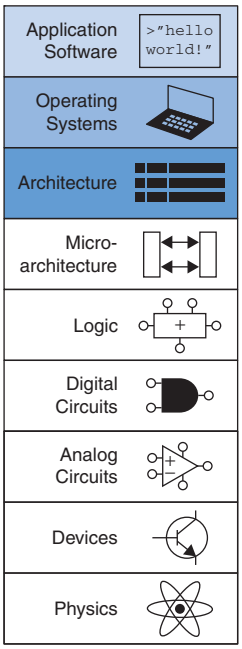
Architecture

6



6.1 INTRODUCTION

The previous chapters introduced digital design principles and building blocks. In this chapter, we jump up a few levels of abstraction to define the architecture of a computer. The *architecture* is the programmer's view of a computer. It is defined by the instruction set (language) and operand locations (registers and memory). Many different architectures exist, such as ARM, x86, MIPS, SPARC, and PowerPC.

The first step in understanding any computer architecture is to learn its language. The words in a computer's language are called *instructions*. The computer's vocabulary is called the *instruction set*. All programs running on a computer use the same instruction set. Even complex software applications, such as word processing and spreadsheet applications, are eventually compiled into a series of simple instructions such as add, subtract, and branch. Computer instructions indicate both the operation to perform and the operands to use. The operands may come from memory, from registers, or from the instruction itself.

Computer hardware understands only 1's and 0's, so instructions are encoded as binary numbers in a format called machine language. Just as we use letters to encode human language, computers use binary numbers to encode machine language. The ARM architecture represents each instruction as a 32-bit word. Microprocessors are digital systems that read and execute machine language instructions. However, humans consider reading machine language to be tedious, so we prefer to represent the instructions in a symbolic format called assembly language.

The instruction sets of different architectures are more like different dialects than different languages. Almost all architectures define basic instructions, such as add, subtract, and branch, that operate on memory or registers. Once you have learned one instruction set, understanding others is fairly straightforward.

A computer architecture does not define the underlying hardware implementation. Often, many different hardware implementations of a single architecture exist. For example, Intel and Advanced Micro Devices (AMD) both sell various microprocessors belonging to the same x86 architecture. They all can run the same programs, but they use different underlying hardware and therefore offer trade-offs in performance, price, and power. Some microprocessors are optimized for high-performance servers, whereas others are optimized for long battery life in laptop computers. The specific arrangement of registers, memories, ALUs, and other building blocks to form a microprocessor is called the microarchitecture and will be the subject of Chapter 7. Often, many different microarchitectures exist for a single architecture.

In this text, we introduce the ARM architecture. This architecture was first developed in the 1980s by Acorn Computer Group, which spun off Advanced RISC Machines Ltd., now known as ARM. Over 10 billion ARM processors are sold every year. Almost all cell phones and tablets contain multiple ARM processors. The architecture is used in everything from pinball machines to cameras to robots to cars to rack-mounted servers. ARM is unusual in that it does not sell processors directly, but rather licenses other companies to build its processors, often as part of a larger system-on-chip. For example, Samsung, Altera, Apple, and Qualcomm all build ARM processors, either using microarchitectures purchased from ARM or microarchitectures developed internally under license from ARM. We choose to focus on ARM because it is a commercial leader and because the architecture is clean, with few idiosyncrasies. We start by introducing assembly language instructions, operand locations, and common programming constructs, such as branches, loops, array manipulations, and function calls. We then describe how the assembly language translates into machine language and show how a program is loaded into memory and executed.

The "ARM architecture" we describe is ARM version 4 (ARMv4), which forms the core of the instruction set. Section 6.7 summarizes new features in versions 5–8 of the architecture. The ARM *Architecture Reference Manual* (ARM), available online, is the authoritative definition of the architecture.

Throughout the chapter, we motivate the design of the ARM architecture using four principles articulated by David Patterson and John Hennessy in their text *Computer Organization and Design*: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises.

6.2 ASSEMBLY LANGUAGE

Assembly language is the human-readable representation of the computer's native language. Each assembly language instruction specifies both the operation to perform and the operands on which to operate. We introduce simple arithmetic instructions and show how these operations are written in assembly language. We then define the ARM instruction operands: registers, memory, and constants.

This chapter assumes that you already have some familiarity with a high-level programming language such as C, C++, or Java.

(These languages are practically identical for most of the examples in this chapter, but where they differ, we will use C.) Appendix C provides an introduction to C for those with little or no prior programming experience.

We used Keil's ARM Microcontroller Development Kit (MDK-ARM) to compile, assemble, and simulate the example assembly code in this chapter. The MDK-ARM is a free development tool that comes with a complete ARM compiler. Labs available on this textbook's companion site (see Preface) show how to install and use this tool to write, compile, simulate, and debug both C and assembly programs.

6.2.1 Instructions

The most common operation computers perform is addition. Code Example 6.1 shows code for adding variables `b` and `c` and writing the result to `a`. The code is shown on the left in a high-level language (using the syntax of C, C++, and Java) and then rewritten on the right in ARM assembly language. Note that statements in a C program end with a semicolon.

Code Example 6.1 ADDITION

High-Level Code	ARM Assembly Code
`a = b + c;`	`ADD a, b, c`

The first part of the assembly instruction, `ADD`, is called the *mnemonic* and indicates what operation to perform. The operation is performed on `b` and `c`, the *source operands*, and the result is written to `a`, the *destination operand*.

Mnemonic (pronounced ni-mon-ik) comes from the Greek word μιμνΕσκεσταιι, to remember. The assembly language mnemonic is easier to remember than a machine language pattern of 0's and 1's representing the same operation.

Code Example 6.2 SUBTRACTION

High-Level Code	ARM Assembly Code
`a = b - c;`	`SUB a, b, c`

Code Example 6.2 shows that subtraction is similar to addition. The instruction format is the same as the `ADD` instruction except for the operation specification, `SUB`. This consistent instruction format is an example of the first design principle:

Design Principle 1: Regularity supports simplicity.

Instructions with a consistent number of operands—in this case, two sources and one destination—are easier to encode and handle in hardware. More complex high-level code translates into multiple ARM instructions, as shown in Code Example 6.3.

In the high-level language examples, single-line comments begin with `//` and continue until the end of the line. Multiline comments begin with `/*` and end with `*/`. In ARM assembly language, only single-line comments

Code Example 6.3 MORE COMPLEX CODE

High-Level Code

```
a = b + c - d;    // single-line comment
                  /* multiple-line
                     comment */
```

ARM Assembly Code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

are used. They begin with a semicolon (`;`) and continue until the end of the line. The assembly language program in Code Example 6.3 requires a temporary variable `t` to store the intermediate result. Using multiple assembly language instructions to perform more complex operations is an example of the second design principle of computer architecture:

Design Principle 2: Make the common case fast.

The ARM instruction set makes the common case fast by including only simple, commonly used instructions. The number of instructions is kept small so that the hardware required to decode the instruction and its operands can be simple, small, and fast. More elaborate operations that are less common are performed using sequences of multiple simple instructions. Thus, ARM is a *reduced instruction set computer* (RISC) architecture. Architectures with many complex instructions, such as Intel's x86 architecture, are *complex instruction set computers* (CISC). For example, x86 defines a "string move" instruction that copies a string (a series of characters) from one part of memory to another. Such an operation requires many, possibly even hundreds, of simple instructions in a RISC machine. However, the cost of implementing complex instructions in a CISC architecture is added hardware and overhead that slows down the simple instructions.

A RISC architecture minimizes the hardware complexity and the necessary instruction encoding by keeping the set of distinct instructions small. For example, an instruction set with 64 simple instructions would need $\log_2 64 = 6$ bits to encode the operation. An instruction set with 256 complex instructions would need $\log_2 256 = 8$ bits of encoding per instruction. In a CISC machine, even though the complex instructions may be used only rarely, they add overhead to all instructions, even the simple ones.

6.2.2 Operands: Registers, Memory, and Constants

An instruction operates on *operands*. In Code Example 6.1, the variables `a`, `b`, and `c` are all operands. But computers operate on 1's and 0's, not variable names. The instructions need a physical location from which to retrieve the binary data. Operands can be stored in registers or memory, or they may be constants stored in the instruction itself. Computers use

various locations to hold operands in order to optimize for speed and data capacity. Operands stored as constants or in registers are accessed quickly, but they hold only a small amount of data. Additional data must be accessed from memory, which is large but slow. ARM (prior to ARMv8) is called a 32-bit architecture because it operates on 32-bit data.

Version 8 of the ARM architecture has been extended to 64 bits, but we will focus on the 32-bit version in this book.

Registers

Instructions need to access operands quickly so that they can run fast. But operands stored in memory take a long time to retrieve. Therefore, most architectures specify a small number of registers that hold commonly used operands. The ARM architecture uses 16 registers, called the *register set* or *register file*. The fewer the registers, the faster they can be accessed. This leads to the third design principle:

Design Principle 3: Smaller is faster.

Looking up information from a small number of relevant books on your desk is a lot faster than searching for the information in the stacks at a library. Likewise, reading data from a small register file is faster than reading it from a large memory. A register file is typically built from a small SRAM array (see Section 5.5.3).

Code Example 6.4 shows the `ADD` instruction with register operands. ARM register names are preceded by the letter 'R'. The variables `a`, `b`, and `c` are arbitrarily placed in R0, R1, and R2. The name R1 is pronounced "register 1" or "R1" or "register R1". The instruction adds the 32-bit values contained in R1 (`b`) and R2 (`c`) and writes the 32-bit result to R0 (`a`). Code Example 6.5 shows ARM assembly code using a register, R4, to store the intermediate calculation of `b + c`:

Code Example 6.4 REGISTER OPERANDS

High-Level Code	ARM Assembly Code
`a = b + c;`	`; R0 = a, R1 = b, R2 = c` `ADD R0, R1, R2 ; a = b + c`

Code Example 6.5 TEMPORARY REGISTERS

High-Level Code	ARM Assembly Code

`a = b + c - d;`	`; R0 = a, R1 = b, R2 = c, R3 = d; R4 = t`
	`ADD R4, R1, R2 ; t = b + c`
	`SUB R0, R4, R3 ; a = t - d`

Example 6.1 TRANSLATING HIGH-LEVEL CODE TO ASSEMBLY LANGUAGE

Translate the following high-level code into ARM assembly language. Assume variables `a–c` are held in registers R0–R2 and `f–j` are in R3–R7.

```
a = b - c;
f = (g + h) - (i + j);
```

Solution: The program uses four assembly language instructions.

```
; ARM assembly code
; R0 = a, R1 = b, R2 = c, R3 = f, R4 = g, R5 = h, R6 = i, R7 = j
  SUB R0, R1, R2     ; a = b - c
  ADD R8, R4, R5     ; R8 = g + h
  ADD R9, R6, R7     ; R9 = i + j
  SUB R3, R8, R9     ; f = (g + h) - (i + j)
```

The Register Set

Table 6.1 lists the name and use for each of the 16 ARM registers. R0–R12 are used for storing variables; R0–R3 also have special uses during procedure calls. R13–R15 are also called SP, LR, and PC, and they will be described later in this chapter.

Constants/Immediates

In addition to register operations, ARM instructions can use constant or *immediate* operands. These constants are called immediates, because their values are immediately available from the instruction and do not require a register or memory access. Code Example 6.6 shows the `ADD` instruction adding an immediate to a register. In assembly code, the immediate is preceded by the # symbol and can be written in decimal or hexadecimal. Hexadecimal constants in ARM assembly language start with 0x, as they

Table 6.1 ARM register set

Name	Use
R0	Argument / return value / temporary variable
R1–R3	Argument / temporary variables
R4–R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Code Example 6.6 IMMEDIATE OPERANDS

High-Level Code	ARM Assembly Code

`a = a + 4;`
`b = a – 12;`
	`; R7 = a, R8 = b`
	`ADD R7, R7, #4 ; a = a + 4`
	`SUB R8, R7, #0xC ; b = a – 12`

Code Example 6.7 INITIALIZING VALUES USING IMMEDIATES

High-Level Code	ARM Assembly Code

`i = 0;`
`x = 4080;`
	`; R4 = i, R5 = x`
	`MOV R4, #0 ; i = 0`
	`MOV R5, #0xFF0 ; x = 4080`

do in C. Immediates are unsigned 8- to 12-bit numbers with a peculiar encoding described in Section 6.4.

The move instruction (`MOV`) is a useful way to initialize register values. Code Example 6.7 initializes the variables `i` and `x` to 0 and 4080, respectively. MOV can also take a register source operand. For example, `MOV R1, R7` copies the contents of register R7 into R1.

Memory

If registers were the only storage space for operands, we would be confined to simple programs with no more than 15 variables. However, data can also be stored in memory. Whereas the register file is small and fast, memory is larger and slower. For this reason, frequently used variables are kept in registers. In the ARM architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. By using a combination of memory and registers, a program can access a large amount of data fairly quickly. Recall from Section 5.5 that memories are organized as an array of data words. The ARM architecture uses 32-bit memory addresses and 32-bit data words.

ARM uses a *byte-addressable* memory. That is, each byte in memory has a unique address, as shown in Figure 6.1(a). A 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4. The most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Both the 32-bit word address and the data value in Figure 6.1(b) are given in hexadecimal. For example, data word 0xF2F1AC07 is stored at memory address 4. By convention, memory is drawn with low memory addresses toward the bottom and high memory addresses toward the top.

ARM provides the *load register* instruction, `LDR`, to read a data word from memory into a register. Code Example 6.8 loads memory word 2 into `a` (R7). In C, the number inside the brackets is the *index* or word number,

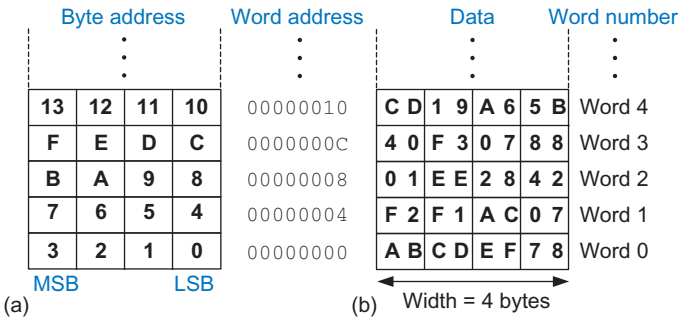


Figure 6.1 **ARM byte-addressable memory showing: (a) byte address and (b) data**

Code Example 6.8 READING MEMORY

High-Level Code	ARM Assembly Code

`a = mem[2];`	`; R7 = a`
	`MOV R5, #0 ; base address = 0`
	`LDR R7, [R5, #8] ; R7 <= data at memory address (R5+8)`

A read from the base address (i.e., index 0) is a special case that requires no offset in the assembly code. For example, a memory read from the base address held in R5 is written as `LDR R3, [R5]`.

which we discuss further in Section 6.3.6. The `LDR` instruction specifies the memory address using a *base register* (R5) and an *offset* (8). Recall that each data word is 4 bytes, so word number 1 is at address 4, word number 2 is at address 8, and so on. The word address is four times the word number. The memory address is formed by adding the contents of the base register (R5) and the offset. ARM offers several modes for accessing memory, as will be discussed in Section 6.3.6.

After the load register instruction (`LDR`) is executed in Code Example 6.8, R7 holds the value 0x01EE2842, which is the data value stored at memory address 8 in Figure 6.1.

ARM uses the *store register* instruction, `STR`, to write a data word from a register into memory. Code Example 6.9 writes the value 42 from register R9 into memory word 5.

