For example, for a web application, the UI is typically written in JavaScript and runs in the browser. Since JavaScript is an interpreted language, hackers can modify the JavaScript to bypass any error checking. Hackers may dispense with JavaScript entirely and send bad messages to the web server. The same is true for all client/server applications. The server must validate all data and not rely on the UI layer.

A weakness with Linux facilities like PIE is that if any shared library linked to disables PIE, then PIE is disabled for the entire application. It is critical to ensure the completed executable still has PIE enabled, otherwise the offending libraries need to be replaced. The same is true for disabling stack execution. There is no good reason to not use PIE, or prevent stack execution, since these do not degrade the performance of the application.

Similarly, if stack canaries are enabled in the code, but the shared libraries being used may not be compiled with this option. Therefore, the code is all protected, but if hackers find a buffer overflow in a routine in a shared library, then they will likely be able to exploit it. Stack canaries are expensive to use, so often programmers use these sparingly or not at all.

Hackers are clever and look for small chinks in an application's armor to exploit. Hackers are patient and if they find one chink, that is not quite enough to use, they keep looking. By combining several bits of information and holes, they can work out how to crack a program's security.

Summary

This chapter was a small glimpse into the world of hacking. This chapter presented how one of the most famous exploits works, namely, exploiting a buffer overrun. Various solutions to the problem were then covered, to make the programs bulletproof. Also, how to fix the code and use the various tools provided by Linux and GNU C was given.

The occurrence of major data breaches at banks, credit agencies, and other online corporate systems happen regularly. Large corporations have the money to hire the best security consultants and use the best tools, yet they are exploited time and again. Take this as a warning to be diligent and conscious of hacking issues in programming.

Having read this far will give a good idea of how to write RISC-V Assembly Language programs for both Linux and microcontrollers. This book gives instructions on how to write basic programs, as well as how to use the floating-point and other extensions to the base instruction set.

Now go forth and experiment. The only way to learn programming is by doing. Think up Assembly Language projects, for example,

1. Control a robot connected to the GPIO pins of either an SBC or microcontroller.
2. Optimize an AI object recognition algorithm with Assembly Language code.
3. Contribute to the RISC-V specific parts of the Linux kernel to improve the operating system's performance.
4. Enhance **GCC** to generate more efficient RISC-V code.
5. Think of something original that might be the next killer application.

Exercises

1. In the discussion of the epilogue code when stack canaries are enabled, it was pointed out that the instruction:

   ```
   xor   a5,a5,a4
   ```

   will set **a5** to zero if **a4** and **a5** are equal. Look up the logic rules for the exclusive or instruction and show how this works.

2. Consider the various APIs for **strcpy**. Choose one for **toupper** and implement it to prevent a buffer overrun.

3. Turn stack canaries for the Chapter 13, "Reading and Understanding Code," **upper.c** program. Play with it to see it working correctly and a stack overrun being caught.

4. Turn on PIE with some of the existing sample programs to ensure they work okay.

5. Does always turning on maximum protection and living with the performance hit the safest approach?

APPENDIX A

The RISC-V Instruction Set

This appendix lists the RISC-V instructions and pseudoinstructions. There is a brief description of each instruction. The instructions are grouped by base group or extension.

RV32I Base Integer Instruction Set

Instruction	Description
add	Add
addi	Add immediate
and	Logical bitwise and
andi	Logical bitwise and immediate
auipc	Add upper immediate to pc
beq	Branch on equal
bge	Branch on greater than or equal
bgeu	Branch on greater than or equal unsigned
blt	Branch on less than

(*continued*)

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

Instruction	Description
bltu	Branch on less than unsigned
bne	Branch on not equal
ebreak	Trigger breakpoint in debugger
ecall	Make operating system call
fence	Fence to order memory and I/O access
jal	Jump and link immediate
jalr	Jump and link register
lb	Load byte
lh	Load half-word
lw	Load word
lui	Load upper immediate
or	Logical bitwise or
ori	Logical bitwise or immediate
sll	Shift left logical
slli	Shift left logical immediate
slt	Set less than
slti	Set less than immediate
sltu	Set less than unsigned
sltiu	Set less than immediate unsigned
sra	Arithmetic shift right
srl	Logical right shift
srli	Logical right shift immediate

(*continued*)

Instruction	Description
sub	Subtract
xor	Logical bitwise exclusive or
xori	Logical bitwise exclusive or immediate

