BLUE FOX EDITION

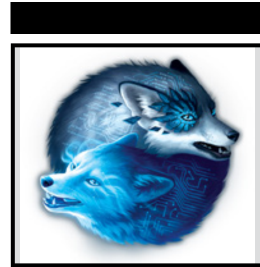
ARM ASSEMBLY INTERNALS & REVERSE ENGINEERING

MARIA AZERIA MARKSTEDTER

WILEY

Blue Fox

Arm Assembly Internals & Reverse Engineering



Blue Fox

Arm Assembly Internals & Reverse Engineering

Maria Markstedter

WILEY

Copyright © 2023 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBN: 978-1-119-74530-3
ISBN: 978-1-119-74673-7 (ebk)
ISBN: 978-1-119-74672-0 (ebk)

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at www.wiley.com/go/permission.

Trademarks: WILEY and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

If you believe you've found a mistake in this book, please bring it to our attention by emailing our Reader Support team at wileysupport@wiley.com with the subject line "Possible Book Errata Submission."

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data: 2023933796

Cover illustration: © Jonas Jödicke
Cover design: Maria Markstedter and Wiley

To my mother, who made countless sacrifices to provide me with the opportunities that enabled me to pursue my dreams.

About the Author

Maria Markstedter is the founder and CEO of Azeria Labs, which provides training courses on Arm reverse engineering and exploitation. Previously, she worked in the fields of pentesting and threat intelligence and served as the chief product officer of the virtualization startup Corellium, Inc.

She has a bachelor's degree in corporate security and a master's degree in enterprise security and worked on exploit mitigation research alongside Arm in Cambridge.

Maria has been recognized for her contributions to the field, having been selected for Forbes' "30 under 30" list for technology in Europe in 2018 and named Forbes Person of the Year in Cybersecurity in 2020. She has also been a member of the Black Hat® EU and US Trainings and Briefings Review Board since 2017.

Acknowledgments

First and foremost, I would like to thank my technical reviewers for spending endless hours patiently reviewing every chapter.

- **Daniel Cuthbert**, who has always been a great friend, supporter, and the best mentor I could ask for
- **Jon Masters**, an Arm genius whose technical knowledge has always inspired me
- **Maddie Stone**, who is a brilliant security researcher and a wonderful person I look up to
- **Matthias Boettcher**, who patiently served as supervisor for my master's thesis at Arm and became a valuable technical reviewer for this book

Thanks to Patrick Wardle for contributing the malware analysis chapter (Chapter 12, "Reversing arm64 macOS Malware") to this book.

Thanks to my editors, Jim Minatel and Kelly Talbot, for pushing me to complete this book during the pandemic and for being so patient with my insufferable perfectionism.

I would also like to thank Runa Sandvik for being the best friend anyone could ask for and for giving me strength and support in difficult times.

Most important, I want to thank all the readers for putting their faith in me.

— Maria Markstedter

Contents at a Glance

Introduction		xxi
Part I	**Arm Assembly Internals**	**1**
Chapter 1	**Introduction to Reverse Engineering**	**3**
Chapter 2	**ELF File Format Internals**	**21**
Chapter 3	**OS Fundamentals**	**69**
Chapter 4	**The Arm Architecture**	**93**
Chapter 5	**Data Processing Instructions**	**129**
Chapter 6	**Memory Access Instructions**	**195**
Chapter 7	**Conditional Execution**	**243**
Chapter 8	**Control Flow**	**275**
Part II	**Reverse Engineering**	**305**
Chapter 9	**Arm Environments**	**307**
Chapter 10	**Static Analysis**	**321**
Chapter 11	**Dynamic Analysis**	**363**
Chapter 12	**Reversing arm64 macOS Malware**	**405**
Index		437