Byte-addressable memories are organized in a big-endian or little-endian fashion, as shown in Figure 6.2. In both formats, a 32-bit word's most significant byte (MSB) is on the left and the least significant byte (LSB) is on the right. Word addresses are the same in both formats and refer to the same four bytes. Only the addresses of bytes within a word

ARMv4 requires *word-aligned addresses* for `LDR` and `STR`, that is, a word address that is divisible by four. Since ARMv6, this alignment restriction can be removed by setting a bit in the ARM system control register, but performance of *unaligned* loads is usually worse. Some architectures, such as x86, allow non-word-aligned data reads and writes, but others, such as MIPS, require strict alignment for simplicity. Of course, byte addresses for load byte and store byte, `LDRB` and `STRB` (discussed in Section 6.3.6), need not be word aligned.

Code Example 6.9 WRITING MEMORY

High-Level Code	ARM Assembly Code

`mem[5] = 42;`	`MOV R1, #0 ; base address = 0`
	`MOV R9, #42`
	`STR R9, [R1, #0x14] ; value stored at memory address (R1+20) = 42`

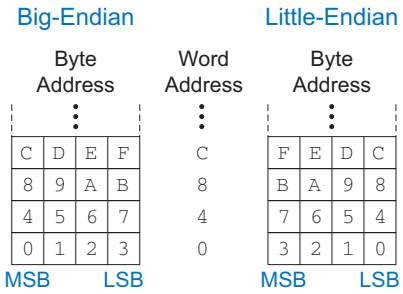
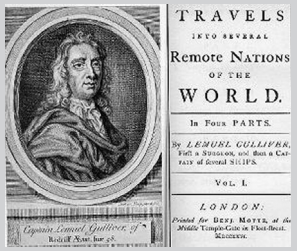


Figure 6.2 Big-endian and little-endian memory addressing

differ. In *big-endian* machines, bytes are numbered starting with 0 at the big (most significant) end. In *little-endian* machines, bytes are numbered starting with 0 at the little (least significant) end.

IBM's PowerPC (formerly found in Macintosh computers) uses big-endian addressing. Intel's x86 architecture (found in PCs) uses little-endian addressing. ARM prefers little-endian but provides support in some versions for *bi-endian* data addressing, which allows data loads and stores in either format. The choice of endianness is completely arbitrary but leads to hassles when sharing data between big-endian and little-endian computers. In examples in this text, we use little-endian format whenever byte ordering matters.



The terms big-endian and little-endian come from Jonathan Swift's *Gulliver's Travels*, first published in 1726 under the pseudonym of Isaac Bickerstaff. In his stories, the Lilliputian king required his citizens (the Little-Endians) to break their eggs on the little end. The Big-Endians were rebels who broke their eggs on the big end.

These terms were first applied to computer architectures by Danny Cohen in his paper "On Holy Wars and a Plea for Peace" published on April Fools Day, 1980 (*USC/ISI IEN 137*). (Photo courtesy of The Brotherton Collection, Leeds University Library.)

6.3 PROGRAMMING

Software languages such as C or Java are called high-level programming languages because they are written at a more abstract level than assembly language. Many high-level languages use common software constructs such as arithmetic and logical operations, conditional execution, if/else statements, for and while loops, array indexing, and function calls. See Appendix C for more examples of these constructs in C. In this section, we explore how to translate these high-level constructs into ARM assembly code.

6.3.1 Data-processing Instructions

The ARM architecture defines a variety of *data-processing* instruction (often called logical and arithmetic instructions in other architectures). We introduce these instructions briefly here because they are necessary to implement higher-level constructs. Appendix B provides a summary of ARM instructions.

Logical Instructions

ARM *logical operations* include `AND`, `ORR` (OR), `EOR` (XOR), and `BIC` (bit clear). These each operate bitwise on two sources and write the result

	Source registers			
R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code		Result			
`AND R3, R1, R2`	R3	0100 0110	1010 0001	0000 0000	0000 0000
`ORR R4, R1, R2`	R4	1111 1111	1111 1111	1111 0001	1011 0111
`EOR R5, R1, R2`	R5	1011 1001	0101 1110	1111 0001	1011 0111
`BIC R6, R1, R2`	R6	0000 0000	0000 0000	1111 0001	1011 0111
`MVN R7, R2`	R7	0000 0000	0000 0000	1111 1111	1111 1111

Figure 6.3 Logical operations

to a destination register. The first source is always a register and the second source is either an immediate or another register. Another logical operation, `MVN` (MoVe and Not), performs a bitwise NOT on the second source (an immediate or register) and writes the result to the destination register. Figure 6.3 shows examples of these operations on the two source values 0x46A1F1B7 and 0xFFFF0000. The figure shows the values stored in the destination register after the instruction executes.

The bit clear (`BIC`) instruction is useful for masking bits (i.e., forcing unwanted bits to 0). `BIC R6, R1, R2` computes R1 AND NOT R2. In other words, `BIC` clears the bits that are asserted in R2. In this case, the top two bytes of R1 are cleared or *masked*, and the unmasked bottom two bytes of R1, 0xF1B7, are placed in R6. Any subset of register bits can be masked.

The `ORR` instruction is useful for combining bitfields from two registers. For example, 0x347A0000 `ORR` 0x000072FC = 0x347A72FC.

Shift Instructions

Shift instructions shift the value in a register left or right, dropping bits off the end. The rotate instruction rotates the value in a register right by up to 31 bits. We refer to both shift and rotate generically as shift operations. ARM shift operations are `LSL` (logical shift left), `LSR` (logical shift right), `ASR` (arithmetic shift right), and `ROR` (rotate right). There is no `ROL` instruction because left rotation can be performed with a right rotation by a complementary amount.

As discussed in Section 5.2.5, left shifts always fill the least significant bits with 0's. However, right shifts can be either logical (0's shift into the most significant bits) or arithmetic (the sign bit shifts into the most significant bits). The amount by which to shift can be an immediate or a register.

Figure 6.4 shows the assembly code and resulting register values for `LSL`, `LSR`, `ASR`, and `ROR` when shifting by an immediate value. R5 is shifted by the immediate amount, and the result is placed in the destination register.

	Source register			
R5	1111 1111	0001 1100	0001 0000	1110 0111

Assembly Code		Result			
`LSL R0, R5, #7`	R0	1000 1110	0000 1000	0111 0011	1000 0000
`LSR R1, R5, #17`	R1	0000 0000	0000 0000	0111 1111	1000 1110
`ASR R2, R5, #3`	R2	1111 1111	1110 0011	1000 0010	0001 1100
`ROR R3, R5, #21`	R3	1110 0000	1000 0111	0011 1111	1111 1000

Figure 6.4 Shift instructions with immediate shift amounts

	Source registers			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
`LSL R4, R8, R6`	R4	0110 1110	0111 0000	0000 0000	0000 0000
`ROR R5, R8, R6`	R5	1100 0001	0110 1110	0111 0000	1000 0001

Figure 6.5 Shift instructions with register shift amounts

Shifting a value left by N is equivalent to multiplying it by 2^N. Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N, as discussed in Section 5.2.5. Logical shifts are also used to extract or assemble bitfields.

Figure 6.5 shows the assembly code and resulting register values for shift operations where the shift amount is held in a register, R6. This instruction uses the *register-shifted register* addressing mode, where one register (R8) is shifted by the amount (20) held in a second register (R6).

Multiply Instructions*

Multiplication is somewhat different from other arithmetic operations. Multiplying two 32-bit numbers produces a 64-bit product. The ARM architecture provides *multiply instructions* that result in a 32-bit or 64-bit product. Multiply (`MUL`) multiplies two 32-bit numbers and produces a 32-bit result. `MUL R1, R2, R3` multiplies the values in R2 and R3 and places the least significant bits of the product in R1; the most significant 32 bits of the product are discarded. This instruction is useful for multiplying small numbers whose result fits in 32 bits. `UMULL` (unsigned multiply long) and `SMULL` (signed multiply long) multiply two 32-bit numbers and produce a 64-bit product. For example, `UMULL R1, R2, R3, R4` performs an unsigned multiply of R3 and R4. The least significant 32 bits of the product is placed in R1 and the most significant 32 bits are placed in R2.

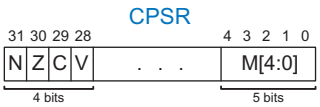


Figure 6.6 Current Program Status Register (CPSR)

The least significant five bits of the CPSR are *mode* bits and will be described in Section 6.6.3.

Other useful instructions for comparing two values are `CMN`, `TST`, and `TEQ`. Each instruction performs an operation, updates the condition flags, and discards the result. `CMN` (compare negative) compares the first source to the negative of the second source by adding the two sources. As will be shown in Section 6.4, ARM instructions only encode positive immediates. So, `CMN R2, #20` is used instead of `CMP R2, #-20`. `TST` (test) ANDs the source operands. It is useful for checking if some portion of the register is zero or nonzero. For example, `TST R2, #0xFF` would set the *Z* flag if the low byte of R2 is 0. `TEQ` (test if equal) checks for equivalence by XOR-ing the sources. Thus, the *Z* flag is set when they are equal and the *N* flag is set when the signs are different.

Each of these instructions also has a multiply-accumulate variant, `MLA`, `SMLAL`, and `UMLAL`, that adds the product to a running 32- or 64-bit sum. These instructions can boost the math performance in applications such as matrix multiplication and signal processing consisting of repeated multiplies and adds.

6.3.2 Condition Flags

Programs would be boring if they could only run in the same order every time. ARM instructions optionally set *condition flags* based on whether the result is negative, zero, etc. Subsequent instructions then execute *conditionally*, depending on the state of those condition flags. The ARM condition flags, also called *status flags*, are negative (*N*), zero (*Z*), carry (*C*), and overflow (*V*), as listed in Table 6.2. These flags are set by the ALU (see Section 5.2.4) and are held in the top 4 bits of the 32-bit *Current Program Status Register* (*CPSR*), as shown in Figure 6.6.

The most common way to set the status bits is with the compare (`CMP`) instruction, which subtracts the second source operand from the first and sets the condition flags based on the result. For example, if the numbers are equal, the result will be zero and the *Z* flag is set. If the first number is an unsigned value that is higher than or the same as the second, the subtraction will produce a carry out and the *C* flag is set.

Subsequent instructions can conditionally execute depending on the state of the flags. The instruction mnemonic is followed by a *condition mnemonic* that indicates when to execute. Table 6.3 lists the 4-bit condition field (*cond*), the condition mnemonic, name, and the state of the condition flags that result in instruction execution (CondEx). For example, suppose a program performs `CMP R4, R5`, and then `ADDEQ R1, R2, R3`. The compare sets the *Z* flag if R4 and R5 are equal, and the `ADDEQ` executes only if the *Z* flag is set. The *cond* field will be used in machine language encodings in Section 6.4.

Table 6.2 Condition flags

Flag	Name	Description
N	Negative	Instruction result is negative, i.e., bit 31 of the result is 1
Z	Zero	Instruction result is zero
C	Carry	Instruction causes a carry out
V	oVerflow	Instruction causes an overflow

Table 6.3 Condition mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Condition mnemonics differ for signed and unsigned comparison. For example, ARM provides two forms of greater than or equal comparison: HS (CS) is used for unsigned numbers and GE for signed. For unsigned numbers, $A - B$ will produce a carry out (C) when $A \geq B$. For signed numbers, $A - B$ will make N and V either both 0 or both 1 when $A \geq B$. Figure 6.7 highlights the difference between HS and GE comparisons with two examples using 4-bit numbers for ease of interpretation.

	Unsigned	Signed
A = 1001_2	A = 9	A = –7
B = 0010_2	B = 2	B = 2

A – B: 1001 + 1110 = 10111 (a)

$NZCV = 0011_2$

HS: TRUE

GE: FALSE

	Unsigned	Signed
A = 0101_2	A = 5	A = 5
B = 1101_2	B = 13	B = –3

A – B: 0101 + 0011 = 1000 (b)

$NZCV = 1001_2$

HS: FALSE

GE: TRUE

Figure 6.7 Signed vs. unsigned comparison: HS vs. GE

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, `SUBS R2, R3, R7` will subtract R7 from R3, put the result in R2, and set the condition flags. Table B.5 in Appendix B summarizes which condition flags are influenced by each instruction. All data-processing instructions will affect the N and Z flags based on whether the result is zero or has the most significant bit set. `ADDS` and `SUBS` also influence V and C, and shifts influence C.

Code Example 6.10 shows instructions that execute conditionally. The first instruction, `CMP R2, R3`, executes unconditionally and sets the condition flags. The remaining instructions execute conditionally, depending on the values of the condition flags. Suppose R2 and R3 contain the values 0x80000000 and 0x00000001. The compare computes R2 – R3 = 0x80000000 – 0x00000001 = 0x80000000 + 0xFFFFFFFF = 0x7FFFFFFF with a carry out ($C = 1$). The sources had opposite signs and the sign of the result differs from the sign of the first source, so the result overflows ($V = 1$). The remaining flags (N and Z) are 0. `ANDHS` executes

Code Example 6.10 CONDITIONAL EXECUTION

ARM Assembly Code

```
CMP   R2, R3
ADDEQ R4, R5, #78
ANDHS R7, R8, R9
ORRMI R10, R11, R12
EORLT R12, R7, R10
```

because $C = 1$. `EORLT` executes because N is 0 and V is 1 (see Table 6.3). Intuitively, `ANDHS` and `EORLT` execute because R2 $\geq$ R3 (unsigned) and R2 $<$ R3 (signed), respectively. `ADDEQ` and `ORRMI` do not execute because the result of R2 – R3 is not zero (i.e., R2 $\neq$ R3) or negative.

6.3.3 Branching

An advantage of a computer over a calculator is its ability to make decisions. A computer performs different tasks depending on the input. For example, if/else statements, switch/case statements, while loops, and for loops all conditionally execute code depending on some test.

One way to make decisions is to use conditional execution to ignore certain instructions. This works well for simple if statements where a small number of instructions are ignored, but it is wasteful for if statements with many instructions in the body, and it is insufficient to handle loops. Thus, ARM and most other architectures use *branch instructions* to skip over sections of code or repeat code.

A program usually executes in sequence, with the program counter (PC) incrementing by 4 after each instruction to point to the next instruction. (Recall that instructions are 4 bytes long and ARM is a byte-addressed architecture.) Branch instructions change the program counter. ARM includes two types of branches: a simple *branch* (`B`) and *branch and link* (`BL`). `BL` is used for function calls and is discussed in Section 6.3.7. Like other ARM instructions, branches can be unconditional or conditional. Branches are also called *jumps* in some architectures.

Code Example 6.11 shows unconditional branching using the branch instruction `B`. When the code reaches the `B TARGET` instruction, the branch is *taken*. That is, the next instruction executed is the `SUB` instruction just after the *label* called `TARGET`.

Assembly code uses labels to indicate instruction locations in the program. When the assembly code is translated into machine code, these labels are translated into instruction addresses (see Section 6.4.3). ARM assembly labels cannot be reserved words, such as instruction mnemonics. Most programmers indent their instructions but not the labels, to help

Code Example 6.11 UNCONDITIONAL BRANCHING

ARM Assembly Code

```
  ADD R1, R2, #17    ; R1 = R2 + 17
  B   TARGET         ; branch to TARGET
  ORR R1, R1, R3     ; not executed
  AND R3, R1, #0xFF  ; not executed

TARGET
  SUB R1, R1, #78    ; R1 = R1 – 78
```

Code Example 6.12 CONDITIONAL BRANCHING

ARM Assembly Code

```
  MOV R0, #4         ; R0 = 4
  ADD R1, R0, R0     ; R1 = R0 + R0 = 8
  CMP R0, R1         ; set flags based on R0–R1 = –4. NZCV = 1000
  BEQ THERE          ; branch not taken (Z != 1)
  ORR R1, R1, #1     ; R1 = R1 OR 1 = 9

THERE
  ADD R1, R1, #78  ; R1 = R1 + 78 = 87
```

make labels stand out. The ARM compiler makes this a requirement: labels must not be indented, and instructions must be preceded by white space. Some compilers, including GCC, require a colon after the label.