Pseudoinstruction	Description
beqz	Branch if equal to zero
bgez	Branch if greater than or equal to zero
bgt	Branch if greater than
bgtu	Branch if greater than unsigned
bgtz	Branch if greater than zero
ble	Branch if less than or equal
bleu	Branch if less than or equal unsigned
blez	Branch if less than or equal to zero
bltz	Branch if less than zero
bnez	Branch if not equal to zero
call	Call far-away subroutine
fence	Fence all memory and I/O
j	Jump
jal	Jump and link
jalr	Jump and link register
jr	Jump register
li	Load immediate
mv	Copy register to register

(*continued*)

Pseudoinstruction	Description
neg	Two's complement
nop	No operation
not	Bitwise one's complement
ret	Return from subroutine
seqz	Set if equal to zero
sgtz	Set if greater than zero
sltz	Set if less than zero
snez	Set if not equal to zero
tail	Tail call far-away subroutine

RV64I Base Integer Instruction Set—in Addition to RV32I

Instruction	Description
addiw	32-bit add immediate
addw	32-bit add
ld	Load 64-bit value
lwu	Load word unsigned
sd	Store 64-bit value
slli	Shift left logical immediate
slliw	32-bit shift left logical immediate
sllw	32-bit shift left logical

(*continued*)

Instruction	Description
srai	Arithmetic shift right immediate
sraiw	32-bit arithmetic shift right immediate
sraw	32-bit arithmetic shift right
srli	Shift right logical immediate
srliw	32-bit shift right logical immediate
srlw	32-bit shift right logical
subw	32-bit subtract

Pseudoinstruction	Description
negw	32-bit two's complement
sext.w	Sign extend word

RV32M Standard Extension

Instruction	Description
div	Division
divu	Division unsigned
mul	Multiplication—lower half of product
mulh	Multiplication—upper half of product
mulhsu	Upper half for signed x unsigned
mulhu	Upper half for unsigned x unsigned
rem	Remainder
remu	Remainder unsigned

RV64M Standard Extension—in Addition to RV32M

Instruction	Description
divuw	32-bit unsigned division
divw	32-bit division
mulw	32-bit multiplication
remuw	32-bit unsinged remainder
remw	32-bit remainder

RV32F Standard Extension

Instruction	Description
fadd.s	Single-precision floating-point addition
fclass.s	Floating-point classify instruction
fcvt.s.w	Convert floating-point to integer
fcvt.s.wu	Convert floating-point to unsigned integer
fcvt.w.s	Convert integer to single-precision floating-point
fcvt.wu.s	Convert unsigned integer to floating-point
fdiv.s	Single-precision floating-point division
feq.s	Floating-point equality comparison
fle.s	Floating-point less than or equal to comparison

(*continued*)

Instruction	Description
flt.s	Floating-point less than comparison
flw	Floating-point load
fmadd.s	Fused multiply-addition
fmax.s	Single-precision floating-point maximum
fmin.s	Single-precision floating-point minimum
fmsub.s	Fused multiply-subtraction
fmul.s	Single-precision floating-point multiplication
fmv.w.x	Move floating-point register to integer register
fmv.x.w	Move integer register to floating-point register
fnmadd.s	Fused multiply-addition with multiply negated
fnmsub.s	Fused multiply-subtraction with multiply negated
fsgnj.s	Floating-point sign injection
fsgnjn.s	Floating-point sign injection not
fsgnjx.s	Floating-point sign injection xor
fsqrt.s	Single-precision floating-point square root
fsub.s	Single-precision floating-point subtraction
fsw	Floating point save

Pseudoinstruction	Description
fabs.s	Absolute value
fmv.s	Move between floating-point registers
fneg.s	Negative of a single-precision number

RV64F Standard Extension—in Addition to RV32F

Instruction	Description
fcvt.l.s	Convert 64-bit integer to single-precision floating-point
fcvt.lu.s	Convert 64-bit unsigned integer to single-precision floating-point
fcvt.s.l	Convert single-precision floating-point to 64-bit integer
fcvt.s.lu	Convert single-precision floating-point to 64-bit unsigned integer

Pseudoinstruction	Description
fabs.d	Absolute value
fmv.d	Move between floating-point registers
fneg.d	Negative of a double-precision number

RV32D Standard Extension

Instruction	Description
fadd.d	Double-precision floating-point addition
fclass.d	Floating-point classify instruction
fcvt.d.w	Convert floating-point to integer
fcvt.d.wu	Convert floating-point to unsigned integer

(*continued*)

