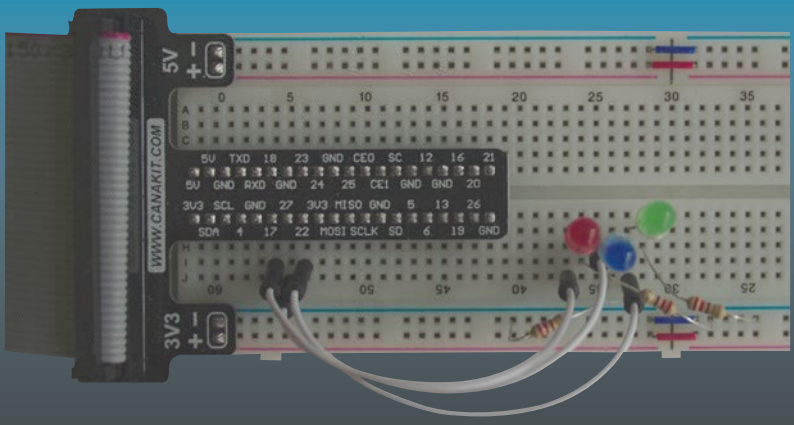
TECHNOLOGY IN ACTION™



Raspberry Pi Assembly Language Programming

ARM Processor Coding

—

Stephen Smith

Apress®

Raspberry Pi Assembly Language Programming

ARM Processor Coding

Stephen Smith

Apress®

Raspberry Pi Assembly Language Programming: ARM Processor Coding

Stephen Smith
Gibsons, BC, Canada

ISBN-13 (pbk): 978-1-4842-5286-4 ISBN-13 (electronic): 978-1-4842-5287-1
https://doi.org/10.1007/978-1-4842-5287-1

Copyright © 2019 by Stephen Smith

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

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/978-1-4842-5286-4. 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 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 worked on 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 a series, *Influence*, available on Amazon.com.

About the Technical Reviewer

Stewart Watkiss is a keen maker, programmer, and author of *Learn Electronics with Raspberry Pi*. He studied at the University of Hull, where he earned a master's degree in electronic 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 computer to school children and at Raspberry Pi events. He has created a number of resources using Pygame Zero, which he makes available on his web site (`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 Jessica Vakili, my coordinating editor, who kept the whole project moving quickly and smoothly. Thanks to Aaron Black, the 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. Thanks to James Markham, my development editor, for all his good work keeping me to standards.

Introduction

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 the Raspberry Pi provide an ideal platform to learn advanced concepts in computing.

Even though the Raspberry Pi is inexpensive and credit card sized, it is still a sophisticated computer with a quad-core processor, a floating-point coprocessor, and a NEON parallel processing unit. What you learn about the Raspberry Pi is directly relevant to any device with an ARM processor, which includes nearly every cell phone and tablet. In fact, by volume, the ARM processor is the number one processor today.

In this book, we will cover how you program the Raspberry Pi at the lowest level; you will be operating as close to the hardware as possible. We will teach the format of the instructions, how to put them together into programs as well as details on the binary data formats they operate on. We will cover how to program the floating-point processor as well as the NEON parallel processor. We cover how to program the GPIO ports to interface to custom hardware, so you can experiment with electronics connected to your Raspberry Pi.

All you need is a Raspberry Pi running Raspbian. This will provide all the tools you need to learn Assembly programming. This is the low cost of entry of running open source software like Raspbian Linux and the GNU Assembler. The last chapter covers 64-bit programming, where you will need to run Ubuntu MATE on your Pi.

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; 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 instance, if you aren't interested in interfacing to hardware, you can pass on Chapter 8, "Programming GPIO Pins," or Chapter 11, "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.

CHAPTER 1

Getting Started

The Raspberry Pi is a credit card–sized computer that costs only US$35. It was originally developed to provide low-cost computers to schools and children, who couldn't afford regular PCs or Macs. Since its release, the Raspberry Pi has been incredibly successful—as of this writing, selling over 25 million units. The Raspberry Pi has become the basis of a whole DIY movement with diverse applications, including home automation control systems, acting as the brain for robots, or linked together to build a personal supercomputer. The Pi is also a great educational tool.

This book will leverage the Raspberry Pi to assist you in learning Assembly language. Programming in Assembly language is programming your computer at the lowest bits and bytes level. People usually program computers in high-level programming languages, like Python, C, Java, C#, or JavaScript. The tools that accompany these languages convert your program to Assembly language, whether they do it all at once or as they run.

Assembly language is specific to the computer processor used. Since we are learning for the Raspberry Pi, we will learn Assembly language for the Advanced RISC Machine (ARM) processor. We will use the Raspbian operating system, a 32-bit operating system based on Debian Linux, so we will learn 32-bit Assembly on the Raspberry Pi's ARM processor.