Branch instructions can execute conditionally based on the condition mnemonics listed in Table 6.3. Code Example 6.12 illustrates the use of `BEQ`, branching dependent on equality ($Z = 1$). When the code reaches the `BEQ` instruction, the Z condition flag is 0 (i.e., `R0` $\neq$ `R1`), so the branch is *not taken*. That is, the next instruction executed is the `ORR` instruction.

6.3.4 Conditional Statements

if, if/else, and switch/case statements are conditional statements commonly used in high-level languages. They each conditionally execute a *block* of code consisting of one or more statements. This section shows how to translate these high-level constructs into ARM assembly language.

if Statements

An if statement executes a block of code, the *if block*, only when a condition is met. Code Example 6.13 shows how to translate an if statement into ARM assembly code.

Code Example 6.13 IF STATEMENT

High-Level Code

```
if (apples == oranges)
   f = i + 1;


f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1       ; apples == oranges ?
  BNE L1           ; if not equal, skip if block
  ADD R2, R3, #1   ; if block: f = i + 1
L1
  SUB R2, R2, R3   ; f = f - i
```

Recall that != is an inequality comparison and == is an equality comparison in the high-level code.

The assembly code for the if statement tests the opposite condition of the one in the high-level code. In Code Example 6.13, the high-level code tests for `apples == oranges`. The assembly code tests for `apples != oranges` using `BNE` to skip the if block if the condition is **not** satisfied. Otherwise, `apples == oranges`, the branch is not taken, and the if block is executed.

Because any instruction can be conditionally executed, the ARM assembly code for Code Example 6.13 could also be written more compactly as shown below.

```
CMP   R0, R1          ; apples == oranges ?
ADDEQ R2, R3, #1      ; f = i + 1 on equality (i.e., Z = 1)
SUB   R2, R2, R3      ; f = f - i
```

This solution with conditional execution is shorter and also faster because it involves one fewer instruction. Moreover, we will see in Section 7.5.3 that branches sometimes introduce extra delay, whereas conditional execution is always fast. This example shows the power of conditional execution in the ARM architecture.

Quiere decir que para el ejemplo 6.13 no tiene sentido hacer este branch sino se deberia utilizar banderas de codiciones. Porque praticamente solo son 2 instrucciones no un bloque de instrucciones.

In general, when a block of code has a single instruction, it is better to use conditional execution rather than branch around it. As the block becomes longer, the branch becomes valuable because it avoids wasting time fetching instructions that will not be executed.

if/else Statements

if/else statements execute one of two blocks of code depending on a condition. When the condition in the if statement is met, the *if block* is executed. Otherwise, the *else block* is executed. Code Example 6.14 shows an example if/else statement.

Like if statements, if/else assembly code tests the opposite condition of the one in the high-level code. In Code Example 6.14, the high-level code tests for `apples == oranges`, and the assembly code tests for `apples != oranges`. If that opposite condition is TRUE, `BNE` skips the if block and executes the else block. Otherwise, the if block executes and finishes with an unconditional branch (`B`) past the else block.

Code Example 6.14 IF/ELSE STATEMENT

High-Level Code

```
if (apples == oranges)
  f = i + 1;

else
  f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
  CMP R0, R1        ; apples == oranges?
  BNE L1            ; if not equal, skip if block
  ADD R2, R3, #1    ; if block: f = i + 1
  B   L2            ; skip else block
L1
  SUB R2, R2, R3    ; else block: f = f - i
L2
```

Again, because any instruction can conditionally execute and because the instructions within the if block do not change the condition flags, the ARM assembly code for Code Example 6.14 could also be written much more succinctly as:

```
CMP   R0, R1        ; apples == oranges?
ADDEQ R2, R3, #1    ; f = i + 1 on equality (i.e., Z = 1)
SUBNE R2, R2, R3    ; f = f - i on not equal (i.e., Z = 0)
```

switch/case Statements*

switch/case statements execute one of several blocks of code depending on the conditions. If no conditions are met, the *default block* is executed. A case statement is equivalent to a series of *nested* if/else statements. Code Example 6.15 shows two high-level code snippets with the same

Code Example 6.15 SWITCH/CASE STATEMENT

High-Level Code

```
switch (button) {
  case 1:   amt = 20;  break;

  case 2:   amt = 50;  break;

  case 3:   amt = 100; break;

  default: amt = 0;
}
// equivalent function using
// if/else statements
  if      (button == 1) amt = 20;
  else if (button == 2) amt = 50;
  else if (button == 3) amt = 100;
  else                  amt = 0;
```

ARM Assembly Code

```
; R0 = button, R1 = amt
  CMP   R0, #1          ; is button 1 ?
  MOVEQ R1, #20         ; amt = 20 if button is 1
  BEQ   DONE            ; break

  CMP   R0, #2          ; is button 2 ?
  MOVEQ R1, #50         ; amt = 50 if button is 2
  BEQ   DONE            ; break

  CMP   R0, #3          ; is button 3?
  MOVEQ R1, #100        ; amt = 100 if button is 3
  BEQ   DONE            ; break

  MOV   R1, #0          ; default amt = 0
DONE
```

functionality: they calculate whether to dispense $20, $50, or $100 from an ATM (automatic teller machine) depending on the button pressed. The ARM assembly implementation is the same for both high-level code snippets.

6.3.5 Getting Loopy

Loops repeatedly execute a block of code depending on a condition. while loops and for loops are common loop constructs used by high-level languages. This section shows how to translate them into ARM assembly language, taking advantage of conditional branching.

while Loops

while loops repeatedly execute a block of code until a condition is *not* met. The while loop in Code Example 6.16 determines the value of `x` such that $2^x = 128$. It executes seven times, until `pow` = 128.

The `int` data type in C refers to a word of data representing a two's complement integer. ARM uses 32-bit words, so an `int` represents a number in the range $[-2^{31}, 2^{31} - 1]$.

Like if/else statements, the assembly code for while loops tests the opposite condition of the one in the high-level code. If that opposite condition is TRUE (in this case, R0 == 128), the while loop is finished. If not (R0 ≠128), the branch isn't taken and the loop body executes.

Code Example 6.16 WHILE LOOP

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1      ; pow = 1
  MOV R1, #0      ; x = 0

WHILE
  CMP R0, #128    ; pow != 128 ?
  BEQ DONE        ; if pow == 128, exit loop
  LSL R0, R0, #1  ; pow = pow * 2
  ADD R1, R1, #1  ; x = x + 1
  B   WHILE       ; repeat loop
DONE
```

In Code Example 6.16, the while loop compares `pow` to 128 and exits the loop if it is equal. Otherwise it doubles `pow` (using a left shift), increments `x`, and branches back to the start of the while loop.

for Loops

It is very common to initialize a variable before a while loop, check that variable in the loop condition, and change that variable each time through the while loop. for loops are a convenient shorthand that combines the initialization, condition check, and variable change in one place. The format of the for loop is:

```
for (initialization; condition; loop operation)
  statement
```

Code Example 6.17 FOR LOOP

High-Level Code

```
int i;
int sum = 0;


for (i = 0; i < 10; i = i + 1) {
   sum = sum + i;
}
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0        ; sum = 0
  MOV R0, #0        ; i = 0          loop initialization

FOR
  CMP R0, #10       ; i < 10 ?       check condition
  BGE DONE          ; if (i >= 10) exit loop
  ADD R1, R1, R0    ; sum = sum + i  loop body
  ADD R0, R0, #1    ; i = i + 1      loop operation
  B   FOR           ; repeat loop
DONE
```

The initialization code executes before the for loop begins. The condition is tested at the beginning of each loop. If the condition is not met, the loop exits. The loop operation executes at the end of each loop.

Code Example 6.17 adds the numbers from 0 to 9. The loop variable, in this case `i`, is initialized to 0 and is incremented at the end of each loop iteration. The for loop executes as long as `i` is less than 10. Note that this example also illustrates relative comparisons. The loop checks the < condition to continue, so the assembly code checks the opposite condition, >=, to exit the loop.

Loops are especially useful for accessing large amounts of similar data stored in memory, which is discussed next.

6.3.6 Memory

For ease of storage and access, similar data can be grouped together into an *array*. An array stores its contents at sequential data addresses in memory. Each array element is identified by a number called its *index*. The number of elements in the array is called the *length* of the array.

Figure 6.8 shows a 200-element array of scores stored in memory. Code Example 6.18 is a grade inflation algorithm that adds 10 points to each of the scores. Note that the code for initializing the scores array is not shown. The index into the array is a variable (`i`) rather than a constant, so we must multiply it by 4 before adding it to the base address.

ARM can *scale* (multiply) the index, add it to the base address, and load from memory in a single instruction. Instead of the `LSL` and `LDR` instruction sequence in Code Example 6.18, we can use a single instruction:

```
LDR R3, [R0, R1, LSL #2]
```

R1 is scaled (shifted left by two) then added to the base address (R0). Thus, the memory address is R0 + (R1 × 4).

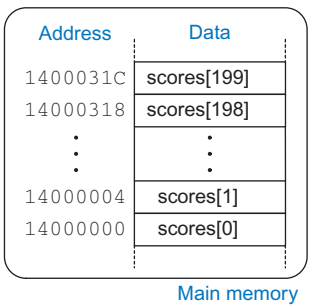


Figure 6.8 **Memory holding `scores[200]` starting at base address 0x14000000**

Code Example 6.18 ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)

  scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code      ...
  MOV R0, #0x14000000      ; R0 = base address
  MOV R1, #0               ; i = 0

LOOP
  CMP R1, #200             ; i < 200?
  BGE L3                   ; if i ≥ 200, exit loop
  LSL R2, R1, #2           ; R2 = i * 4
  LDR R3, [R0, R2]         ; R3 = scores[i]
  ADD R3, R3, #10          ; R3 = scores[i] + 10
  STR R3, [R0, R2]         ; scores[i] = scores[i] + 10
  ADD R1, R1, #1           ; i = i + 1
  B   LOOP                 ; repeat loop
L3
```

In addition to scaling the index register, ARM provides offset, pre-indexed, and post-indexed addressing to enable dense and efficient code for array accesses and function calls. Table 6.4 gives examples of each indexing mode. In each case, the base register is R1 and the offset is R2. The offset can be subtracted by writing –R2. The offset may also be an immediate in the range of 0–4095 that can be added (e.g., #20) or subtracted (e.g., #–20).

Offset addressing calculates the address as the base register ± the offset; the base register is unchanged. *Pre-indexed addressing* calculates the address as the base register ± the offset and updates the base register to this new address. *Post-indexed addressing* calculates the address as the base register only and then, after accessing memory, the base register is updated to the base register ± the offset. We have seen many examples of offset indexing mode. Code Example 6.19 shows the for loop from Code Example 6.18 rewritten to use post-indexing, eliminating the `ADD` to increment `i`.

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Code Example 6.19 FOR LOOP USING POST-INDEXING

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code    ...
  MOV R0, #0x14000000    ; R0 = base address
  ADD R1, R0, #800       ; R1 = base address + (200*4)

LOOP
  CMP R0, R1             ; reached end of array?
  BGE L3                 ; if yes, exit loop
  LDR R2, [R0]           ; R2 = scores[i]
  ADD R2, R2, #10        ; R2 = scores[i] + 10
  STR R2, [R0], #4       ; scores[i] = scores[i] + 10
                         ; then R0 = R0 + 4
  B   LOOP               ; repeat loop
  L3
```

Bytes and Characters

Numbers in the range [−128, 127] can be stored in a single byte rather than an entire word. Because there are much fewer than 256 characters on an English language keyboard, English characters are often represented by bytes. The C language uses the type `char` to represent a byte or character.

Early computers lacked a standard mapping between bytes and English characters, so exchanging text between computers was difficult. In 1963, the American Standards Association published the *American Standard Code for Information Interchange* (*ASCII*), which assigns each text character a unique byte value. Table 6.5 shows these character encodings for printable characters. The ASCII values are given in hexadecimal. Lowercase and uppercase letters differ by 0x20 (32).

ARM provides load byte (`LDRB`), load signed byte (`LDRSB`), and store byte (`STRB`) to access individual bytes in memory. `LDRB` zero-extends the byte, whereas `LDRSB` sign-extends the byte to fill the entire 32-bit register. `STRB` stores the least significant byte of the 32-bit register into the specified byte address in memory. All three are illustrated in Figure 6.9, with

Other programming languages, such as Java, use different character encodings, most notably Unicode. Unicode uses 16 bits to represent each character, so it supports accents, umlauts, and Asian languages. For more information, see www.unicode.org.

`LDRH`, `LDRSH`, and `STRH` are similar, but access 16-bit *halfwords*.

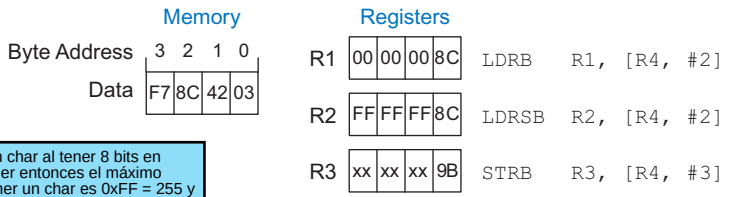


Figure 6.9 **Instructions for loading and storing bytes**

IMP: Note mae que un char al tener 8 bits en UFT-8 entonces al tener entonces el máximo número que puede tener un char es 0xFF = 255 y como un registro tiene 32 bits. Los primeros 8 bits son el char los restantes 24 bits, se puede decidir como almacenar. Importante una palabra (word o fila) entonces tiene apenas 4 letras.
LDRB - lee el char y rellena con 0
LDRSB - lee el char y rellena con el signo
STRB - escribe en memoria los primeros 8 bits en la posicion de la memoria que se pasa

ASCII codes developed from earlier forms of character encoding. Beginning in 1838, telegraph machines used Morse code, a series of dots (.) and dashes (–), to represent characters. For example, the letters A, B, C, and D were represented as – , – ... , – . – . , and – .. , respectively. The number of dots and dashes varied with each letter. For efficiency, common letters used shorter codes.

In 1874, Jean-Maurice-Emile Baudot invented a 5-bit code called the Baudot code. For example, A, B, C, and D were represented as 00011, 11001, 01110, and 01001.

However, the 32 possible encodings of this 5-bit code were not sufficient for all the English characters, but 8-bit encoding was. Thus, as electronic communication became prevalent, 8-bit ASCII encoding emerged as the standard.

Table 6.5 ASCII encodings

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	‘	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

the base address R4 being 0. `LDRB` loads the byte at memory address 2 into the least significant byte of R1 and fills the remaining register bits with 0. `LDRSB` loads this byte into R2 and sign-extends the byte into the upper 24 bits of the register. `STRB` stores the least significant byte of R3 (0x9B) into memory byte 3; it replaces 0xF7 with 0x9B. The more significant bytes of R3 are ignored.

A series of characters is called a *string*. Strings have a variable length, so programming languages must provide a way to determine the length or end of the string. In C, the null character (0x00) signifies the end of a string. For example, Figure 6.10 shows the string “Hello!” (0x48 65 6C 6C 6F 21 00) stored in memory. The string is seven bytes long

Example 6.2 USING LDRB AND STRB TO ACCESS A CHARACTER ARRAY

The following high-level code converts a 10-entry array of characters from lowercase to uppercase by subtracting 32 from each array entry. Translate it into ARM assembly language. Remember that the address difference between array elements is now 1 byte, not 4 bytes. Assume that R0 already holds the base address of `chararray`.