Contents

Introduction		**xxi**
Part I	**Arm Assembly Internals**	**1**
Chapter 1	**Introduction to Reverse Engineering**	**3**
	Introduction to Assembly	3
	Bits and Bytes	3
	Character Encoding	5
	Machine Code and Assembly	6
	Assembling	9
	Cross-Assemblers	13
	High-Level Languages	15
	Disassembling	16
	Decompilation	17
Chapter 2	**ELF File Format Internals**	**21**
	Program Structure	21
	High-Level vs. Low-Level Languages	22
	The Compilation Process	24
	Cross-Compiling for Other Architectures	25
	Assembling and Linking	27
	The ELF File Overview	30
	The ELF File Header	31
	The ELF File Header Information Fields	32
	The Target Platform Fields	33
	The Entry Point Field	34
	The Table Location Fields	34
	ELF Program Headers	34
	The PHDR Program Header	36
	The INTERP Program Header	36

The LOAD Program Headers 36
The DYNAMIC Program Header 37
The NOTE Program Header 37
The TLS Program Header 38
The GNU_EH_FRAME Program Header 38
The GNU_STACK Program Header 39
The GNU_RELRO Program Header 41
ELF Section Headers 43
The ELF Meta-Sections 45
The String Table Section 46
The Symbol Table Section 46
The Main ELF Sections 46
The .text Section 47
The .data Section 47
The .bss Section 47
The .rodata Section 47
The .tdata and .tbss Sections 48
Symbols 48
Global vs. Local Symbols 50
Weak Symbols 50
Symbol Versions 51
Mapping Symbols 51
The Dynamic Section and Dynamic Loading 52
Dependency Loading (NEEDED) 53
Program Relocations 54
Static Relocations 55
Dynamic Relocations 56
The Global Offset Table (GOT) 57
The Procedure Linkage Table (PLT) 57
The ELF Program Initialization and Termination Sections 58
Initialization and Termination Order 60
Thread-Local Storage 60
The Local-Exec TLS Access Model 65
The Initial-Exec TLS Access Model 65
The General-Dynamic TLS Access Model 66
The Local-Dynamic TLS Access Model 67

Chapter 3 OS Fundamentals 69
OS Architecture Overview 69
User Mode vs. Kernel Mode 70
Processes 70
System Calls 72
Objects and Handles 77
Threads 79
Process Memory Management 80

Memory Pages 82
Memory Protections 82
Anonymous and Memory-Mapped Memory 84
Memory-Mapped Files and Modules 84
Address Space Layout Randomization 87
Stack Implementations 90
Shared Memory 91

Chapter 4 The Arm Architecture 93
Architectures and Profiles 93
The Armv8-A Architecture 95
Exception Levels 96
Armv8-A TrustZone Extension 97
Exception Level Changes 99
Armv8-A Execution States 101
The AArch64 Execution State 102
The A64 Instruction Set 103
AArch64 Registers 104
The Program Counter 106
The Stack Pointer 107
The Zero Register 107
The Link Register 108
The Frame Pointer 109
The Platform Register (x18) 109
The Intraprocedural Call Registers 110
SIMD and Floating-Point Registers 110
System Registers 111
PSTATE 112
The AArch32 Execution State 114
A32 and T32 Instruction Sets 114
The A32 Instruction Set 114
The T32 Instruction Set 115
Switching Between Instruction Sets 115
AArch32 Registers 118
The Program Counter 119
The Stack Pointer 120
The Frame Pointer 120
The Link Register 121
The Intraprocedural Call Register (IP, r12) 121
The Current Program Status Register 121
The Application Program Status Register 122
The Execution State Registers 124
The Instruction Set State Register 124
The IT Block State Register (ITSTATE) 125

