TECHNOLOGY IN ACTION™

Programming with 64-Bit ARM Assembly Language

Single Board Computer Development for Raspberry Pi and Mobile Devices

—

Stephen Smith

Apress®

Programming with 64-Bit ARM Assembly Language

Single Board Computer Development for Raspberry Pi and Mobile Devices

Stephen Smith

Apress®

Programming with 64-Bit ARM Assembly Language: Single Board Computer Development for Raspberry Pi and Mobile Devices

Stephen Smith
Gibsons, BC, Canada

ISBN-13 (pbk): 978-1-4842-5880-4 ISBN-13 (electronic): 978-1-4842-5881-1
https://doi.org/10.1007/978-1-4842-5881-1

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

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. 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 rights@apress.com, or visit http://www.apress.com/rights-permissions.

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 via the book's product page, located at www.apress.com/978-1-4842-5880-4. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

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

Table of Contents

About the Author

Stephen Smith is also the author of the Apress title *Raspberry Pi Assembly Language Programming*. He is a retired Software Architect, located in Gibsons, BC, Canada. He's been developing software since high school, or way too many years to record. He was the Chief Architect for the Sage 300 line of accounting products for 23 years. Since retiring, he has pursued artificial intelligence, earned his Advanced HAM Radio License, and enjoys mountain biking, hiking, and nature photography. He continues to write his popular technology blog at smist08.wordpress.com and has written two science fiction novels in the *Influence* series available on Amazon.com.

About the Technical Reviewer

Stewart Watkiss is a keen maker and programmer. He has a master's degree in electronic engineering from the University of Hull and a master's degree in computer science from Georgia Institute of Technology.

He has over 20 years of experience in the IT industry, working in computer networking, Linux system administration, technical support, and cyber security. While working toward Linux certification, he created the web site `www.penguintutor.com`. The web site originally provided information for those studying toward certification but has since added information on electronics, projects, and learning computer programming.

Stewart often gives talks and runs workshops at local Raspberry Pi events. He is also a STEM Ambassador and Code Club volunteer helping to support teachers and children learning programming.

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 folk at Apress who made the whole process easy and enjoyable. A special shout-out to Jessica Vakili, my coordinating editor, who kept the whole project moving quickly and smoothly. Thanks to Aaron Black, senior editor, who recruited me and got the project started. Thanks to Stewart Watkiss, my technical reviewer, who helped make this a far better book.

Introduction

Everyone seems to carry a smartphone and/or a tablet. Nearly all of these devices have one thing in common; they use an ARM central processing unit (CPU). All of these devices are computers just like your laptop or business desktop. The difference is that they need to use less power, in order to function for at least a day on one battery charge, therefore the popularity of the ARM CPU.

At the basic level, how are these computers programmed? What provides the magical foundation for all the great applications (apps) that run on them, yet use far less power than a laptop computer? This book delves into how these are programmed at the bare metal level and provides insight into their architecture.

Assembly Language is the native lowest level way to program a computer. Each processing chip has its own Assembly Language. This book covers programming the ARM 64-bit processor. If you really want to learn how a computer works, learning Assembly Language is a great way to get into the nitty-gritty details. The popularity and low cost of single board computers (SBCs) like the Raspberry Pi and NVidia Jetson Nano provide ideal platforms to learn advanced concepts in computing.

Even though all these devices are low powered and compact, they're still sophisticated computers with a multicore processor, floating-point coprocessor, and a NEON parallel processing unit. What you learn about any one of these is directly relevant to any device with an ARM processor, which by volume is the number one processor on the market today.

In this book, we cover how to program all these devices at the lowest level, operating as close to the hardware as possible. You will learn the following:

- The format of the instructions and how to put them together into programs, as well as details on the binary data formats they operate on
- How to program the floating-point processor, as well as the NEON parallel processor
- About devices running Google's Android, Apple's iOS, and Linux
- How to program the hardware directly using the Raspberry Pi's GPIO ports

The simplest way to learn this is with a Raspberry Pi running a 64-bit flavor of Linux such as Kali Linux. This provides all the tools you need to learn Assembly programming. There's optional material that requires an Apple Mac and iPhone or iPad, as well as optional material that requires an Intel-based computer and an Android device.

This book contains many working programs that you can play with, use as a starting point, or study. The only way to learn programming is by doing, so don't be afraid to experiment, as it is the only way you will learn.

Even if you don't use Assembly programming in your day-to-day life, knowing how the processor works at the Assembly 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 you with your Python programming.

The book is designed to be followed in sequence, but there are chapters that can be skipped or skimmed, for example, if you aren't interested in interfacing to hardware, you can skip Chapter 8, "Programming GPIO Pins," or Chapter 12, "Floating-Point Operations," if you will never do numerical computing.