```
// high-level code
// chararray[10] declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
  chararray[i] = chararray[i] - 32;
```

Solution:

```
; ARM assembly code
; R0 = base address of chararray (initialized earlier), R1 = i
            MOV   R1, #0            ; i = 0
LOOP        CMP   R1, #10           ; i < 10 ?
            BGE   DONE              ; if (i >=10), exit loop
            LDRB  R2, [R0, R1]      ; R2 = mem[R0+R1] = chararray[i]
            SUB   R2, R2, #32       ; R2 = chararray[i] - 32
            STRB  R2, [R0, R1]      ; chararray[i] = R2
            ADD   R1, R1, #1        ; i = i + 1
            B     LOOP              ; repeat loop
DONE
```

and extends from address 0x1522FFF0 to 0x1522FFF6. The first character of the string (H = 0x48) is stored at the lowest byte address (0x1522FFF0).

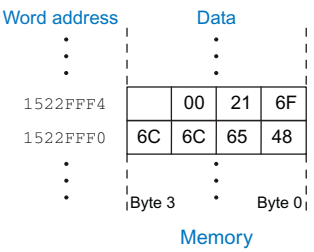


Figure 6.10 **The string "Hello!" stored in memory**

6.3.7 Function Calls

High-level languages support *functions* (also called *procedures* or *subroutines*) to reuse common code and to make a program more modular and readable. Functions have inputs, called *arguments*, and an output, called the *return value*. Functions should calculate the return value and cause no other unintended side effects.

When one function calls another, the calling function, the *caller*, and the called function, the *callee*, must agree on where to put the arguments and the return value. In ARM, the caller conventionally places up to four arguments in registers R0–R3 before making the function call,

and the callee places the return value in register R0 before finishing. By following this convention, both functions know where to find the arguments and return value, even if the caller and callee were written by different people.

The callee must not interfere with the behavior of the caller. This means that the callee must know where to return to after it completes and it must not trample on any registers or memory needed by the caller. The caller stores the return address in the link register LR at the same time it jumps to the callee using the branch and link instruction (`BL`). The callee must not overwrite any architectural state or memory that the caller is depending on. Specifically, the callee must leave the *saved registers* (R4–R11, and LR) and the *stack*, a portion of memory used for temporary variables, unmodified.

This section shows how to call and return from a function. It shows how functions access input arguments and the return value and how they use the stack to store temporary variables.

Function Calls and Returns

ARM uses the branch and link instruction (`BL`) to call a function and moves the link register to the PC (`MOV PC, LR`) to return from a function. Code Example 6.20 shows the `main` function calling the `simple` function. `main` is the caller, and `simple` is the callee. The `simple` function is called with no input arguments and generates no return value; it just returns to the caller. In Code Example 6.20, instruction addresses are given to the left of each ARM instruction in hexadecimal.

MOV PC, LR es el retorno de una funcion

`BL` (branch and link) and `MOV PC, LR` are the two essential instructions needed for a function call and return. `BL` performs two tasks: it stores the *return address* of the next instruction (the instruction

Code Example 6.20 simple FUNCTION CALL

High-Level Code

```
int main() {

  simple();
  ...
}

// void means the function returns no value
void simple() {
  return;
}
```

ARM Assembly Code

```
0x00008000 MAIN     ...
...                 ...
0x00008020          BL  SIMPLE    ; call the simple function
...


0x0000902C  SIMPLE  MOV PC, LR    ; return
```

after BL) in the link register (LR), and it branches to the target instruction.

In Code Example 6.20, the main function calls the simple function by executing the branch and link instruction (BL). BL branches to the SIMPLE label and stores 0x00008024 in LR. The simple function returns immediately by executing the instruction MOV PC, LR, copying the return address from the LR back to the PC. The main function then continues executing at this address (0x00008024).

Input Arguments and Return Values

The simple function in Code Example 6.20 receives no input from the calling function (main) and returns no output. By ARM convention, functions use R0–R3 for input arguments and R0 for the return value. In Code Example 6.21, the function diffofsums is called with four arguments and returns one result. result is a local variable, which we choose to keep in R4.

According to ARM convention, the calling function, main, places the function arguments from left to right into the input registers, R0–R3. The called function, diffofsums, stores the return value in the return register, R0. When a function with more than four arguments is called, the additional input arguments are placed on the stack, which we discuss next.

Remember that PC and LR are alternative names for R15 and R14, respectively. ARM is unusual in that PC is part of the register set, so a function return can be done with a MOV instruction. Many other instruction sets keep the PC in a special register and use a special return or jump instruction to return from functions.

These days, ARM compilers do a function return using BX LR. The BX branch and exchange instruction is like a branch, but it also can transition between the standard ARM instruction set and the Thumb instruction set described in Section 6.7.1. This chapter doesn't use the Thumb or BX instructions and thus sticks with the ARMv4 MOV PC, LR method.

We will see in Chapter 7 that treating the PC as an ordinary register complicates the implementation of the processor.

Code Example 6.21 FUNCTION CALL WITH ARGUMENTS AND RETURN VALUES

High-Level Code

```
int main() {
  int y;
  . . .
  y = diffofsums(2, 3, 4, 5);
  . . .
}



int diffofsums(int f, int g, int h, int i) {
  int result;

  result = (f + g) - (h + i);
  return result;
}
```

ARM Assembly Code

```
; R4 = y
MAIN
  . . .
  MOV R0, #2        ; argument 0 = 2
  MOV R1, #3        ; argument 1 = 3
  MOV R2, #4        ; argument 2 = 4
  MOV R3, #5        ; argument 3 = 5
  BL  DIFFOFSUMS    ; call function
  MOV R4, R0        ; y = returned value
  . . .

; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1    ; R8 = f + g
  ADD R9, R2, R3    ; R9 = h + i
  SUB R4, R8, R9    ; result = (f + g) - (h + i)
  MOV R0, R4        ; put return value in R0
  MOV PC, LR        ; return to caller
```

Code Example 6.21 has some subtle errors. Code Examples 6.22–6.25 show improved versions of the program.

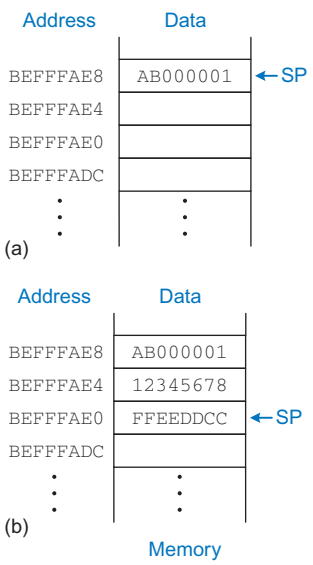


Figure 6.11 **The stack (a) before expansion and (b) after two-word expansion**

The stack is typically stored upside down in memory such that the top of the stack is actually the lowest address and the stack grows downward toward lower memory addresses. This is called a *descending stack*. ARM also allows for *ascending stacks* that grow up toward higher memory addresses. The stack pointer typically points to the topmost element on the stack; this is called a *full stack*. ARM also allows for *empty stacks* in which SP points one word beyond the top of the stack. The ARM *Application Binary Interface* (ABI) defines a standard way in which functions pass variables and use the stack so that libraries developed by different compilers can interoperate. It specifies a *full descending* stack, which we will use in this chapter.

The Stack

OJO el Stack es la RAM!

The stack is memory that is used to save information within a function. The stack expands (uses more memory) as the processor needs more scratch space and contracts (uses less memory) when the processor no longer needs the variables stored there. Before explaining how functions use the stack to store temporary values, we explain how the stack works.

The stack is a last-in-first-out (LIFO) queue. Like a stack of dishes, the last item *pushed* onto the stack (the top dish) is the first one that can be *popped* off. Each function may allocate stack space to store local variables but must deallocate it before returning. The *top of the stack* is the most recently allocated space. Whereas a stack of dishes grows up in space, the ARM stack grows down in memory. The stack expands to lower memory addresses when a program needs more scratch space.

Figure 6.11 shows a picture of the stack. The stack pointer, SP (R13), is an ordinary ARM register that, by convention, *points* to the *top of the stack*. A pointer is a fancy name for a memory address. SP points to (gives the address of) data. For example, in Figure 6.11(a), the stack pointer, SP, holds the address value 0XBEFFFAE8 and points to the data value 0xAB000001.

The stack pointer (SP) starts at a high memory address and decrements to expand as needed. Figure 6.11(b) shows the stack expanding to allow two more data words of temporary storage. To do so, SP decrements by eight to become 0xBEFFFAE0. Two additional data words, 0x12345678 and 0xFFEEDDCC, are temporarily stored on the stack.

One of the important uses of the stack is to save and restore registers that are used by a function. Recall that a function should calculate a return value but have no other unintended side effects. In particular, it should not modify any registers besides R0, the one containing the return value. The `diffofsums` function in Code Example 6.21 violates this rule because it modifies R4, R8, and R9. If `main` had been using these registers before the call to `diffofsums`, their contents would have been corrupted by the function call.

To solve this problem, a function saves registers on the stack before it modifies them, then restores them from the stack before it returns. Specifically, it performs the following steps:

1. Makes space on the stack to store the values of one or more registers
2. Stores the values of the registers on the stack
3. Executes the function using the registers
4. Restores the original values of the registers from the stack
5. Deallocates space on the stack

Code Example 6.22 shows an improved version of `diffofsums` that saves and restores R4, R8, and R9. Figure 6.12 shows the stack before, during, and after a call to the `diffofsums` function from Code Example 6.22. The stack starts at 0xBEF0F0FC. `diffofsums` makes room for three words on the stack by decrementing the stack pointer SP by 12. It then stores the current values held in R4, R8, and R9 in the newly allocated space. It executes the rest of the function, changing the values in these three registers. At the end of the function, `diffofsums` restores the values of these registers from the stack, deallocates its stack space, and returns. When the function returns, R0 holds the result, but there

La razon por la cual se almacena los valores previos en los registros de la pila, es porque en un asm no se puede operar ningun dato que no sea un registro por lo tanto. Una pila almacena valores previous de la funcion y no las varaibles de la funcion.

Code Example 6.22 FUNCTION SAVING REGISTERS ON THE STACK

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
  SUB   SP, SP, #12      ; make space on stack for 3 registers
  STR   R9, [SP, #8]     ; save R9 on stack
  STR   R8, [SP, #4]     ; save R8 on stack
  STR   R4, [SP]         ; save R4 on stack

  ADD  R8, R0, R1        ; R8 = f + g
  ADD  R9, R2, R3        ; R9 = h + i
  SUB  R4, R8, R9        ; result = (f + g) - (h + i)
  MOV  R0, R4            ; put return value in R0

  LDR   R4, [SP]         ; restore R4 from stack
  LDR   R8, [SP, #4]     ; restore R8 from stack
  LDR   R9, [SP, #8]     ; restore R9 from stack
  ADD   SP, SP, #12      ; deallocate stack space

  MOV   PC, LR           ; return to caller
```

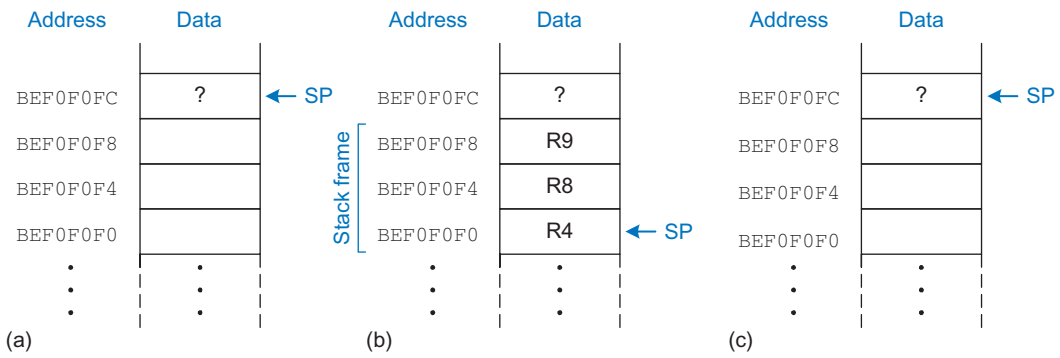


Figure 6.12 The stack: (a) before, (b) during, and (c) after the `diffofsums` function call

are no other side effects: R4, R8, R9, and SP have the same values as they did before the function call.

The stack space that a function allocates for itself is called its *stack frame*. `diffofsums`'s stack frame is three words deep. The principle of modularity tells us that each function should access only its own stack frame, not the frames belonging to other functions.

Loading and Storing Multiple Registers

Saving and restoring registers on the stack is such a common operation that ARM provides Load Multiple and Store Multiple instructions (`LDM` and `STM`) that are optimized to this purpose. Code Example 6.23 rewrites `diffofsums` using these instructions. The stack holds exactly the same information as in the previous example, but the code is much shorter.

Code Example 6.23 SAVING AND RESTORING MULTIPLE REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  STMFD SP!, {R4, R8, R9}        ; push R4/8/9 on full descending stack

  ADD   R8, R0, R1               ; R8 = f + g
  ADD   R9, R2, R3               ; R9 = h + i
  SUB   R4, R8, R9               ; result = (f + g) - (h + i)
  MOV   R0, R4                   ; put return value in R0

  LDMFD SP!, {R4, R8, R9}        ; pop R4/8/9 off full descending stack

  MOV   PC, LR                   ; return to caller
```

`LDM` and `STM` come in four flavors for full and empty descending and ascending stacks (`FD`, `ED`, `FA`, `EA`). The `SP!` in the instructions indicates to store the data relative to the stack pointer and to update the stack pointer after the store or load. `PUSH` and `POP` are synonyms for `STMFD SP!, {regs}` and `LDMFD SP!, {regs}`, respectively, and are the preferred way to save registers on the conventional full descending stack.

Preserved Registers

Code Examples 6.22 and 6.23 assume that all of the used registers (R4, R8, and R9) must be saved and restored. If the calling function does not use those registers, the effort to save and restore them is wasted. To avoid this waste, ARM divides registers into *preserved* and *nonpreserved* categories. The preserved registers include R4–R11. The nonpreserved registers are R0–R3 and R12. SP and LR (R13 and R14)

must also be preserved. A function must save and restore any of the preserved registers that it wishes to use, but it can change the nonpreserved registers freely.

Code Example 6.24 shows a further improved version of `diffofsums` that saves only R4 on the stack. It also illustrates the preferred `PUSH` and `POP` synonyms. The code reuses the nonpreserved argument registers R1 and R3 to hold the intermediate sums when those arguments are no longer necessary.

Code Example 6.24 REDUCING THE NUMBER OF PRESERVED REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
   PUSH  {R4}                  ; save R4 on stack
   ADD   R1, R0, R1            ; R1 = f + g
   ADD   R3, R2, R3            ; R3 = h + i
   SUB   R4, R1, R3            ; result = (f + g) - (h + i)
   MOV   R0, R4                ; put return value in R0
   POP   {R4}                  ; pop R4 off stack
   MOV   PC, LR                ; return to caller
```

`PUSH` (and `POP`) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction. For example, `PUSH {R8, R1, R3}` will store R1 at the lowest memory address, then R3 and finally R8 at the next higher memory addresses on the stack.