Endianness state 126
Mode and Exception Mask Bits 126

Chapter 5 Data Processing Instructions 129

Shift and Rotate Operations 131
Logical Shift Left 132
Logical Shift Right 133
Arithmetic Shift Right 133
Rotate Right 134
Rotate Right with Extend 134
Instruction Forms 135
Shift by a Constant Immediate Form 136
Shift by Register Form 138
Bitfield Manipulation Operations 140
Bitfield Move 141
Sign- and Zero-Extend Operations 145
Bitfield Extract and Insert 150
Logical Operations 153
Bitwise *AND* 153
The TST Instruction 154
Bitwise Bit Clear 155
Bitwise OR 155
Bitwise OR NOT 156
Bitwise Exclusive OR 158
The TEQ instruction 158
Exclusive OR NOT 159
Arithmetic Operations 159
Addition and Subtraction 159
Reverse Subtract 161
Compare 162
CMP Instruction Operation Behavior 163
Multiplication Operations 165
Multiplications on A64 166
Multiplications on A32/T32 167
Least Significant Word Multiplications 169
Most Significant Word Multiplications 171
Halfword Multiplications 173
Vector (Dual) Multiplications 176
Long (64-Bit) Multiplications 179
Division Operations 186
Move Operations 187
Move Constant Immediate 188
Move Immediate and MOVT on A32/T32 188
Move Immediate, MOVZ, and MOVK on A64 189
Move Register 190
Move with NOT 192

Chapter 6 Memory Access Instructions 195
Instructions Overview 195
Addressing Modes and Offset Forms 197
Offset Addressing 200
Constant Immediate Offset 201
Register Offsets 207
Pre-Indexed Mode 209
Pre-Indexed Mode Example 210
Post-Indexed Addressing 212
Post-Indexed Addressing Example 213
Literal (PC-Relative) Addressing 214
Loading Constants 215
Loading an Address into a Register 218
Load and Store Instructions 222
Load and Store Word or Doubleword 222
Load and Store Halfword or Byte 224
Example Using Load and Store 226
Load and Store Multiple (A32) 228
Example for STM and LDM 235
A More Complicated Example Using STM and LDM 237
Load and Store Pair (A64) 238

Chapter 7 Conditional Execution 243
Conditional Execution Overview 243
Conditional Codes 244
The NZCV Condition Flags 245
Signed vs. Unsigned Integer Overflows 246
Condition Codes 248
Conditional Instructions 249
The If-Then (IT) Instruction in Thumb 250
Flag-Setting Instructions 252
The Instruction "S" Suffix 253
The S Suffix on Add and Subtract Instructions 253
The S Suffix on Logical Shift Instructions 256
The S Suffix on Multiply Instructions 257
The S Suffix on Other Instructions 257
Test and Comparison Instructions 257
Compare (CMP) 258
Compare Negative (CMN) 260
Test Bits (TST) 261
Test Equality (TEQ) 264
Conditional Select Instructions 265
Conditional Comparison Instructions 268
Boolean AND Conditionals Using CCMP 269
Boolean OR Conditionals Using CCMP 272

Chapter 8	**Control Flow**	**275**
	Branch Instructions	275
	Conditional Branches and Loops	277
	Test and Compare Branches	281
	Table Branches (T32)	282
	Branch and Exchange	284
	Subroutine Branches	288
	Functions and Subroutines	290
	The Procedure Call Standard	291
	Volatile vs. Nonvolatile Registers	293
	Arguments and Return Values	293
	Passing Larger Values	295
	Leaf and Nonleaf Functions	298
	Leaf Functions	298
	Nonleaf Functions	299
	Prologue and Epilogue	299
Part II	**Reverse Engineering**	**305**
Chapter 9	**Arm Environments**	**307**
	Arm Boards	308
	Emulation with QEMU	310
	QEMU User-Mode Emulation	310
	QEMU Full-System Emulation	314
	Firmware Emulation	315
Chapter 10	**Static Analysis**	**321**
	Static Analysis Tools	322
	Command-Line Tools	322
	Disassemblers and Decompilers	322
	Binary Ninja Cloud	323
	Call-By-Reference Example	328
	Control Flow Analysis	334
	Main Function	336
	Subroutine	336
	Converting to char	341
	if Statement	343
	Quotient Division	345
	for Loop	347
	Analyzing an Algorithm	349
Chapter 11	**Dynamic Analysis**	**363**
	Command-Line Debugging	364
	GDB Commands	365
	GDB Multiuser	366
	GDB Extension: GEF	368
	Installation	369
	Interface	370

Useful GEF Commands 370
Examine Memory 374
Watch Memory Regions 376
Vulnerability Analyzers 377
checksec 379
Radare2 381
Debugging 382
Remote Debugging 385
Radare2 386
IDA Pro 388
Debugging a Memory Corruption 390
Debugging a Process with GDB 398