Instruction	Description
fcvt.w.d	Convert integer to double-precision floating-point
fcvt.wu.d	Convert unsigned integer to floating-point
fdiv.d	Double-precision floating-point division
feq.d	Floating-point equality comparison
fle.d	Floating-point less than or equal to comparison
flt.d	Floating-point less than comparison
fld	Floating-point load
fmadd.d	Fused multiply-addition
fmax.d	Double-precision floating-point maximum
fmin.d	Double-precision floating-point minimum
fmsub.d	Fused multiply-subtraction
fmul.d	Double-precision floating-point multiplication
fnmadd.d	Fused multiply-addition with multiply negated
fnmsub.d	Fused multiply-subtraction with multiply negated
fsgnj.d	Floating-point sign injection
fsgnjn.d	Floating-point sign injection not
fsgnjx.d	Floating-point sign injection xor
fsqrt.d	Double-precision floating-point square root
fsub.d	Double-precision floating-point subtraction
fsd	Floating point save

RV64D Standard Extension—in Addition to RV32D

Instruction	Description
fcvt.l.d	Convert 64-bit integer to double-precision floating-point
fcvt.lu.d	Convert 64-bit unsigned integer to double-precision floating-point
fcvt.d.l	Convert double-precision floating-point to 64-bit integer
fcvt.d.lu	Convert double-precision floating-point to 64-bit unsigned integer
fmv.d.x	Move floating-point register to integer register
fmv.x.d	Move integer register to floating-point register

APPENDIX B

Binary Formats

This appendix describes the basic characteristics of the data types we have been working with.

Integers

The following table provides the basic integer data types we have used. Signed integers are represented in two's complement form.

Table B-1. *Size, Alignment, Range, and C Type for the Basic Integer Types*

Size	Type	Alignment in Bytes	Range	C Type
8	Signed	1	–128 to 127	Signed char
8	Unsigned	1	0 to 255	Char
16	Signed	2	–32,768 to 32,767	Short
16	Unsigned	2	0 to 65,535	Unsigned short
32	Signed	4	–2,147,483,648 to 2,147,483,647	Int
32	Unsigned	4	0 to 4,294,967,295	Unsigned int
64	Signed	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Long long
64	Unsigned	8	0 to 18,446,744,073,709,551,615	Unsigned long long

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

Floating Point

The RISC-V floating-point extensions use the IEEE-754 standard for representing floating-point numbers. All floating-point numbers are signed.

Table B-2. *Size, Alignment, Positive Range, and C Type for Floating Point Numbers*

Size	Alignment in Bytes	Range	C Type
32	4	1.175494351e-38 to 3.40282347e+38	Float
64	8	2.22507385850720138e-308 to 1.79769313486231571e+308	Double

Note These ranges are for normalized values; the RISC-V processor will allow floats to become unnormalized to avoid underflow.

Addresses

All addresses or pointers are either 32-bit or 64-bit depending on the processor.

Table B-3. *Size, Range, and C Type of a Pointer*

Size	Range	C Type
32	0 to 4,294,967,295	Void *
64	0 to 18,446,744,073,709,551,615	Void *

APPENDIX C

Assembler Directives

This appendix lists a useful selection of GNU Assembler directives. It includes all the directives used in this book, and a few more that are commonly used.

Directive	Description
.align	Pad the location counter to a particular storage boundary.
.ascii	Defines memory for an ASCII string with no NULL terminator.
.asciz	Defines memory for an ASCII string and adds a NULL terminator.
.byte	Defines memory for bytes.
.data	Assembles following code to the end of the data subsection.
.double	Defines memory for double floating point data.
.dword	Defines storage for 64-bit integers.
.else	Part of conditional assembly.
.elseif	Part of conditional assembly.
.endif	Part of conditional assembly.
.endm	End of a macro definition.
.endr	End of a repeat block.
.equ	Defines values for symbols.
.fill	Defines and fills some memory.

(*continued*)

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

Directive	Description
.float	Define memory for single precision floating point data.
.global	Makes a symbol global, needed if reference from other files.
.hword	Defines memory for 16-bit integers.
.if	Marks the beginning of code to be conditionally assembled.
.include	Merges a file into the current file.
.int	Defines storage for 32-bit integers.
.long	Defines storage for 32-bit integers (same as .int).
.macro	Define a macro.
.octa	Defines storage for 64-bit integers.
.quad	Same as .octa.
.rept	Repeats a block of code multiple times.
.set	Sets the value of a symbol to an expression.
.short	Same as .hword.
.single	Same as .float.
.text	Generates following instructions into the code section.
.word	Same as .int.