I hope you enjoy your introduction to Assembly Language. Learning it for one processor family will help you with any other processor architectures you encounter through your career.

Source Code Location

The source code for the example code in the book is located on the Apress GitHub site at the following URL:

https://github.com/Apress/Programming-with-64-Bit-ARM--Assembly-Language

The code is organized by chapter and includes some answers to the programming exercises.

CHAPTER 1

Getting Started

The ARM processor was originally developed by Acorn Computers in Great Britain, who wanted to build a successor to the BBC Microcomputer used for educational purposes. The BBC Microcomputer used the 6502 processor, which was a simple processor with a simple instruction set. The problem was there was no successor to the 6502. The engineers working on the Acorn computer weren't happy with the microprocessors available at the time, since they were much more complicated than the 6502, and they didn't want to make just another IBM PC clone. They took the bold move to design their own and founded Advanced RISC Machines Ltd. to do it. They developed the Acorn computer and tried to position it as the successor to the BBC Microcomputer. The idea was to use reduced instruction set computer (RISC) technology as opposed to complex instruction set computer (CISC) as championed by Intel and Motorola. We will talk at length about what these terms mean later.

Developing silicon chips is costly, and without high volumes, manufacturing them is expensive. The ARM processor probably wouldn't have gone anywhere except that Apple came calling. They were looking for a processor for a new device under development—the iPod. The key selling point for Apple was that as the ARM processor was RISC, it used less silicon than CISC processors and as a result used far less power. This meant it was possible to build a device that ran for a long time on a single battery charge.

S. Smith, *Programming with 64-Bit ARM Assembly Language*,
https://doi.org/10.1007/978-1-4842-5881-1_1

The Surprise Birth of the 64-Bit ARM

The early iPhones and Android phones were all based on 32-bit ARM processors. At that time, even though most server and desktop operating systems moved to 64 bits, it was believed that there was no need in the mobile world for 64 bits. Then in 2013, Apple shocked the ARM world by introducing the 64-bit capable A7 chip and started the migration of all iOS programs to 64 bits. The performance gains astonished everyone and caught all their competitors flat footed. Now, all newer ARM processors support 64-bit processing, and all the major ARM operating systems have moved to 64 bits.

Two benefits of ARM 64-bit programming are that ARM cleaned up their instruction set and simplified Assembly Language programming. They also adapted the code, so that it will run more efficiently on modern processors with larger execution pipelines. There are still a lot of details and complexities to master, but if you have experience in 32-bit ARM, you will find 64-bit programming simpler and more consistent.

However, there is still a need for 32-bit processing, for instance, Raspbian, the default operating system for the Raspberry Pi, is 32 bits, along with several real-time and embedded systems. If you have 1GB of memory or less, 32 bits is better, but once you have more than 1GB of RAM, then the benefits of 64-bit programming become hard to ignore.

Unlike Intel, ARM doesn't manufacture chips; it just licenses the designs for others to optimize and manufacture. With Apple onboard, suddenly there was a lot of interest in ARM, and several big manufacturers started producing chips. With the advent of smartphones, the ARM chip really took off and now is used in pretty much every phone and tablet. ARM processors power some Chromebooks and even Microsoft's Surface Pro X.

The ARM processor is the number one processor in the computer market. Each year the ARM processors powering the leading-edge phones become more and more powerful. We are starting to see ARM-based servers used in datacenters, including Amazon's AWS. There are several ARM-based laptops and desktop computers in the works.

What You Will Learn

You will learn Assembly Language programming for the ARM running in 64-bit mode. Everything you will learn is directly applicable to all ARM devices running in 64-bit mode. Learning Assembly Language for one processor gives you the tools to learn it for another processor, perhaps, the forthcoming RISC-V, a new open source RISC processor that originated from Berkeley University. The RISC-V architecture promises high functionality and speed for less power and cost than an equivalent ARM processor.

In all devices, the ARM processor isn't just a CPU; it's a system on a chip. This means that most of the computer is 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 an ARM processor with multiple cores, meaning that it can process instructions for multiple programs running at once. It likely contains several coprocessors for things like floating-point calculations, a graphics processing unit (GPU), and specialized multimedia support. There are extensions available for cryptography, advanced virtualization, and security monitoring.

Why Use Assembly

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 web sites like Facebook, to productivity software like LibreOffice. If you learn to be a good programmer in a couple of these, you can 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 is harder, as you must solve problems with memory addressing and CPU registers that is all handled transparently by high-level languages. So why would you want to learn Assembly Language programming? Here are ten reasons people 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 coprocessors, 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 and handles hardware security and multithreading/multitasking 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 your computer run faster**: The best way to make Linux faster is to improve the GNU C compiler. If you improve the ARM 64-bit Assembly code produced by GNU C, then every program compiled by GCC benefits.