Chapter 12 Reversing arm64 macOS Malware 405
Background 406
macOS arm64 Binaries 407
macOS Hello World (arm64) 410
Hunting for Malicious arm64 Binaries 413
Analyzing arm64 Malware 419
Anti-Analysis Techniques 420
Anti-Debugging Logic (via ptrace) 421
Anti-Debugging Logic (via sysctl) 425
Anti-VM Logic (via SIP Status and the Detection of VM Artifacts) 429
Conclusion 435

Index 437

Introduction

Let's address the elephant in the room: why "Blue Fox"?

This book was originally supposed to contain an overview of the Arm instruction set, chapters on reverse engineering, and chapters on exploit mitigation internals and bypass techniques. The publisher and I soon realized that covering these topics to a satisfactory extent would make this book about 1,000 pages long. For this reason, we decided to split it into two books: Blue Fox and Red Fox.

The Blue Fox edition covers the analyst view; teaching you everything you need to know to get started in reverse engineering. Without a solid understanding of the fundamentals, you can't move to more advanced topics such as vulnerability analysis and exploit development. The Red Fox edition will cover the offensive security view: understanding exploit mitigation internals, bypass techniques, and common vulnerability patterns.

As of this writing, the Arm architecture reference manual for the Armv8-A architecture (and Armv9-A extensions) contains 11,952 pages[1] and continues to expand. This reference manual was around 8,000 pages[2] long when I started writing this book two years ago.

Security researchers who are used to reverse engineering x86/64 binaries but want to adopt to the new era of Arm-powered devices are having a hard time finding digestible resources on the Arm instruction set, especially in the context of reverse engineering or binary analysis. Arm's architecture reference manual can be both overwhelming and discouraging. In this day and age, nobody has time to read a 12,000-page deeply technical document, let alone identify

[1] (version I.a.) https://developer.arm.com/documentation/ddi0487/latest
[2] (version F.a.) https://developer.arm.com/documentation/ddi0487/latest

the most relevant or most commonly used instructions and memorize them. The truth is that you don't need to know every single Arm instruction to be able to reverse engineer an Arm binary. Many instructions have very specific use cases that you may or may not ever encounter during your analysis.

The purpose of this book is to make it easier for people to get familiar with the Arm instruction set and gain enough knowledge to apply it in their professional lives. I spent countless hours dissecting the Arm reference manual and categorizing the most common instruction types and their syntax patterns so you don't have to. But this book isn't a list of the most common Arm instructions. It contains explanations you won't find anywhere else, not even in the Arm manual itself. The basic descriptions of a given instruction in the Arm manual are rather brief. That is fine for trivial instructions like `MOV` or `ADD`. However, many common instructions perform complex operations that are difficult to understand from their descriptions alone. For this reason, many of the instructions you will encounter in this book are accompanied by graphical illustrations explaining what is actually happening under the hood.

If you're a beginner in reverse engineering, it is important to understand the binary's file format, its sections, how it compiles from source code into machine code, and the environment it depends on. Because of limited space and time, this book cannot cover every file format and operating system. It instead focuses on Linux environments and the ELF file format. The good news is, regardless of platform or file format, Arm instructions are Arm instructions. Even if you reverse engineer an Arm binary compiled for macOS or Windows, the meaning of the instructions themselves remains the same.

This book begins with an introduction explaining what instructions are and where they come from. In the second chapter, you will learn about the ELF file format and its sections, along with a basic overview of the compilation process. Since binary analysis would be incomplete without understanding the context they are executed in, the third chapter provides an overview of operating system fundamentals.

With this background knowledge, you are well prepared to delve into the Arm architecture in Chapter 4. You can find the most common data processing instructions in Chapter 5, followed by an overview of memory access instructions in Chapter 6. These instructions are a significant part of the Arm architecture, which is also referred to as a Load/Store architecture. Chapters 7 and 8 discuss conditional execution and control flow, which are crucial components of reverse engineering.