The Raspberry Pi 3 has an ARM processor that can operate in 64-bit mode, but Raspbian doesn't do that. We will highlight some important differences between 32-bit and 64-bit Assembly, but all our sample programs will be in 32-bit ARM Assembler and will be compiled to run under Raspbian.

© Stephen Smith 2019
S. Smith, *Raspberry Pi Assembly Language Programming*,
https://doi.org/10.1007/978-1-4842-5287-1_1

About the ARM Processor

The ARM processor was originally developed by a group 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. They weren't happy with the microprocessors that were around at the time, since they were much more complicated than the 6502 and they didn't want to make another IBM PC clone. They took the bold move to design their own. They developed the Acorn computer that used it 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 talk at length about what these terms really mean later.

Developing silicon chips is an expensive proposition, and unless you can get a good volume going, manufacturing is expensive. The ARM processor probably wouldn't have gone anywhere except that Apple came calling looking for a processor for a new device they had 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.

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 even power some Chromebooks. The ARM processor is the number one processor in the computer market.

What You Will Learn

You will learn Assembly language programming for the ARM processor on the Raspberry Pi, but everything you learn is directly applicable to all these other devices. Learning Assembly language for one processor gives you the tools to learn it for another processor, perhaps, the forthcoming RISC-V.

The chip that is the brains of the Raspberry Pi isn't just a processor, it is also a system on a chip. This means that most of the computer is all on one chip. This chip contains an ARM quad-core processor, meaning that it can process instructions for four programs running at once. It also contains several coprocessors for things like floating-point calculations, a graphics processing unit (GPU) and specialized multimedia support.

ARM does a good job at supporting coprocessors and allowing manufacturers to build their chips in a modular manner incorporating the elements they need. All Raspberry Pi include a floating-point coprocessor (**FPU**). Newer Raspberry Pi have advanced capabilities such as NEON parallel processors. Table 1-1 gives an overview of the units we will be programming and which Raspberry Pi support them. In Table 1-1, **SoC** is system on a chip and contains the Broadcom part number for the unit incorporated.

Table 1-1. *Common Raspberry Pi models and their capabilities relevant to this book*

Model	SoC	Memory	Divide instruction	FPU	NEON coprocessor	64-Bit support
Pi A+	BCM2835	256 MB		v2		
Pi B	BCM2835	512 MB		v2		
Pi Zero	BCM2835	512 MB		v2		
Pi 2	BCM2836	1 GB	Yes	v3	Yes	Yes
Pi 3	BCM2837	1 GB	Yes	v4	Yes	Yes
Pi 3+	BCM2837B0	1 GB	Yes	v4	Yes	Yes
Pi 4	BCM2711	1, 2, or 4 GB	Yes	v4	Yes	Yes

Why Use Assembly

Most programmers today write in a high-level programming language like Python, C#, Java, JavaScript, Go, Julia, Scratch, Ruby, Swift, or C. These are highly productive languages that 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 multiple 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 ever want to learn Assembly language programming? Here are ten reasons people learn and use Assembly language:

1. Even if you don't write Assembly language code, knowing how the computer works internally allows you to write more efficient code. You can make your data structures easier to access and write code in a style that allows the compiler to generate more efficient 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 very 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. You want to make the Raspberry Pi faster. The best way to make Raspbian faster is to improve the GNU C compiler. If you improve the ARM 32-bit Assembly code produced by GNU C, then every Linux program compiled for the Pi benefits.

5. You might be interfacing your Pi to a hardware device, either through USB or the GPIO ports, and the speed of data transfer is highly sensitive 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 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. 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. Perhaps you work for a hardware company that makes a single board computer competitor 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. You usually need to 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 enabled by finding a secret keyboard combination to pop them up. Finding them requires reverse engineering the program and reading Assembly language.

Tools You Need

This book is designed so that all you need is a Raspberry Pi that runs the Raspbian operating system. Raspbian is based on Debian Linux, so anything you know about Linux is directly useful. There are other operating systems for the Pi, but we will only cover Raspbian in this book.

A Raspberry Pi 3, either the B or B+ model, is ideal. Most of what is in this book runs on older models as well, as the differences are largely in the coprocessor units and the amount of memory. We will talk about how to develop programs to run on the compact A models and the Raspberry Pi Zero, but you wouldn't want to develop your programs directly on these.

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

- GNC 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.

You will need a text editor to create the source program files. Any text editor can be used. Raspbian 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.

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

- A better programmer's calculator
- A better code analysis tool

All of these are open source and you can install them for free.