5. **To interface your computer to a hardware device**: When interfacing your computer through USB or GPIO ports, the speed of data transfer is highly sensitive as to how fast your program can process the data. Perhaps, there are a lot of bit level manipulations that are easier to program in Assembly.

6. **To do faster machine learning or three-dimensional (3D) graphics programming**: Both applications rely on fast matrix mathematics. If you can make this faster with Assembly and/or using the coprocessors, then you can make your AI-based robot or video game that much better.

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

8. **To manage single board computer competitors to the Raspberry Pi**: These 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.

Tools You Need

The best way to learn programming is by doing. The easiest way to play with 64-bit ARM Assembly Language is with an inexpensive single board computer (SBC) like the Raspberry Pi or NVidia Jetson Nano. We will cover developing for Android and iOS, but these sections are optional. In addition to a computer, you will need

- A text editor
- Some optional specialty programs

Raspberry Pi 4 or NVidia Jetson Nano

The Raspberry Pi 4 with 4GB of RAM is an excellent computer to run 64-bit Linux. If you use a Raspberry Pi 4, then you need to download and install a 64-bit version of Linux. These are available from Kali, Ubuntu, Gentoo, Manjaro, and others. I find Kali Linux works very well and will be using it to test all the programs in this book. You can find the Kali Linux downloads here: `www.offensive-security.com/kali-linux-arm-images/`.

Although you can run 64-bit Linux on a Raspberry Pi 3 or a Raspberry Pi 4 with 1GB of RAM, I find these slow and bog down if you run too many programs. I wouldn't recommend these, but you can use them in a pinch.

The NVidia Jetson Nano uses 64-bit Ubuntu Linux. This is an excellent platform for learning ARM 64-bit Assembly Language. The Jetson Nano also has 128 CUDA graphics processing cores that you can play with.

One of the great things about the Linux operating system is that it is intended to be used for programming and as a result has many programming tools preinstalled, including

- GNU Compiler Collection (GCC) that we will use to build our Assembly Language programs. We will use GCC for compiling C programs in later chapters.
- GNU Make to build our programs.
- GNU Debugger (GDB) to find and solve problems in our programs.

Text Editor

You will need a text editor to create the source program files. Any text editor can be used. Linux usually includes several by default, both command line and via the GUI. Usually, you learn Assembly Language after you've already mastered a high-level language like C or Java. So, chances are you already have a favorite editor and can continue to use it.

Specialty Programs

We will mention other helpful programs throughout the book that you can optionally use, but aren't required, for example:

- The Android SDK
- Apple's XCode IDE

- A better code analysis tool, like Ghidra, which we will discuss in Chapter 15, "Reading and Understanding Code"

All of these are either open source or free, but there may be some restrictions on where you can install them.

Now we will switch gears to how computers represent numbers. We always hear that computers only deal in zeros and ones; now we'll look at how they put them together to represent larger numbers.

Computers and Numbers

We typically represent numbers using base 10. 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

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

There is nothing special about using 10 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; rather, everything is a switch that is either on or off. As a result, computers are programmed to use base 2 arithmetic. Thus, a computer recognizes a number like 1011 as

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

This is extremely efficient for computers, but we are using four digits for the decimal number 11 rather than two digits. The big disadvantage for humans is that writing, or even keyboarding, binary numbers is tiring.

Computers are incredibly structured, with their numbers being the same size in storage used. When designing computers, it doesn't make sense to have different sized numbers, so a few common sizes have taken hold and become standard.

A byte is 8 binary bits or digits. In our preceding example with 4 bits, there are 16 possible combinations of 0s and 1s. This means 4 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 0–9 and then the letters A–F for 10–15. We can then represent a byte (8 bits) as two base 16 digits. We refer to base 16 numbers as hexadecimal (Figure 1-1).

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

Figure 1-1. *Representing hexadecimal digits*

Since a byte holds 8 bits, it can represent 2^8 (256) numbers. Thus, the byte e6 represents

$$\begin{aligned} e6 &= e * 16^1 + 6 * 16^0 \\ &= 14 * 16 + 6 \\ &= 230 \text{ (decimal)} \\ &= 1110\ 0110 \text{ (binary)} \end{aligned}$$

We call a 32-bit quantity a word and it is represented by 4 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 the contents of one register. Even though we are running 64 bits, the ARM reference documentation refers to a word as 32 bits, a halfword is 16 bits, and a doubleword is 64 bits. We will see this terminology throughout this book and the ARM documentation.

If this is confusing or scary, don't worry. The tools will do all the conversions for you. It's just a matter of understanding what is presented to you on screen. Also, if you need to specify an exact binary number, usually you do so in hexadecimal, although all the tools accept all the formats.