Chapter 9 is where it starts to get particularly interesting for reverse engineers. Knowing the different types of Arm environments is crucial, especially when you perform dynamic analysis and need to analyze binaries during execution.

With the information provided so far, you are already well equipped for your next reverse engineering adventure. To get you started, Chapter 10 includes an

overview of the most common static analysis tools, followed by small practical static analysis examples you can follow step-by-step.

Reverse engineering would be boring without dynamic analysis to observe how a program behaves during execution. In Chapter 11, you will learn about the most common dynamic analysis tools as well as examples of useful commands you can use during your analysis. This chapter concludes with two practical debugging examples: debugging a memory corruption vulnerability and debugging a process in GDB.

Reverse engineering is useful for a variety of use cases. You can use your knowledge of the Arm instruction set and reverse engineering techniques to expand your skill set into different areas, such as vulnerability analysis or malware analysis.

Reverse engineering is an invaluable skill for malware analysts, but they also need to be familiar with the environment a given malware sample was compiled for. To get you started in this area, this book includes a chapter on analyzing arm64 macOS malware (Chapter 12) written by Patrick Wardle, who is also the author of *The Art of Mac Malware*.[3] Unlike previous chapters, this chapter does not focus on Arm assembly. Instead, it introduces you to common anti-analysis techniques that macOS malware uses to avoid being analyzed. The purpose of this chapter is to provide an introduction to macOS malware compatible with Apple Silicon (M1/M2) so that anyone interested in hunting and analyzing Arm-based macOS malware can get a head start.

This book took a little over two years to write. I began writing in March 2020, when the pandemic hit and put us all in quarantine. Two years and a lot of sweat and tears later, I'm happy to finally see it come to life. Thank you for putting your faith in me. I hope that this book will serve as a useful guide as you embark on your reverse engineering journey and that it will make the process smoother and less intimidating.

[3] https://taomm.org

Part I

Arm Assembly Internals

If you've just picked up this book from the shelf, you're probably interested in learning how to reverse engineer compiled Arm binaries because major tech vendors are now embracing the Arm architecture. Perhaps you're a seasoned veteran of x86-64 reverse engineering but want to stay ahead of the curve and learn more about the architecture that is starting to take over the processor market. Perhaps you're looking to get started on security analysis to find vulnerabilities in Arm-based software or analyze Arm-based malware. Or perhaps you're just getting started in reverse engineering and have hit a point where a deeper level of detail is required to achieve your goal.

Wherever you are on your journey into the Arm-based universe of reverse engineering, this book is about preparing you, the reader, to understand the language of Arm binaries, showing you how to analyze them, and, more importantly, preparing you for the future of Arm devices.

Learning assembly language and how to analyze compiled software is useful in a wide variety of applications. As with every skill, learning the syntax can seem difficult and complicated at first, but it eventually becomes easier with practice.

In the first part of this book, we'll look at the fundamentals of Arm's main Cortex-A architecture, specifically the Armv8-A, and the main instructions you'll encounter when reverse engineering software compiled for this platform. In the second part of the book, we'll look at some common tools and techniques for reverse engineering. To give you inspiration for different applications of Arm-based reverse engineering, we will look at practical examples, including how to analyze malware compiled for Apple's M1 chip.

CHAPTER 1

Introduction to Reverse Engineering

Introduction to Assembly

If you're reading this book, you've probably already heard about this thing called the *Arm assembly language* and know that understanding it is the key to analyzing binaries that run on Arm. But what is this language, and why does it exist? After all, programmers usually write code in high-level languages such as C/C++, and hardly anyone programs in assembly directly. High-level languages are, after all, far more convenient for programmers to program in.

Unfortunately, these high-level languages are too complex for processors to interpret directly. Instead, programmers *compile* these high-level programs down into the binary *machine code* that the processor can run.

This machine code is not quite the same as assembly language. If you were to look at it directly in a text editor, it would look unintelligible. Processors *also* don't run assembly language; they run only machine code. So, why is it so important in reverse engineering?

To understand the purpose of assembly, let's do a quick tour of the history of computing to see how we got to where we are and how everything connects.