Remember that when one function calls another, the former is the caller and the latter is the callee. The callee must save and restore any preserved registers that it wishes to use. The callee may change any of the nonpreserved registers. Hence, if the caller is holding active data in a nonpreserved register, the caller needs to save that nonpreserved register before making the function call and then needs to restore it afterward. For these reasons, preserved registers are also called *callee-save*, and nonpreserved registers are called *caller-save*.

Table 6.6 summarizes which registers are preserved. R4–R11 are generally used to hold local variables within a function, so they must be saved. LR must also be saved, so that the function knows where to return.

Table 6.6 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: R4–R11	Temporary register: R12
Stack pointer: SP (R13)	Argument registers: R0–R3
Return address: LR (R14)	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

The convention of which registers are preserved or not preserved is part of the Procedure Call Standard for the ARM Architecture, rather than of the architecture itself. Alternate procedure call standards exist.

R0–R3 and R12 are used to hold temporary results. These calculations typically complete before a function call is made, so they are not preserved, and it is rare that the caller needs to save them.

R0–R3 are often overwritten in the process of calling a function. Hence, they must be saved by the caller if the caller depends on any of its own arguments after a called function returns. R0 certainly should not be preserved, because the callee returns its result in this register. Recall that the Current Program Status Register (CPSR) holds the condition flags. It is not preserved across function calls.

The stack above the stack pointer is automatically preserved as long as the callee does not write to memory addresses above SP. In this way, it does not modify the stack frame of any other functions. The stack pointer itself is preserved, because the callee deallocates its stack frame before returning by adding back the same amount that it subtracted from SP at the beginning of the function.

The astute reader or an optimizing compiler may notice that the local variable `result` is immediately returned without being used for anything else. Hence, we can eliminate the variable and simply store it in the return register R0, eliminating the need to push and pop R4 and to move `result` from R4 to R0. Code Example 6.25 shows this even further optimized `diffofsums`.

Nonleaf Function Calls

A function that does not call others is called a *leaf function*; `diffofsums` is an example. A function that does call others is called a *nonleaf function*. As mentioned, nonleaf functions are somewhat more complicated because they may need to save nonpreserved registers on the stack before they call another function and then restore those registers afterward. Specifically:

Caller save rule: Before a function call, the caller must save any nonpreserved registers (R0–R3 and R12) that it needs after the call. After the call, it must restore these registers before using them.

Callee save rule: Before a callee disturbs any of the preserved registers (R4–R11 and LR), it must save the registers. Before it returns, it must restore these registers.

Code Example 6.25 OPTIMIZED `diffofsums` FUNCTION CALL

ARM Assembly Code

```
DIFFOFSUMS
  ADD  R1, R0, R1    ; R1 = f + g
  ADD  R3, R2, R3    ; R3 = h + i
  SUB  R0, R1, R3    ; return (f + g) - (h + i)
  MOV  PC, LR        ; return to caller
```

Code Example 6.26 demonstrates a nonleaf function `f1` and a leaf function `f2` including all the necessary saving and preserving of registers. Suppose `f1` keeps `i` in R4 and `x` in R5. `f2` keeps `r` in R4. `f1` uses preserved registers R4, R5, and LR, so it initially pushes them on the stack according to the callee save rule. It uses R12 to hold the intermediate result `(a – b)` so that it does not need to preserve another register for this calculation. Before calling `f2`, `f1` pushes R0 and R1 onto the stack according to the caller save rule because these are nonpreserved registers that `f2` might change and that `f1` will still need after the call. Although R12 is also a nonpreserved register that `f2` could overwrite, `f1` no longer needs R12 and doesn't have to save it. `f1` then passes the argument to `f2` in R0, makes the function call, and uses the result in R0. `f1` then restores R0 and R1 because it still needs them. When `f1` is done, it puts the return value in R0, restores preserved registers R4, R5, and LR, and returns. `f2` saves and restores R4 according to the callee save rule.

A nonleaf function overwrites LR when it calls another function using `BL`. Thus, a nonleaf function must always save LR on its stack and restore it before returning.

Code Example 6.26 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
  PUSH {R4, R5, LR}   ; save preserved registers used by f1
  ADD  R5, R0, R1     ; x = (a + b)
  SUB  R12, R0, R1    ; temp = (a - b)
  MUL  R5, R5, R12    ; x = x * temp = (a + b) * (a - b)
  MOV  R4, #0         ; i = 0
FOR
  CMP  R4, R0         ; i < a?
  BGE  RETURN         ; no: exit loop
  PUSH {R0, R1}       ; save nonpreserved registers
  ADD  R0, R1, R4     ; argument is b + i
  BL   F2             ; call f2(b+i)
  ADD  R5, R5, R0     ; x = x + f2(b+i)
  POP  {R0, R1}       ; restore nonpreserved registers
  ADD  R4, R4, #1     ; i++
  B    FOR            ; continue for loop
RETURN
  MOV  R0, R5         ; return value is x
  POP  {R4, R5, LR}   ; restore preserved registers
  MOV  PC, LR         ; return from f1

; R0 = p, R4 = r
F2
  PUSH {R4}           ; save preserved registers used by f2
  ADD  R4, R0, 5      ; r = p + 5
  ADD  R0, R4, R0     ; return value is r + p
  POP  {R4}           ; restore preserved registers
  MOV  PC, LR         ; return from f2
```

On careful inspection, one might note that `f2` does not modify R1, so `f1` did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call.

An optimizing compiler could observe that `f2` is a leaf procedure and could allocate `r` to a nonpreserved register, avoiding the need to save and restore R4.

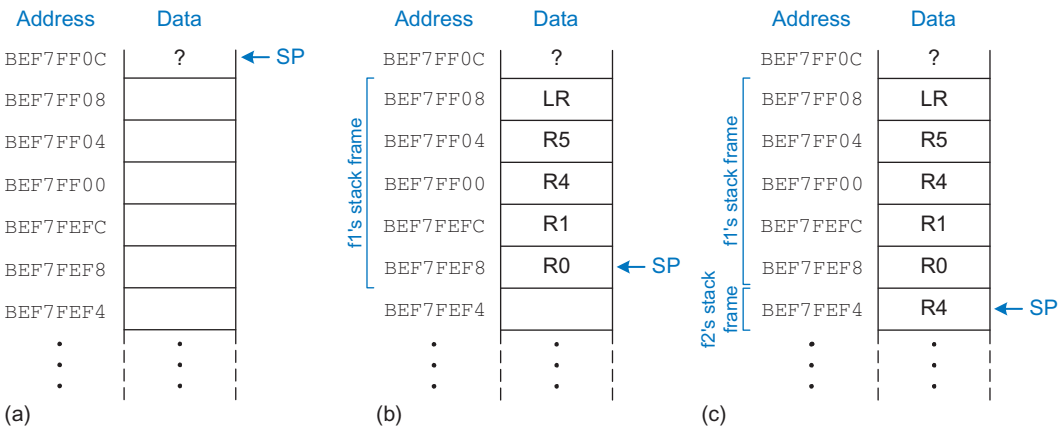


Figure 6.13 **The stack: (a) before function calls, (b) during `f1`, and (c) during `f2`**

Figure 6.13 shows the stack during execution of `f1`. The stack pointer originally starts at 0xBEF7FF0C.

Recursive Function Calls

A *recursive function* is a nonleaf function that calls itself. Recursive functions behave as both caller and callee and must save both preserved and nonpreserved registers. For example, the factorial function can be written as a recursive function. Recall that $factorial(n) = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$. The factorial function can be rewritten recursively as $factorial(n) = n \times factorial(n - 1)$, as shown in Code Example 6.27. The factorial of 1 is simply 1. To conveniently refer to program addresses, we show the program starting at address 0x8500.

According to the callee save rule, `factorial` is a nonleaf function and must save LR. According to the caller save rule, `factorial` will need `n` after calling itself, so it must save R0. Hence, it pushes both registers onto the stack at the start. It then checks whether `n` ≤ 1. If so, it puts the return value of 1 in R0, restores the stack pointer, and returns to the caller. It does not have to reload LR and R0 in this case, because they were never modified. If `n` > 1, the function recursively calls `factorial(n - 1)`. It then restores the value of `n` and the link register (LR) from the stack, performs the multiplication, and returns this result. Notice that the function cleverly restores `n` into R1, so as not to overwrite the returned value. The multiply instruction (`MUL R0, R1, R0`) multiplies `n` (R1) and the returned value (R0) and puts the result in R0.

Code Example 6.27 `factorial` RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;



  else
    return (n * factorial(n - 1));
}
```

ARM Assembly Code

```
0x8500  FACTORIAL  PUSH  {R0, LR}    ; push n and LR on stack
0x8504             CMP   R0, #1      ; R0 <= 1?
0x8508             BGT   ELSE        ; no: branch to else
0x850C             MOV   R0, #1      ; otherwise, return 1
0x8510             ADD   SP, SP, #8  ; restore SP
0x8514             MOV   PC, LR      ; return
0x8518  ELSE       SUB   R0, R0, #1  ; n = n - 1
0x851C             BL    FACTORIAL   ; recursive call
0x8520             POP   {R1, LR}    ; pop n (into R1) and LR
0x8524             MUL   R0, R1, R0  ; R0 = n * factorial(n - 1)
0x8528             MOV   PC, LR      ; return
```

Figure 6.14 shows the stack when executing `factorial(3)`. For illustration, we show SP initially pointing to 0xBEFF0FF0, as shown in Figure 6.14(a). The function creates a two-word stack frame to hold `n` (R0) and LR. On the first invocation, `factorial` saves R0 (holding `n` = 3) at 0xBEFF0FE8 and LR at 0xBEFF0FEC, as shown in Figure 6.14(b). The function then changes `n` to 2 and recursively calls `factorial(2)`, making LR hold 0x8520. On the second invocation, it saves R0 (holding n=2) at 0xBEFF0FE0 and LR at 0xBEFF0FE4. This time, we know that LR contains 0x8520. The function then changes `n` to 1 and recursively calls `factorial(1)`. On the third invocation, it saves R0 (holding `n` = 1) at

For clarity, we will always save registers at the start of a procedure call. An optimizing compiler might observe that there is no need to save R0 and LR when $n \leq 1$, and thus push registers only in the `ELSE` portion of the function.

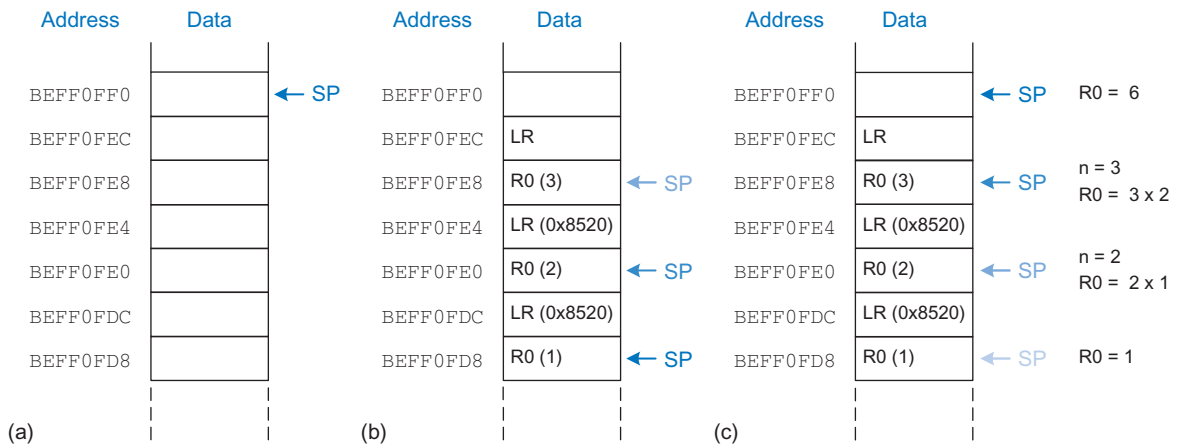


Figure 6.14 **Stack: (a) before, (b) during, and (c) after factorial function call with `n` = 3**

0xBEFF0FD8 and LR at 0xBEFF0FDC. This time, LR again contains 0x8520. The third invocation of `factorial` returns the value 1 in R0 and deallocates the stack frame before returning to the second invocation. The second invocation restores `n` (into R1) to 2, restores LR to 0x8520 (it happened to already have this value), deallocates the stack frame, and returns $R0 = 2 \times 1 = 2$ to the first invocation. The first invocation restores `n` (into R1) to 3, restores LR to the return address of the caller, deallocates the stack frame, and returns $R0 = 3 \times 2 = 6$. Figure 6.14(c) shows the stack as the recursively called functions return. When `factorial` returns to the caller, the stack pointer is in its original position (0xBEFF0FF0), none of the contents of the stack above the pointer have changed, and all of the preserved registers hold their original values. R0 holds the return value, 6.

Additional Arguments and Local Variables*

Functions may have more than four input arguments and may have too many local variables to keep in preserved registers. The stack is used to store this information. By ARM convention, if a function has more than four arguments, the first four are passed in the argument registers as usual. Additional arguments are passed on the stack, just above SP. The caller must expand its stack to make room for the additional arguments. Figure 6.15(a) shows the caller's stack for calling a function with more than four arguments.

A function can also declare local variables or arrays. Local variables are declared within a function and can be accessed only within that function. Local variables are stored in R4–R11; if there are too many local variables, they can also be stored in the function's stack frame. In particular, local arrays are stored on the stack.

Figure 6.15(b) shows the organization of a callee's stack frame. The stack frame holds the temporary registers and link register (if they need to be saved because of a subsequent function call), and any of the saved

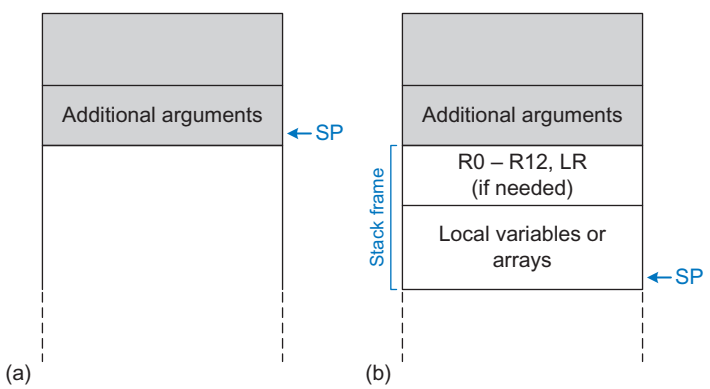


Figure 6.15 **Stack usage: (a) before and (b) after call**

registers that the function will modify. It also holds local arrays and any excess local variables. If the callee has more than four arguments, it finds them in the caller's stack frame. Accessing additional input arguments is the one exception in which a function can access stack data not in its own stack frame.

6.4 MACHINE LANGUAGE

Assembly language is convenient for humans to read. However, digital circuits understand only 1's and 0's. Therefore, a program written in assembly language is translated from mnemonics to a representation using only 1's and 0's called *machine language*. This section describes ARM machine language and the tedious process of converting between assembly and machine language.

ARM uses 32-bit instructions. Again, regularity supports simplicity, and the most regular choice is to encode all instructions as words that can be stored in memory. Even though some instructions may not require all 32 bits of encoding, variable-length instructions would add complexity. Simplicity would also encourage a single instruction format, but that is too restrictive. However, this issue allows us to introduce the last design principle:

Design Principle 4: Good design demands good compromises.