APPENDIX D

ASCII Character Set

Here is the ASCII Character Set. The characters from 0 to 127 are standard. The characters from 128 to 255 are taken from: code page 437, which is the character set of the original IBM PC.

Dec	Hex	Char	Description
0	00	NUL	Null
1	01	SOH	Start of Header
2	02	STX	Start of Text
3	03	ETX	End of Text
4	04	EOT	End of Transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal Tab
10	0A	LF	Line Feed
11	0B	VT	Vertical Tab
12	0C	FF	Form Feed
13	0D	CR	Carriage Return

(*continued*)

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

Dec	Hex	Char	Description
14	0E	SO	Shift Out
15	0F	SI	Shift In
16	10	DLE	Data Link Escape
17	11	DC1	Device Control 1
18	12	DC2	Device Control 2
19	13	DC3	Device Control 3
20	14	DC4	Device Control 4
21	15	NAK	Negative Acknowledge
22	16	SYN	Synchronize
23	17	ETB	End of Transmission Block
24	18	CAN	Cancel
25	19	EM	End of Medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File Separator
29	1D	GS	Group Separator
30	1E	RS	Record Separator
31	1F	US	Unit Separator
32	20	space	Space
33	21	!	Exclamation mark
34	22	“	Double quote
35	23	#	Number
36	24	$	Dollar sign

(*continued*)

Dec	Hex	Char	Description
37	25	%	Percent
38	26	&	Ampersand
39	27	'	Single quote
40	28	(	Left parenthesis
41	29	)	Right parenthesis
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Minus
46	2E	.	Period
47	2F	/	Slash
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon

(*continued*)

Dec	Hex	Char	Description
60	3C	<	Less than
61	3D	=	Equality sign
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	At sign
65	41	A	Capital A
66	42	B	Capital B
67	43	C	Capital C
68	44	D	Capital D
69	45	E	Capital E
70	46	F	Capital F
71	47	G	Capital G
72	48	H	Capital H
73	49	I	Capital I
74	4A	J	Capital J
75	4B	K	Capital K
76	4C	L	Capital L
77	4D	M	Capital M
78	4E	N	Capital N
79	4F	O	Capital O
80	50	P	Capital P
81	51	Q	Capital Q
82	52	R	Capital R

(*continued*)

Dec	Hex	Char	Description
83	53	S	Capital S
84	54	T	Capital T
85	55	U	Capital U
86	56	V	Capital V
87	57	W	Capital W
88	58	X	Capital X
89	59	Y	Capital Y
90	5A	Z	Capital Z
91	5B	[	Left square bracket
92	5C	\	Backslash
93	5D	]	Right square bracket
94	5E	^	Caret/circumflex
95	5F	_	Underscore
96	60	`	Grave/accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i

(*continued*)

Dec	Hex	Char	Description
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Left curly bracket
124	7C	|	Vertical bar
125	7D	}	Right curly bracket
126	7E	~	Tilde
127	7F	DEL	Delete
128	80	Ç	

(*continued*)

Dec	Hex	Char	Description
129	81	ü	
130	82	é	
131	83	â	
132	84	ä	
133	85	à	
134	86	å	
135	87	ç	
136	88	ê	
137	89	ë	
138	8A	è	
139	8B	ï	
140	8C	î	
141	8D	ì	
142	8E	Ä	
143	8F	Å	
144	90	É	
145	91	æ	
146	92	Æ	
147	93	ô	
148	94	ö	
149	95	ò	
150	96	û	
151	97	ù	

(*continued*)

Dec	Hex	Char	Description
152	98	ÿ	
153	99	Ö	
154	9A	Ü	
155	9B	¢	
156	9C	£	
157	9D	¥	
158	9E	₧	
159	9F	*ƒ*	
160	A0	á	
161	A1	í	
162	A2	ó	
163	A3	ú	
164	A4	ñ	
165	A5	Ñ	
166	A6	ª	
167	A7	º	
168	A8	¿	
169	A9	⌐	
170	AA	¬	
171	AB	½	
172	AC	¼	
173	AD	¡	
174	AE	«	

(*continued*)

Dec	Hex	Char	Description
175	AF	»	
176	B0	░	
177	B1	▒	
178	B2	▓	
179	B3	│	
180	B4	┤	
181	B5	╡	
182	B6	╢	
183	B7	╖	
184	B8	╕	
185	B9	╣	
186	BA	║	
187	BB	╗	
188	BC	╝	
189	BD	╜	
190	BE	╛	
191	BF	┐	
192	C0	└	
193	C1	┴	
194	C2	┬	
195	C3	├	
196	C4	─	
197	C5	┼	