Bits and Bytes

Back in the mists of time when it all started, people decided to create computers and have them perform simple tasks. Computers don't speak our human

languages—they are just electronic devices after all—and so we needed a way to communicate with them electronically. At the lowest level, computers operate on electrical signals, and these signals are formed by switching electrical voltages between one of two levels: on and off.

The first problem is that we need a way to describe these "ons" and "offs" for communication, storage, and simply describing the state of the system. Since there are two states, it was only natural to use the binary system for encoding these values. Each binary digit (or *bit*) could be 0 or 1. Although each bit can store only the smallest amount of information possible, stringing multiple bits together allows representation of much larger numbers. For example, the number 30,284,334,537 could be represented in just 35 bits as the following:

```
11100001101000101100100010111001001
```

Already this system allows for encoding large numbers, but now we have a new problem: where does one number in memory (or on a magnetic tape) end and the next one begin? This is perhaps a strange question to ask modern readers, but back when computers were first being designed, this was a serious problem. The simplest solution here would be to create fixed-size groupings of bits. Computer scientists, never wanting to miss out on a good naming pun, called this group of binary digits or bits a *byte*.

So, how many bits should be in a byte? This might seem like a blindingly obvious question to our modern ears, since we all know that a modern byte is 8 bits. But it was not always so.

Originally, different systems made different choices for how many bits would be in their bytes. The predecessor of the 8-bit byte we know today is the 6-bit Binary Coded Decimal Interchange Code (BCDIC) format for representing alphanumeric information used in early IBM computers, such as the IBM 1620 in 1959. Before that, bytes were often 4 bits long, and before that, a byte stood for an arbitrary number of bits greater than 1. Only later, with IBM's 8-bit Extended Binary Coded Decimal Interchange Code (EBCDIC), introduced in the 1960s in its mainframe computer product line System/360 and which had byte-addressable memory with 8-bit bytes, did the byte start to standardize around having 8 bits. This then led to the adoption of the 8-bit storage size in other widely used computer systems, including the Intel 8080 and Motorola 6800.

The following excerpt is from a book titled *Planning a Computer System*, published 1962, listing three main reasons for adopting the 8-bit byte[1]:

> *1. Its full capacity of 256 characters was considered to be sufficient for the great majority of applications.*

[1]Planning a Computer System, Project Stretch, McGraw-Hill Book Company, Inc., 1962. (`http://archive.computerhistory.org/resources/text/IBM/Stretch/pdfs/Buchholz_102636426.pdf`)

2. Within the limits of this capacity, a single character is represented by a single byte, so that the length of any particular record is not dependent on the coincidence of characters in that record.

3. 8-bit bytes are reasonably economical of storage space.

An 8-bit byte can hold one of 256 uniquely different values from 00000000 to 11111111. The interpretation of those values, of course, depends on the software using it. For example, we can store positive numbers in those bytes to represent a positive number from 0 to 255 inclusive. We can also use the two's complement scheme to represent *signed* numbers from –128 to 127 inclusive.

Character Encoding

Of course, computers didn't just use bytes for encoding and processing integers. They would also often store and process human-readable letters and numbers, called *characters*.

Early character encodings, such as ASCII, had settled on using 7 bits per byte, but this gave only a limited set of 128 possible characters. This allowed for encoding English-language letters and digits, as well as a few symbol characters and control characters, but could not represent many of the letters used in other languages. The EBCDIC standard, using its 8-bit bytes, chose a different character set entirely, with code pages for "swapping" to different languages. But ultimately this character set was too cumbersome and inflexible.

Over time, it became clear that we needed a truly universal character set, supporting all the world's living languages and special symbols. This culminated in the creation of the Unicode project in 1987. A few different Unicode encodings exist, but the dominant encoding used on the Web is UTF-8. Characters within the ASCII character -set are included verbatim in UTF-8, and "extended characters" can spread out over multiple consecutive bytes.

Since characters are now encoded as bytes, we can represent characters using two hexadecimal digits. For example, the characters *A*, *R*, and *M* are normally encoded with the octets shown in Figure 1.1.