ARM makes the compromise of defining three main instruction formats: *Data-processing*, *Memory*, and *Branch*. This small number of formats allows for some regularity among instructions, and thus simpler decoder hardware, while also accommodating different instruction needs. Data-processing instructions have a first source register, a second source that is either an immediate or a register, possibly shifted, and a destination register. The Data-processing format has several variations for these second sources. Memory instructions have three operands: a base register, an offset that is either an immediate or an optionally shifted register, and a register that is the destination on an `LDR` and another source on an `STR`. Branch instructions take one 24-bit immediate branch offset. This section discusses these ARM instruction formats and shows how they are encoded into binary. Appendix B provides a quick reference for all the ARMv4 instructions.

6.4.1 Data-processing Instructions

The data-processing instruction format is the most common. The first source operand is a register. The second source operand can be an immediate or an optionally shifted register. A third register is the destination. Figure 6.16 shows the data-processing instruction format. The 32-bit instruction has six fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*.

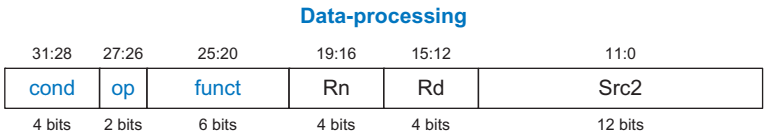


Figure 6.16 **Data-processing instruction format**

The operation the instruction performs is encoded in the fields highlighted in blue: *op* (also called the opcode or operation code) and *funct* or function code; the *cond* field encodes conditional execution based on flags described in Section 6.3.2. Recall that $cond = 1110_2$ for unconditional instructions. *op* is 00_2 for data-processing instructions.

The operands are encoded in the three fields: *Rn*, *Rd*, and *Src2*. *Rn* is the first source register and *Src2* is the second source; *Rd* is the destination register.

Figure 6.17 shows the format of the *funct* field and the three variations of *Src2* for data-processing instructions. *funct* has three subfields: *I*, *cmd*, and *S*. The *I*-bit is 1 when *Src2* is an immediate. The *S*-bit is 1 when the instruction sets the condition flags. For example, `SUBS R1, R9, #11` has $S = 1$. *cmd* indicates the specific data-processing instruction, as given in Table B.1 in Appendix B. For example, *cmd* is 4 (0100_2) for `ADD` and 2 (0010_2) for `SUB`.

Rd is short for "register destination." *Rn* and *Rm* unintuitively indicate the first and second register sources.

Three variations of *Src2* encoding allow the second source operand to be (1) an immediate, (2) a register (*Rm*) optionally shifted by a constant (*shamt5*), or (3) a register (*Rm*) shifted by another register (*Rs*). For the latter two encodings of *Src2*, *sh* encodes the type of shift to perform, as will be shown in Table 6.8.

Data-processing instructions have an unusual immediate representation involving an 8-bit unsigned immediate, *imm8*, and a 4-bit rotation, *rot*. *imm8* is rotated right by $2 \times rot$ to create a 32-bit constant. Table 6.7 gives example rotations and resulting 32-bit constants for the 8-bit immediate 0xFF. This representation is valuable because it

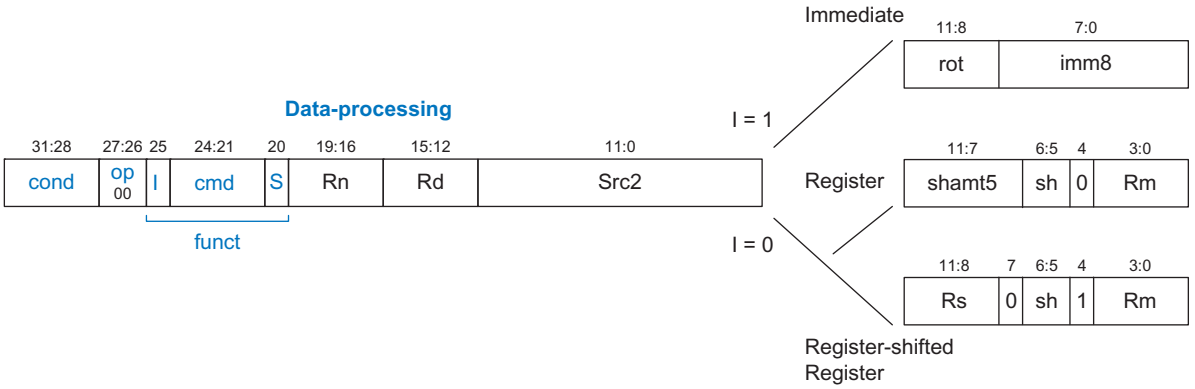


Figure 6.17 **Data-processing instruction format showing the *funct* field and *Src2* variations**

Table 6.7 Immediate rotations and resulting 32-bit constant for *imm8* = 0xFF

rot	32-bit Constant
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
…	…
1111	0000 0000 0000 0000 0000 0011 1111 1100

If an immediate has multiple possible encodings, the representation with the smallest rotation value *rot* is used. For example, #12 would be represented as (*rot*, *imm8*) = (0000, 00001100), not (0001, 00110000).

permits many useful constants, including small multiples of any power of two, to be packed into a small number of bits. Section 6.6.1 describes how to generate arbitrary 32-bit constants.

Figure 6.18 shows the machine code for `ADD` and `SUB` when *Src2* is a register. The easiest way to translate from assembly to machine code is to write out the values of each field and then convert these values to binary. Group the bits into blocks of four to convert to hexadecimal to make the machine language representation more compact. Beware that the destination is the first register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. *Rn* and *Rm* are the first and second source operands, respectively. For example, the assembly instruction `ADD R5, R6, R7` has *Rn* = 6, *Rd* = 5, and *Rm* = 7.

Figure 6.19 shows the machine code for `ADD` and `SUB` with an immediate and two register operands. Again, the destination is the first

Assembly Code

`ADD R5, R6, R7`
(0xE0865007)
`SUB R8, R9, R10`
(0xE049800A)

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
1110_2	00_2	0	0010_2	0	9	8	0	0	0	10
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0100	0	0110	0101	00000	00	0	0111
1110	00	0	0010	0	1001	1000	00000	00	0	1010
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Figure 6.18 Data-processing instructions with three register operands

Assembly Code

`ADD R0, R1, #42`
(0xE281002A)
`SUB R2, R3, #0xFF0`
(0xE2432EFF)

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110	00	1	0100	0	0001	0000	0000	00101010
1110	00	1	0010	0	0011	0010	1110	11111111
cond	op	I	cmd	S	Rn	Rd	rot	imm8

Figure 6.19 Data-processing instructions with an immediate and two register operands

register in an assembly language instruction, but it is the second register field (*Rd*) in the machine language instruction. The immediate of the `ADD` instruction (42) can be encoded in 8 bits, so no rotation is needed (*imm8* = 42, *rot* = 0). However, the immediate of `SUB R2, R3, 0xFF0` cannot be encoded directly using the 8 bits of *imm8*. Instead, *imm8* is 255 (0xFF), and it is rotated right by 28 bits (*rot* = 14). This is easiest to interpret by remembering that the right rotation by 28 bits is equivalent to a left rotation by 32−28 = 4 bits.

Shifts are also data-processing instructions. Recall from Section 6.3.1 that the amount by which to shift can be encoded using either a 5-bit immediate or a register.

Figure 6.20 shows the machine code for logical shift left (`LSL`) and rotate right (`ROR`) with immediate shift amounts. The *cmd* field is 13 (1101_2) for all shift instruction, and the shift field (*sh*) encodes the type of shift to perform, as given in Table 6.8. *Rm* (i.e., R5) holds the 32-bit value to be shifted, and *shamt5* gives the number of bits to shift. The shifted result is placed in *Rd*. *Rn* is not used and should be 0.

Figure 6.21 shows the machine code for `LSR` and `ASR` with the shift amount encoded in the least significant 8 bits of *Rs* (R6 and R12). As

Table 6.8 *sh* field encodings

Instruction	sh	Operation
`LSL`	00_2	Logical shift left
`LSR`	01_2	Logical shift right
`ASR`	10_2	Arithmetic shift right
`ROR`	11_2	Rotate right

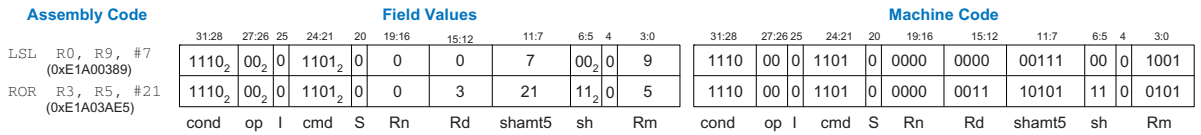


Figure 6.20 Shift instructions with immediate shift amounts

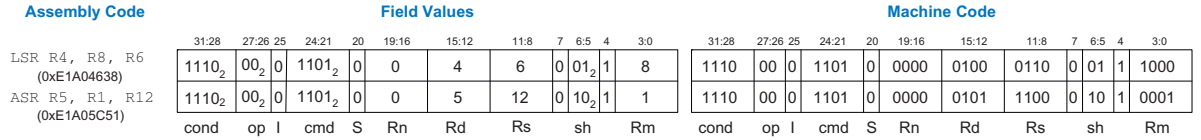


Figure 6.21 Shift instructions with register shift amounts

Table 6.9 Offset type control bits for memory instructions

Bit	Meaning	
	$\bar{I}$	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

before, *cmd* is 13 (1101_2), *sh* encodes the type of shift, *Rm* holds the value to be shifted, and the shifted result is placed in *Rd*. This instruction uses the *register-shifted register* addressing mode, where one register (*Rm*) is shifted by the amount held in a second register (*Rs*). Because the least significant 8 bits of *Rs* are used, *Rm* can be shifted by up to 255 positions. For example, if *Rs* holds the value 0xF001001C, the shift amount is 0x1C (28). A logical shift by more than 31 bits pushes all the bits off the end and produces all 0's. Rotate is cyclical, so a rotate by 50 bits is equivalent to a rotate by 18 bits.

Table 6.10 Index mode control bits for memory instructions

P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

6.4.2 Memory Instructions

Memory instructions use a format similar to that of data-processing instructions, with the same six overall fields: *cond*, *op*, *funct*, *Rn*, *Rd*, and *Src2*, as shown in Figure 6.22. However, memory instructions use a different *funct* field encoding, have two variations of *Src2*, and use an *op* of 01_2. *Rn* is the base register, *Src2* holds the offset, and *Rd* is the destination register in a load or the source register in a store. The offset is either a 12-bit unsigned immediate (*imm12*) or a register (*Rm*) that is optionally shifted by a constant (*shamt5*). *funct* is composed of six control bits: $\bar{I}$, *P*, *U*, *B*, *W*, and *L*. The $\bar{I}$ (immediate) and *U* (add) bits determine whether the offset is an immediate or register and whether it should be added or subtracted, according to Table 6.9. The *P* (pre-index) and *W* (writeback) bits specify the index mode according to Table 6.10. The *L* (load) and *B* (byte) bits specify the type of memory operation according to Table 6.11.

Table 6.11 Memory operation type control bits for memory instructions

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

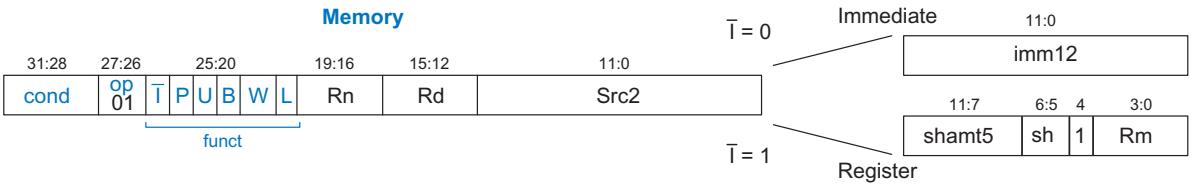


Figure 6.22 Memory instruction format for LDR, STR, LDRB, and STRB

Example 6.3 TRANSLATING MEMORY INSTRUCTIONS INTO MACHINE LANGUAGE

Translate the following assembly language statement into machine language.

```
STR R11, [R5], #-26
```

Notice the counterintuitive encoding of post-indexing mode.

Solution: `STR` is a memory instruction, so it has an *op* of 01_2. According to Table 6.11, $L = 0$ and $B = 0$ for `STR`. The instruction uses post-indexing, so according to Table 6.10, $P = 0$ and $W = 0$. The immediate offset is subtracted from the base, so $\overline{I} = 0$ and $U = 0$. Figure 6.23 shows each field and the machine code. Hence, the machine language instruction is 0xE405B01A.

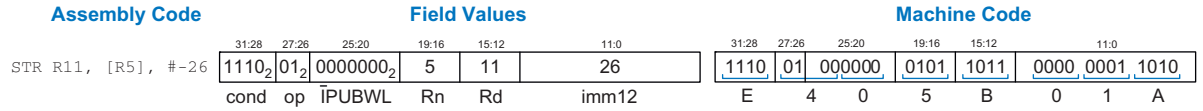


Figure 6.23 Machine code for the memory instruction of Example 6.3

6.4.3 Branch Instructions

Branch instructions use a single 24-bit signed immediate operand, *imm24*, as shown in Figure 6.24. As with data-processing and memory instructions, branch instructions begin with a 4-bit condition field and a 2-bit *op*, which is 10_2. The *funct* field is only 2 bits. The upper bit of *funct* is always 1 for branches. The lower bit, *L*, indicates the type of branch operation: 1 for `BL` and 0 for `B`. The remaining 24-bit two's complement *imm24* field is used to specify an instruction address relative to PC + 8.

Code Example 6.28 shows the use of the branch if less than (`BLT`) instruction and Figure 6.25 shows the machine code for that instruction. The *branch target address* (*BTA*) is the address of the next instruction to execute if the branch is taken. The `BLT` instruction in Figure 6.25 has a BTA of 0x80B4, the instruction address of the `THERE` label.

The 24-bit immediate field gives the number of instructions between the BTA and PC + 8 (two instructions past the branch). In this case, the value in the immediate field (*imm24*) of `BLT` is 3 because the BTA (0x80B4) is three instructions past PC + 8 (0x80A8).

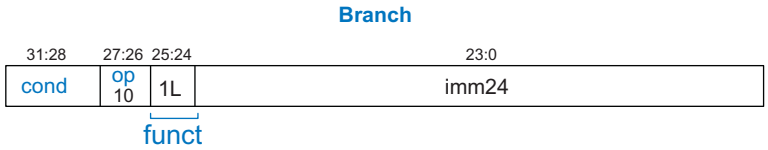


Figure 6.24 Branch instruction format

Code Example 6.28 CALCULATING THE BRANCH TARGET ADDRESS

ARM Assembly Code

```
0x80A0          BLT THERE
0x80A4          ADD R0, R1, R2
0x80A8          SUB R0, R0, R9
0x80AC          ADD SP, SP, #8
0x80B0          MOV PC, LR
0x80B4 THERE    SUB R0, R0, #1
0x80B8          ADD R3, R3, #0x5
```

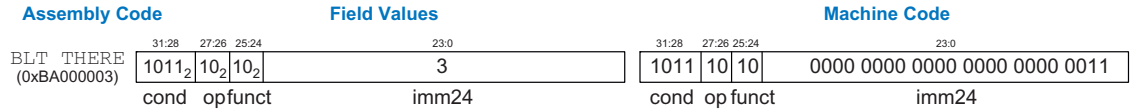


Figure 6.25 **Machine code for branch if less than (BLT)**

The processor calculates the BTA from the instruction by sign-extending the 24-bit immediate, shifting it left by 2 (to convert words to bytes), and adding it to PC + 8.