Now we are going to 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 10 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: 10 fingers and 10 toes.

Computers don't have fingers and toes, and in their world, everything is a switch that is either on or off. As a result, it is natural for computers

to use base 2 arithmetic. Thus, to a computer a number like 1011 is represented by

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

This is great for computers, but we are using 4 digits for the decimal number 11 rather than 2 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 all sorts of 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 to 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 28 (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 are running the ARM processor in 32-bit mode; 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 perhaps the contents of one register.

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, though all the tools accept all the formats.

A handy tool is the Linux Gnome calculator (Figure 1-2). The calculator included with Raspbian can perform math in different bases in its scientific mode, but the Gnome calculator has a nicer Programming Mode which shows a numbers representation in multiple bases at once. To install it, use the command line

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

Run it from the Accessories menu (probably the second calculator there). If you put it in "Programming Mode," you can do the conversions and it shows you numbers in several formats at once.

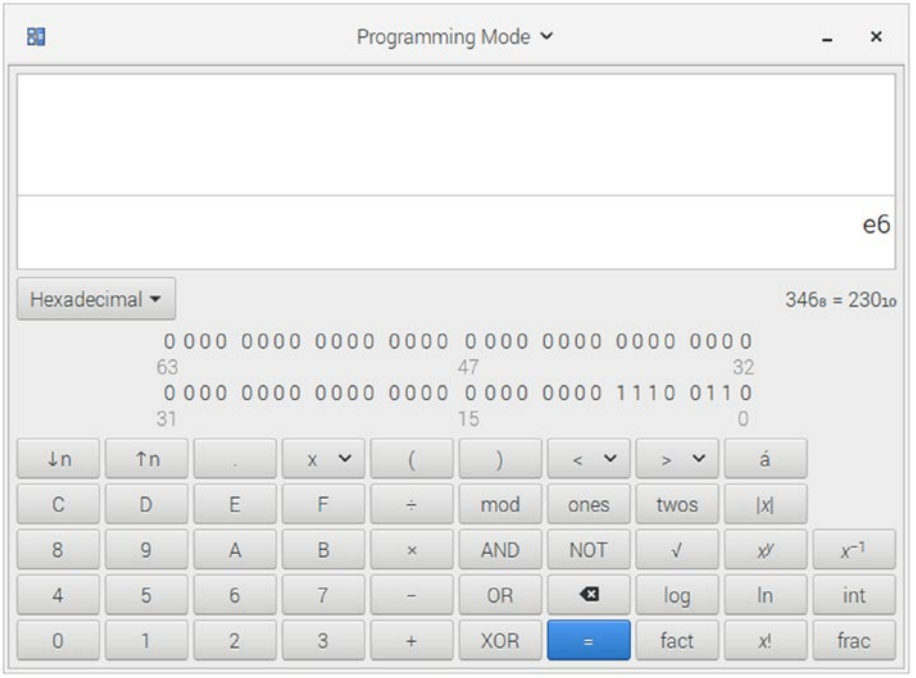


Figure 1-2. *The Gnome calculator*

This is how we represent computer memory. There is a bit more complexity in how signed integers are represented and how arithmetic works. We'll cover that a bit later when we go to do some arithmetic.

In the Assembler we represent hexadecimal numbers (hex for short) with a 0x in front. So 0x1B is how we would specify the hex number 1B.

ARM Assembly Instructions

In this section, we introduce some basic architectural elements of the ARM processor and start to look at the form of its machine code instructions. The ARM is what is called a Reduced Instruction Set Computer (RISC), which theoretically will make learning Assembly easier. There are fewer instructions and each instruction is simpler, so the processor can execute

each instruction much quicker. While this is true, the ARM system on a chip used in the Raspberry Pi is a highly sophisticated computer. The core ARM processors handle multiple instruction sets, and then there are theinstruction sets for all the coprocessors.

Our approach to this is to divide and conquer. In the first few chapters of this book, we will cover only the 32-bit standard ARM Assembly instructions. This means that the following topics are deferred to later chapters where they can be covered in detail without introducing too much confusion:

- Instructions for the floating-point processor
- Instructions for the NEON processor
- Instructions for 64 bits
- Thumb mode instructions (special 16-bit compact mode)

In this manner, we just need to attack one topic at a time. Each set of instructions is consistent and easy to understand.

In technical computer topics, there are often chicken and egg problems in presenting the material. The purpose of this section is to introduce all the terms and ideas we will use later. Hopefully, this introduces all the terms, so they are familiar when we cover them in full detail.

CPU Registers

In all computers, data is not operated in the computer's memory; instead, it is loaded into a CPU register, then the data processing or arithmetic operation is performed in 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.