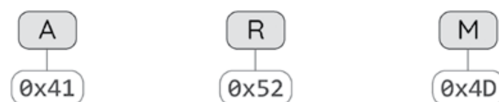


Figure 1.1: Letters A, R, and M and their hexadecimal values

Each hexadecimal digit can be encoded with a 4-bit pattern ranging from 0000 to 1111, as shown in Figure 1.2.

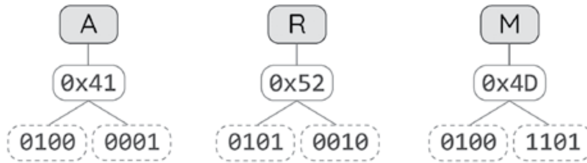


Figure 1.2: Hexadecimal ASCII values and their 8-bit binary equivalents

Since two hexadecimal values are required to encode an ASCII character, 8 bits seemed like the ideal for storing text in most written languages around the world, or a multiple of 8 bits for characters that cannot be represented in 8 bits alone.

Using this pattern, we can more easily interpret the meaning of a long string of bits. The following bit pattern encodes the word *Arm*:

```
0100 0001 0101 0010 0100 1101
```

Machine Code and Assembly

One uniquely powerful aspect of computers, as opposed to the mechanical calculators that predated them, is that they can also encode their *logic* as data. This *code* can also be stored in memory or on disk and be processed or changed on demand. For example, a software update can completely change the operating system of a computer without the need to purchase a new machine.

We've already seen how numbers and characters are encoded, but how is this logic encoded? This is where the processor architecture and its instruction set comes into play.

If we were to create our own computer processor from scratch, we could design our own *instruction encoding*, mapping binary patterns to machine codes that our processor can interpret and respond to, in effect, creating our own "machine language." Since machine codes are meant to "instruct" the circuitry to perform an "operation," these machine codes are also referred to as *instruction codes*, or, more commonly, *operation codes* (*opcodes*).

In practice, most people use existing computer processors and therefore use the instruction encodings defined by the processor manufacturer. On Arm, instruction encodings have a fixed size and can be either 32-bit or 16-bit, depending on the instruction set in use by the program. The processor fetches and interprets each instruction and runs each in turn to perform the logic of the program. Each instruction is a binary pattern, or *instruction encoding*, which follows specific rules defined by the Arm architecture.

By way of example, let's assume we're building a tiny 16-bit instruction set and are defining how each instruction will look. Our first task is to designate part of the encoding as specifying exactly what *type* of instruction is to be run, called the *opcode*. For example, we might set the first 7 bits of the instruction to be an *opcode* and specify the opcodes for addition and subtraction, as shown in Table 1.1.

Table 1.1: Addition and Subtraction Opcodes

OPERATION	OPCODE
Addition	0001110
Subtraction	0001111

Writing machine code by hand is possible but unnecessarily cumbersome. In practice, we'll want to write assembly in some human-readable "assembly language" that will be converted into its machine code equivalent. To do this, we should also define the shorthand for the instruction, called the instruction *mnemonic*, as shown in Table 1.2.

Table 1.2: Mnemonics

OPERATION	OPCODE	MNEMONIC
Addition	0001110	`ADD`
Subtraction	0001111	`SUB`

Of course, it's not sufficient to tell a processor to just do an "addition." We also need to tell it *what* two things to add and what to do with the result. For example, if we write a program that performs "a = b + c," the values of *b* and *c* need to be stored somewhere before the instruction begins, and the instruction needs to know where to write the result *a* to.

In most processors, and Arm processors in particular, these temporary values are usually stored in *registers*, which store a small number of "working" values. Programs can pull data in from memory (or disk) into registers ready to be processed and can spill result data back to longer-term storage after processing.

The number and naming conventions of registers are architecture-dependent. As software has become more and more complex, programs must often juggle larger numbers of values at the same time. Storing and operating on these values in registers is faster than doing so in memory directly, which means that registers reduce the number of times a program needs to access memory and result in faster execution.