(*continued*)

Dec	Hex	Char	Description
198	C6	╞	
199	C7	╟	
200	C8	╚	
201	C9	╔	
202	CA	╩	
203	CB	╦	
204	CC	╠	
205	CD	═	
206	CE	╬	
207	CF	╧	
208	D0	╨	
209	D1	╤	
210	D2	╥	
211	D3	╙	
212	D4	╘	
213	D5	╒	
214	D6	╓	
215	D7	╫	
216	D8	╪	
217	D9	┘	
218	DA	┌	
219	DB	█	
220	DC	▄	

(*continued*)

Dec	Hex	Char	Description
221	DD	▌	
222	DE	▐	
223	DF	▀	
224	E0	α	
225	E1	ß	
226	E2	Γ	
227	E3	π	
228	E4	Σ	
229	E5	σ	
230	E6	μ	
231	E7	τ	
232	E8	Φ	
233	E9	Θ	
234	EA	Ω	
235	EB	δ	
236	EC	∞	
237	ED	φ	
238	EE	ε	
239	EF	∩	
240	F0	≡	
241	F1	±	
242	F2	≥	
243	F3	≤	

(*continued*)

Dec	Hex	Char	Description
244	F4	⌠	
245	F5	⌡	
246	F6	÷	
247	F7	≈	
248	F8	°	
249	F9	•	
250	FA	·	
251	FB	√	
252	FC	n	
253	FD	2	
254	FE	■	
255	FF		

APPENDIX E

Answers to Exercises

This appendix has answers to selected exercise. For program code, check the online source code at the Apress GitHub site.

Chapter 2

2-1. 177 (0xb1), 233 (0xe9)

2-2. -14, -125

2-3. 0x118

2-4. 0x218

2-5. addi t2, t1, 15

2-6. 0x007302b3

Chapter 3

3-2. The **simple.S** and **makefile** contained in the source code for this book.

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

Chapter 5

5-1. Look at the **CodeSnippets.S** file in this book's associated source code which contains examples of everything in this chapter.

5-2. These instructions were designed to do this, where the **auipc** instruction provides the upper 20 bits in its immediate argument and then **addi** can add the remaining 12 bits from its immediate argument.

Chapter 6

6-1. It would basically be reversed for instance to push and then pop x5:

```
# push x5 to the stack
addi  sp, sp, 16  # must be a multiple of 16
sd    x5, -8(sp)  # save the value of x5 to
                    the stack

# pop x5 from the  stack
ld    x5, -8(sp)  # load x5
addi  sp, sp, -16 # restore the stack pointer
```

6-2. x9, x20, and x23 need to be save/restored. The following code assumes 64-bit registers

```
# push x9, x20, x23 to the stack
addi   sp, sp, -32 # must be a multiple of 16
sd     x9, 0(sp)
sd     x20, 8(sp)
sd     x23, 16(sp)
```

```
# pop x9, x20 and x23 from the stack
Ld    x23, 16(sp)
ld    x20, 8(sp)
ld    x9, 0(sp)
addi  sp, sp, 32   # restore the stack pointer
```

6-5. This allows clever register usage to avoid frequent pushing and popping to and from the stack.

Chapter 8

8-1. Get/set the IP address, configure various TCP/IP network options like whether you want to receive broadcast packets.

8-2. The main constraint is usually making the electronics inexpensive, and this is done at the expense of ease of programming.

8-3. Any access to physical memory and hardware registers is dangerous and discouraged. Safe access is always through a device driver that enforces Linux security.

Chapter 10

10-5. See the source code associated with this book.

Chapter 12

12-1. **x5** is still shifted for all non-lowercase letters, for example, uppercase letters, punctuation marks, symbols, and numbers; these need to be shifted back in an else clause adding complexity again.

12-2. If using instructions added in a newer version of the RISC-V architecture, then an illegal instruction exception will result if a program is run on any RISC-V processor using an earlier version of the architecture. Do not limit the target audience by eliminating too many customers. However, using new more advanced instructions can give a boost in performance and reduced code size. *Chapter 13* contains an example of this with the **orc.b** instruction from the **Zbb** extension.

Index

A

B

S. Smith, *RISC-V Assembly Language Programming*, Maker Innovations Series,
https://doi.org/10.1007/979-8-8688-0137-2

C

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W, X, Y

Z