Example 6.4 CALCULATING THE IMMEDIATE FIELD FOR PC-RELATIVE ADDRESSING

Calculate the immediate field and show the machine code for the branch instruction in the following assembly program.

```
0x8040 TEST   LDRB  R5, [R0, R3]
0x8044        STRB  R5, [R1, R3]
0x8048        ADD   R3, R3, #1
0x8044        MOV   PC, LR
0x8050        BL    TEST
0x8054        LDR   R3, [R1], #4
0x8058        SUB   R4, R3, #9
```

Solution: Figure 6.26 shows the machine code for the branch and link instruction (BL). Its branch target address (0x8040) is six instructions behind PC + 8 (0x8058), so the immediate field is -6.

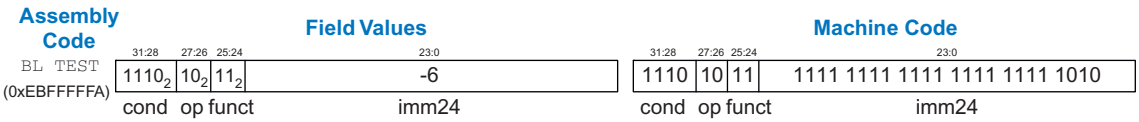


Figure 6.26 BL **machine code**

ARM is unusual among RISC architectures in that it allows the second source operand to be shifted in register and base addressing modes. This requires a shifter in series with the ALU in the hardware implementation but significantly reduces code length in common programs, especially array accesses. For example, in an array of 32-bit data elements, the array index must be left-shifted by 2 to compute the byte offset into the array. Any type of shift is permitted, but left shifts for multiplication are most common.

6.4.4 Addressing Modes

This section summarizes the modes used for addressing instruction operands. ARM uses four main modes: register, immediate, base, and PC-relative addressing. Most other architectures provide similar addressing modes, so understanding these modes helps you easily learn other assembly languages. Register and base addressing have several submodes described below. The first three modes (register, immediate, and base addressing) define modes of reading and writing operands. The last mode (PC-relative addressing) defines a mode of writing the program counter (PC). Table 6.12 summarizes and gives examples of each addressing mode.

Data-processing instructions use register or immediate addressing, in which the first source operand is a register and the second is a register or immediate, respectively. ARM allows the second register to be optionally shifted by an amount specified in an immediate or a third register. Memory instructions use base addressing, in which the base address comes from a register and the offset comes from an immediate, a register, or a register shifted by an immediate. Branches use PC-relative addressing in which the branch target address is computed by adding an offset to PC + 8.

6.4.5 Interpreting Machine Language Code

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	`ADD R3, R2, R1`	R3 ← R2 + R1
Immediate-shifted register	`SUB R4, R5, R9, LSR #2`	R4 ← R5 – (R9 >> 2)
Register-shifted register	`ORR R0, R10, R2, ROR R7`	R0 ← R10 | (R2 ROR R7)
Immediate	`SUB R3, R2, #25`	R3 ← R2 – 25
Base		
Immediate offset	`STR R6, [R11, #77]`	mem[R11+77] ← R6
Register offset	`LDR R12, [R1, –R5]`	R12 ← mem[R1 – R5]
Immediate-shifted register offset	`LDR R8, [R9, R2, LSL #2]`	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	`B LABEL1`	Branch to `LABEL1`

formats start with a 4-bit condition field and a 2-bit *op*. The best place to begin is to look at the *op*. If it is 00_2, then the instruction is a data-processing instruction; if it is 01_2, then the instruction is a memory instruction; if it is 10_2, then it is a branch instruction. Based on that, the rest of the fields can be interpreted.

Example 6.5 TRANSLATING MACHINE LANGUAGE TO ASSEMBLY LANGUAGE

Translate the following machine language code into assembly language.

```
0xE0475001
0xE5949010
```

Solution: First, we represent each instruction in binary and look at bits 27:26 to find the *op* for each instruction, as shown in Figure 6.27. The *op* fields are 00_2 and 01_2, indicating a data-processing and memory instruction, respectively. Next, we look at the *funct* field of each instruction.

The *cmd* field of the data-processing instruction is 2 (0010_2) and the *I*-bit (bit 25) is 0, indicating that it is a `SUB` instruction with a register *Src2*. *Rd* is 5, *Rn* is 7, and *Rm* is 1.

The *funct* field for the memory instruction is 011001_2. $B = 0$ and $L = 1$, so this is an `LDR` instruction. $P = 1$ and $W = 0$, indicating offset addressing. $\overline{I} = 0$, so the offset is an immediate. $U = 1$, so the offset is added. Thus, it is a load register instruction with an immediate offset that is added to the base register. *Rd* is 9, *Rn* is 4, and *imm12* is 16. Figure 6.27 shows the assembly code equivalent of the two machine instructions.

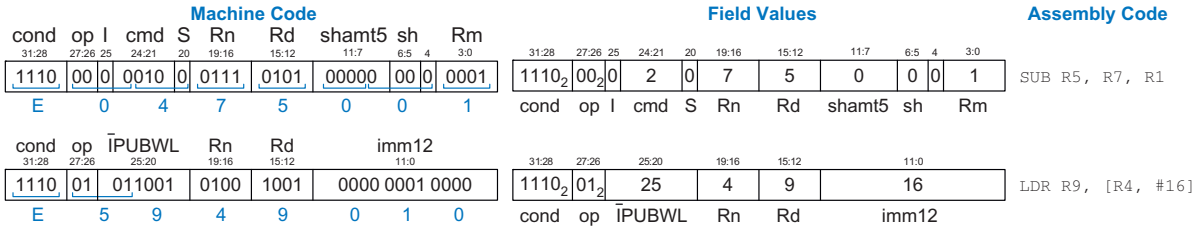


Figure 6.27 Machine code to assembly code translation

6.4.6 The Power of the Stored Program

A program written in machine language is a series of 32-bit numbers representing the instructions. Like other binary numbers, these instructions can be stored in memory. This is called the *stored program* concept, and it is a key reason why computers are so powerful. Running a different

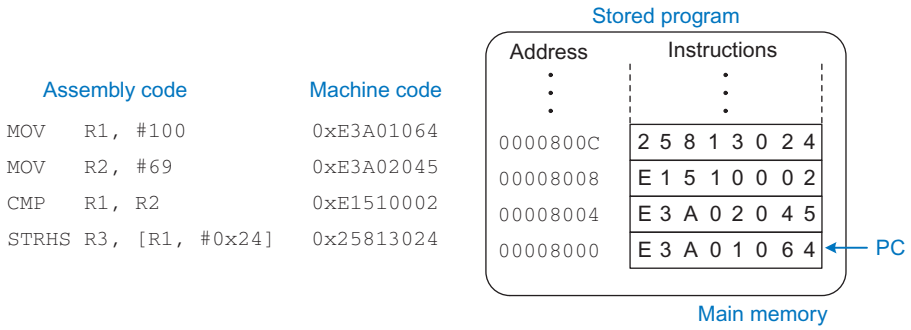


Figure 6.28 Stored program

program does not require large amounts of time and effort to reconfigure or rewire hardware; it only requires writing the new program to memory. In contrast to dedicated hardware, the stored program offers general-purpose computing. In this way, a computer can execute applications ranging from a calculator to a word processor to a video player simply by changing the stored program.

Instructions in a stored program are retrieved, or *fetched*, from memory and executed by the processor. Even large, complex programs are simply a series of memory reads and instruction executions.

Figure 6.28 shows how machine instructions are stored in memory. In ARM programs, the instructions are normally stored starting at low addresses, in this case 0x00008000. Remember that ARM memory is byte-addressable, so 32-bit (4-byte) instruction addresses advance by 4 bytes, not 1.

To run or execute the stored program, the processor fetches the instructions from memory sequentially. The fetched instructions are then decoded and executed by the digital hardware. The address of the current instruction is kept in a 32-bit register called the program counter (PC), which is register R15. For historical reasons, a read to the PC returns the address of the current instruction plus 8.

To execute the code in Figure 6.28, the PC is initialized to address 0x00008000. The processor fetches the instruction at that memory address and executes the instruction, 0xE3A01064 (`MOV R1, #100`). The processor then increments the PC by 4 to 0x00008004, fetches and executes that instruction, and repeats.

The *architectural state* of a microprocessor holds the state of a program. For ARM, the architectural state includes the register file and status registers. If the operating system (OS) saves the architectural state at some point in the program, it can interrupt the program, do something else, and then restore the state such that the program continues properly, unaware that it was ever interrupted. The architectural state is also of great importance when we build a microprocessor in Chapter 7.

Ada Lovelace, 1815–1852. A British mathematician who wrote the first computer program. It calculated the Bernoulli numbers using Charles Babbage's Analytical Engine. She was the daughter of the poet Lord Byron.

6.5 LIGHTS, CAMERA, ACTION: COMPILING, ASSEMBLING, AND LOADING*

Until now, we have shown how to translate short high-level code snippets into assembly and machine code. This section describes how to compile and assemble a complete high-level program and how to load the program into memory for execution. We begin by introducing an example ARM *memory map*, which defines where code, data, and stack memory are located.

Figure 6.29 shows the steps required to translate a program from a high-level language into machine language and to start executing that program. First, a *compiler* translates the high-level code into assembly code. The *assembler* translates the assembly code into machine code and puts it in an object file. The *linker* combines the machine code with code from libraries and other files and determines the proper branch addresses and variable locations to produce an entire executable program. In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the *loader* loads the program into memory and starts execution. The remainder of this section walks through these steps for a simple program.

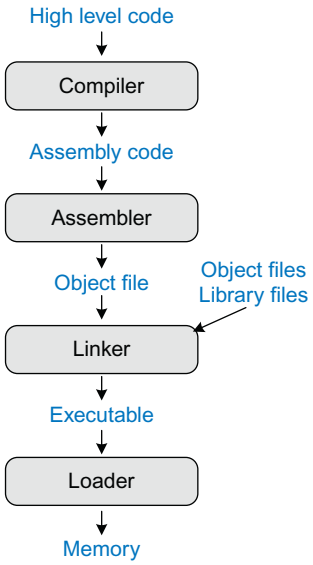


Figure 6.29 Steps for translating and starting a program

6.5.1 The Memory Map

With 32-bit addresses, the ARM address space spans 2^{32} bytes (4 GB). Word addresses are multiples of 4 and range from 0 to 0xFFFFFFFC. Figure 6.30 shows an example memory map. The ARM architecture divides the address space into five parts or segments: the text segment,

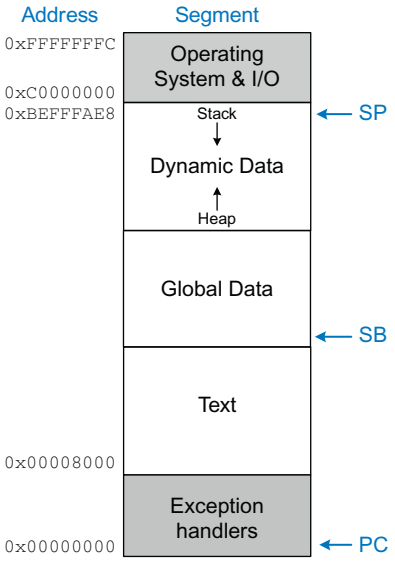


Figure 6.30 Example ARM memory map

We present an example ARM memory map here; however, in ARM, the memory map is somewhat flexible. While the exception vector table must be located at 0x0 and memory-mapped I/O is typically located at the high memory addresses, the user can define where the text (code and constant data), stack, and global data are placed. Moreover, at least historically, most ARM systems have less than 4 GB of memory.

global data segment, dynamic data segment, and segments for exception handlers, the operating system (OS) and input/output (I/O). The following sections describe each segment.

The Text Segment

The *text segment* stores the machine language program. ARM also calls this the *read-only* (RO) *segment*. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The *global data segment* stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Global variables are allocated in memory before the program begins executing. ARM also calls this the *read/write* (RW) *segment*. Global variables are typically accessed using a *static base* register that points to the start of the global segment. ARM conventionally uses R9 as the static base pointer (SB).

The Dynamic Data Segment

The *dynamic data segment* holds the stack and the heap. The data in this segment is not known at start-up but is dynamically allocated and deallocated throughout the execution of the program.

Upon start-up, the operating system sets up the stack pointer (SP) to point to the top of the stack. The stack typically grows downward, as shown here. The stack includes temporary storage and local variables, such as arrays, that do not fit in the registers. As discussed in Section 6.3.7, functions also use the stack to save and restore registers. Each stack frame is accessed in last-in-first-out order.

The *heap* stores data that is allocated by the program during runtime. In C, memory allocations are made by the `malloc` function; in C++ and Java, `new` is used to allocate memory. Like a heap of clothes on a dorm room floor, heap data can be used and discarded in any order. The heap typically grows upward from the bottom of the dynamic data segment.

If the stack and heap ever grow into each other, the program's data can become corrupted. The memory allocator tries to ensure that this never happens by returning an out-of-memory error if there is insufficient space to allocate more dynamic data.

The Exception Handler, OS, and I/O Segments

The lowest part of the ARM memory map is reserved for the exception vector table and exception handlers, starting at address 0x0 (see Section 6.6.3). The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see Section 9.2).

Grace Hopper, 1906–1992. Graduated from Yale University with a Ph.D. in mathematics. Developed the first compiler while working for the Remington Rand Corporation and was instrumental in developing the COBOL programming language. As a naval officer, she received many awards, including a World War II Victory Medal and the National Defense Service Medal.

6.5.2 Compilation

A compiler translates high-level code into assembly language. The examples in this section are based on GCC, a popular and widely used free compiler, running on the Raspberry Pi single-board computer

Code Example 6.29 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables


int sum(int a, int b) {
  return (a + b);
}


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
```

ARM Assembly Code

```
  .text
  .global  sum
  .type    sum, %function
sum:
  add      r0, r0, r1
  bx       lr
  .global  main
  .type    main, %function
main:
  push     {r3, lr}
  mov      r0, #2
  ldr      r3, .L3
  str      r0, [r3, #0]
  mov      r1, #3
  ldr      r3, .L3+4
  str      r1, [r3, #0]
  bl       sum
  ldr      r3, .L3+8
  str      r0, [r3, #0]
  pop      {r3, pc}
.L3:
  .word    f
  .word    g
  .word    y
```

In Code Example 6.29, global variables are accessed using two memory instructions: one to load the *address* of the variable, and a second to read or write the variable. The addresses of the global variables are placed after the code, starting at label `.L3`. `LDR R3, .L3` loads the *address* of `f` into `R3`, and `STR R0, [R3, #0]` writes to `f`; `LDR R3, .L3+4` loads the *address* of `g` into `R3`, and `STR R1, [R3, #0]` writes to `g`, and so on. Section 6.6.1 describes this assembly code construct further.

(see Section 9.3). Code Example 6.29 shows a simple high-level program with three global variables and two functions, along with the assembly code produced by GCC.

To compile, assemble, and link a C program named `prog.c` with GCC, use the command:

```
gcc -O1 -g prog.c -o prog
```

This command produces an executable output file called `prog`. The `-O1` flag asks the compiler to perform basic optimizations rather than producing grossly inefficient code. The `-g` flag tells the compiler to include debugging information in the file.

To see the intermediate steps, we can use GCC's `-S` flag to compile but not assemble or link.