Going back to our earlier example, we were designing a 16-bit instruction to perform an operation that adds a value to a register and writes the result into another register. Since we use 7 bits for the operation (`ADD`/`SUB`) itself, the remaining 9 bits can be used for encoding the source and the destination registers and a constant value we want to add or subtract. In this example, we split the remaining bits evenly and assign the shortcuts and respective machine codes shown in Table 1.3.

Table 1.3: Manually Assigning the Machine Codes

OPERATION	MNEMONIC	MACHINE CODE
Addition	ADD	0001110
Subtraction	SUB	0001111
Integer value 2	#2	010
Operand register	R0	000
Destination register	R1	001

Instead of generating these machine codes by hand, we could instead write a little program that converts the syntax `ADD R1, R0, #2` (R1 = R0 + 2) into the corresponding machine-code pattern and hand that machine-code pattern to our example processor. See Table 1.4.

Table 1.4: Programming the Machine Codes

INSTRUCTION	BINARY MACHINE CODE	HEXADECIMAL ENCODING
ADD R1, R0, #2	0001110 010 000 001	0x1C81
SUB R1, R0, #2	0001111 010 000 001	0x1E81

The bit pattern we constructed represents one of the instruction encodings for 16-bit `ADD` and `SUB` instructions that are part of the T32 instruction set. In Figure 1.3 you can see its components and how they are ordered in the instruction encoding.

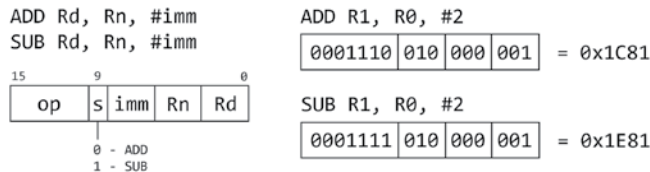


Figure 1.3: 16-bit Thumb encoding of ADD and SUB immediate instruction

Of course, this is just a simplified example. Modern processors provide hundreds of possible instructions, often with more complex subencodings. For example, Arm defines the load register instruction (with the LDR mnemonic) that loads a 32-bit value from memory into a register, as illustrated in Figure 1.4.

In this instruction, the "address" to load is specified in register 2 (called `R2`), and the read value is written to register 3 (called `R3`).

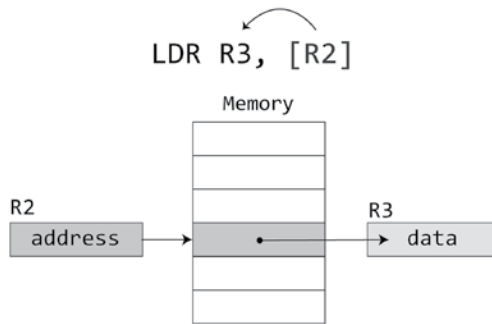


Figure 1.4: LDR instruction loading a value from the address in `R2` to register `R3`

The syntax of writing brackets around `R2` indicates that the value in `R2` is to be interpreted as an address in memory, rather than an ordinary value. In other words, we do not want to copy the value in `R2` into `R3`, but rather fetch the contents of memory at *the address* given by `R2` and load that value into `R3`. There are many reasons for a program to reference a memory location, including calling a function or loading a value from memory into a register.

This is, in essence, the difference between machine code and assembly code. Assembly language is the human-readable syntax that shows how each encoded instruction should be interpreted. Machine code, by contrast, is the actual binary data ingested and processed by the actual processor, with its encoding specified precisely by the processor designer.

Assembling

Since processors understand only machine code, and not assembly language, how do we convert between them? To do this we need a program to convert our handwritten assembly instructions into their machine-code equivalents. The programs that perform this task are called *assemblers*.

In practice, assemblers are capable not only of understanding and translating individual instructions into machine code but also of interpreting *assembler directives*[2] that direct the assembler to do other things, such as switch between data and code or assemble different instruction sets. Therefore, the terms *assembly language* and *assembler language* are just two ways of looking at the same thing. The syntax and meaning of individual assembler directives and expressions depend on the specific assembler.

[2] `https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html`