```
gcc -O1 -S prog.c -o prog.s
```

The output, `prog.s`, is rather verbose, but the interesting parts are shown in Code Example 6.29. Note that GCC requires labels to be followed by a colon. The GCC output is in lowercase and has other assembler directives not discussed here. Observe that `sum` returns using the `BX` instruction rather than `MOV PC, LR`. Also, observe that GCC elected to save and restore R3 even though it is not one of the preserved registers. The addresses of the global variables will be stored in a table starting at label .L3.

6.5.3 Assembling

An assembler turns the assembly language code into an *object file* containing machine language code. GCC can create the object file from either `prog.s` or directly from `prog.c` using

```
gcc -c prog.s -o prog.o
```

or

```
gcc -O1 -g -c prog.c -o prog.o
```

The assembler makes two passes through the assembly code. On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names. The names and addresses of the symbols are kept in a symbol table. On the second pass through the code, the assembler produces the machine language code. Addresses for labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

We can *disassemble* the object file using the `objdump` command to see the assembly language code beside the machine language code. If the code was originally compiled with `–g`, the disassembler also shows the corresponding lines of C code:

```
objdump -S prog.o
```

The following shows the disassembly of section .text:

```
00000000 <sum>:
int sum(int a, int b) {
    return (a + b);
}
  0:  e0800001   add  r0, r0, r1
  4:  e12fff1e   bx  lr

00000008 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8:   e92d4008   push  {r3, lr}
       f = 2;
  c:   e3a00002   mov   r0, #2
 10:   e59f301c   ldr   r3, [pc, #28]  ; 34 <main+0x2c>
 14:   e5830000   str   r0, [r3]
       g = 3;
 18:   e3a01003   mov   r1, #3
 1c:   e59f3014   ldr   r3, [pc, #20]  ; 38 <main+0x30>
 20:   e5831000   str   r1, [r3]
       y = sum(f,g);
 24:   ebffffe   bl    0 <sum>
```

Recall from Section 6.4.6 that a read to PC returns the address of the current instruction plus 8. So, `LDR R3, [PC, #28]` loads f's address, which is just after the code at: (PC + 8) + 28 = (0x10 + 0x8) + 0x1C = 0x34.

```
  28:  e59f300c   ldr    r3, [pc, #12]  ; 3c <main+0x34>
  2c:  e5830000   str    r0, [r3]
  return y;
}
  30:  e8bd8008   pop  {r3, pc}
  ...
```

We can also view the symbol table from the object file using `objdump` with the `-t` flag. The interesting parts are shown below. Observe that the `sum` function starts at address 0 and has a size of 8 bytes. `main` starts at address 8 and has size 0x38. The global variable symbols `f`, `g`, and `h` are listed and are 4 bytes each, but they have not yet been assigned addresses.

```
objdump -t prog.o

SYMBOL TABLE:
00000000 l d .text  00000000 .text
00000000 l d .data  00000000 .data
00000000 g F .text  00000008 sum
00000008 g F .text  00000038 main
00000004   O *COM*  00000004 f
00000004   O *COM*  00000004 g
00000004   O *COM*  00000004 y
```

6.5.4 Linking

Most large programs contain more than one file. If the programmer changes only one of the files, it would be wasteful to recompile and reassemble the other files. In particular, programs often call functions in library files; these library files almost never change. If a file of high-level code is not changed, the associated object file need not be updated. Also, a program typically involves some start-up code to initialize the stack, heap, and so forth, that must be executed before calling the `main` function.

The job of the linker is to combine all of the object files and the start-up code into one machine language file called the *executable* and assign addresses for global variables. The linker relocates the data and instructions in the object files so that they are not all on top of each other. It uses the information in the symbol tables to adjust the code based on the new label and global variable addresses. Invoke GCC to link the object file using:

```
gcc prog.o -o prog
```

We can again disassemble the executable using:

```
objdump -S -t prog
```

The start-up code is too lengthy to show, but our program begins at address 0x8390 in the text segment and the global variables are assigned addresses

starting at 0x10570 in the global segment. Notice the `.word` assembler directives defining the addresses of the global variables `f`, `g`, and `y`.

```
00008390 <sum>:

int sum(int a, int b) {
  return (a + b);
}
  8390:  e0800001   add  r0, r0, r1
  8394:  e12fff1e   bx  lr

00008398 <main>:

int f, g, y; // global variables

int sum(int a, int b);

int main(void) {
  8398:  e92d4008   push  {r3, lr}
  f = 2;
  839c:  e3a00002   mov  r0, #2
  83a0:  e59f301c   ldr  r3, [pc, #28]  ; 83c4 <main+0x2c>
  83a4:  e5830000   str  r0, [r3]
  g = 3;
  83a8:  e3a01003   mov  r1, #3
  83ac:  e59f3014   ldr  r3, [pc, #20]  ; 83c8 <main+0x30>
  83b0:  e5831000   str  r1, [r3]
  y = sum(f,g);
  83b4:  ebfffff5   bl  8390 <sum>
  83b8:  e59f300c   ldr  r3, [pc, #12]  ; 83cc <main+0x34>
  83bc:  e5830000   str  r0, [r3]
  return y;
}
  83c0:  e8bd8008   pop    {r3, pc}
  83c4:  00010570   .word  0x00010570
  83c8:  00010574   .word  0x00010574
  83cc:  00010578   .word  0x00010578
```

The instruction `LDR R3, [PC, #28]` in the executable loads from address (PC + 8) + 28 = (0x83A0 + 0x8) + 0x1C = 0x83C4. This memory address contains the value 0x10570, the location of global variable `f`.

The executable also contains an updated symbol table with the relocated addresses of the functions and global variables.

```
SYMBOL TABLE:
000082e4 l  d  .text  00000000    .text
00010564 l  d  .data  00000000    .data
00008390 g    F .text  00000008    sum
00008398 g    F .text  00000038    main
00010570 g    O .bss   00000004    f
00010574 g    O .bss   00000004    g
00010578 g    O .bss   00000004    y
```

6.5.5 Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. The operating system jumps to the beginning of the program to begin executing. Figure 6.31 shows the memory map at the beginning of program execution.

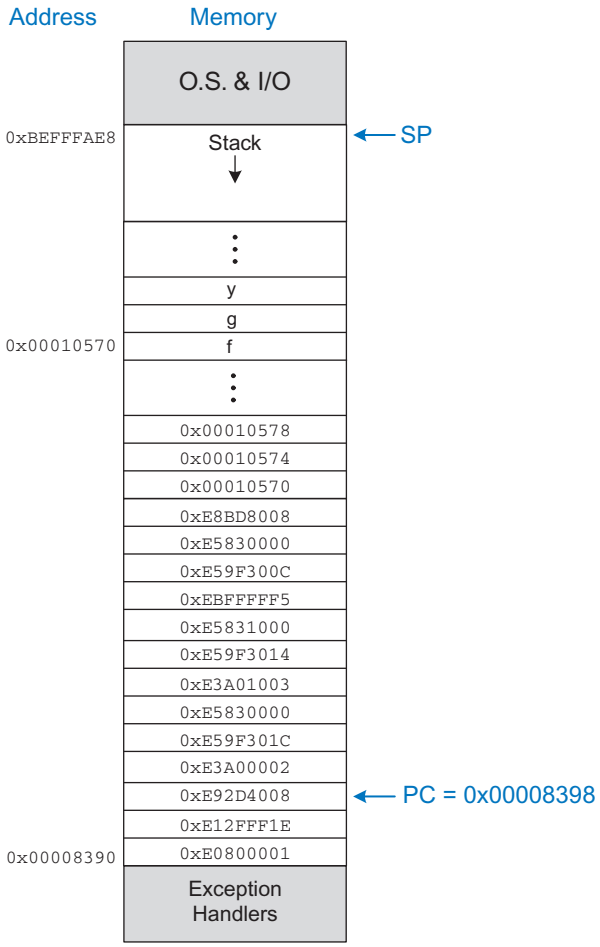


Figure 6.31 Executable loaded in memory

6.6 ODDS AND ENDS*

This section covers a few optional topics that do not fit naturally elsewhere in the chapter. These topics include loading 32-bit literals, NOPs, and exceptions.

6.6.1 Loading Literals

Many programs need to load 32-bit literals, such as constants or addresses. `MOV` only accepts a 12-bit source, so the `LDR` instruction is used to load these numbers from a *literal pool* in the text segment. ARM assemblers accept loads of the form

```
LDR Rd, =literal
LDR Rd, =label
```

The first loads a 32-bit constant specified by `literal`, and the second loads the address of a variable or pointer in the program specified by `label`. In both cases, the value to load is kept in a *literal pool*, which is a portion of the text segment containing literals. The literal pool must be less than 4096 bytes from the `LDR` instruction so that the load can be performed as `LDR Rd, [PC, #offset_to_literal]`. The program must be careful to branch around the literal pool because executing literals would be nonsensical or worse.

Code Example 6.30 illustrates loading a literal. As shown in Figure 6.32, suppose the `LDR` instruction is at address 0x8110 and the literal is at 0x815C. Remember that reading the PC returns the address 8 bytes beyond the current instruction being executed. Hence, when the `LDR` is executed, reading the PC returns 0x8118. Thus, the `LDR` uses an offset of 0x44 to find the literal pool: `LDR R1, [PC, #0x44]`.

Code Example 6.30 LARGE IMMEDIATE USING A LITERAL POOL

High-level code

```
int a = 0x2B9056F;
```

ARM Assembly Code

```
; R1 = a
  LDR R1, =0x2B9056F
  ...
```

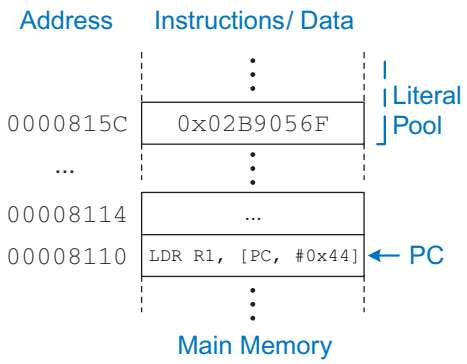


Figure 6.32 Example literal pool

Pseudoinstructions are not actually part of the instruction set but are shorthand for instructions or instruction sequences that are commonly used by programmers and compilers. The assembler translates pseudoinstructions into one or more actual instructions.

6.6.2 NOP

NOP is a mnemonic for "no operation" and is pronounced "no op." It is a *pseudoinstruction* that does nothing. The assembler translates it to `MOV R0, R0` (0xE1A00000). NOPs are useful to, among other things, achieve some delay or align instructions.

6.6.3 Exceptions

An *exception* is like an unscheduled function call that branches to a new address. Exceptions may be caused by hardware or software. For example, the processor may receive notification that the user pressed a key on a keyboard. The processor may stop what it is doing, determine which key was pressed, save it for future reference, and then resume the program that was running. Such a hardware exception triggered by an input/output (I/O) device such as a keyboard is often called an *interrupt*. Alternatively, the program may encounter an error condition such as an undefined instruction. The program then branches to code in the operating system (OS), which may choose to either emulate the unimplemented instruction or terminate the offending program. Software exceptions are sometimes called *traps*. A particularly important form of a trap is a *system call*, whereby the program invokes a function in the OS running at a higher privilege level. Other causes of exceptions include reset and attempts to read nonexistent memory.

Like any other function call, an exception must save the return address, jump to some address, do its work, clean up after itself, and return to the program where it left off. Exceptions use a *vector* table to determine where to jump to the *exception handler* and use *banked registers* to maintain extra copies of key registers so that they will not corrupt the registers in the active program. Exceptions also change the *privilege level* of the program, allowing the exception handler to access protected parts of memory.

Execution Modes and Privilege Levels

An ARM processor can operate in one of several execution modes with different privilege levels. The different modes allow an exception to take place in an exception handler without corrupting state; for example, an interrupt could occur while the processor is executing operating system code in Supervisor mode, and a subsequent Abort exception could occur if the interrupt attempted to access an invalid memory address. The exception handlers would eventually return and resume the supervisor code. The mode is specified in the bottom bits of the Current Program Status Register (CPSR), as was shown in Figure 6.6. Table 6.13 lists execution modes and their encodings. User mode operates at privilege level PL0, which is unable to access protected portions of memory such as the operating system code. The other modes operate at privilege level PL1, which can access all system resources. Privilege levels are important so that buggy or malicious user code cannot corrupt other programs or crash or infect the system.

Exception Vector Table

When an exception occurs, the processor branches to an offset in the *exception vector table*, depending on the cause of the exception. Table 6.14 describes the vector table, which is normally located starting at address 0x00000000 in memory. For example, when an interrupt occurs, the processor branches to address 0x00000018. Similarly, on

Table 6.13 ARM execution modes

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001

Table 6.14 Exception vector table

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

ARM also supports a High Vectors mode in which the exception vector table starts at address 0xFFFF0000. For example, the system may boot using a vector table in ROM at address 0x00000000. Once the system starts up, the OS may write an updated vector table in RAM at 0xFFFF0000 and put the system into High Vectors mode.

power-up, the processor branches to address 0x00000000. Each exception vector offset typically contains a branch instruction to an exception handler, code that handles the exception and then either exits or returns to the user code.

Banked Registers

Before an exception changes the PC, it must save the return address in the LR so that the exception handler knows where to return. However, it must take care not to disturb the value already in the LR, which the program will need later. Therefore, the processor maintains a bank of different registers to use as LR during each of the execution modes. Similarly, the exception handler must not disturb the status register bits.

Hence, a bank of *saved program status registers* (SPSRs) is used to hold a copy of the CPSR during exceptions.

If an exception takes place while a program is manipulating its stack frame, the frame might be in an unstable state (e.g., data has been written onto the stack but the stack pointer is not yet pointing to the top of stack). Hence, each execution mode also uses its own stack and banked copy of SP pointing to the top of its stack. Memory must be reserved for each execution mode's stack and banked versions of the stack pointers must be initialized at start-up.

The first thing that an exception handler must do is to push all of the registers it might change onto the stack. This takes some time. ARM has a *fast interrupt* execution mode FIQ in which R8–R12 are also banked. Thus, the exception handler can immediately begin without saving these registers.

Exception Handling

Now that we have defined execution modes, exception vectors, and banked registers, we can define what occurs during an exception. Upon detecting an exception, the processor:

1. Stores the CPSR into the banked SPSR
2. Sets the execution mode and privilege level based on the type of exception
3. Sets *interrupt mask* bits in the CPSR so that the exception handler will not be interrupted
4. Stores the return address into the banked LR
5. Branches to the exception vector table based on the type of exception

The processor then executes the instruction in the exception vector table, typically a branch to the exception handler. The handler usually pushes other registers onto its stack, takes care of the exception, and pops the registers back off the stack. The exception handler returns using the `MOVS PC, LR` instruction, a special flavor of `MOV` that performs the following cleanup:

1. Copies the banked SPSR to the CPSR to restore the status register
2. Copies the banked LR (possibly adjusted for certain exceptions) to the PC to return to the program where the exception occurred
3. Restores the execution mode and privilege level

Exception-Related Instructions

Programs operate at a low privilege level, whereas the operating system has a higher privilege level. To transition between levels in a controlled way, the program places arguments in registers and issues a *supervisor call* (`SVC`) instruction, which generates an exception and raises the

privilege level. The OS examines the arguments and performs the requested function, and then returns to the program.

The OS and other code operating at PL1 can access the banked registers for the various execution modes using the `MRS` (move to register from special register) and `MSR` (move to special register from register) instructions. For example, at boot time, the OS will use these instructions to initialize the stacks for exception handlers.

Start-up

On start-up, the processor jumps to the reset vector and begins executing *boot loader* code in supervisor mode. The boot loader typically configures the memory system, initializes the stack pointer, and reads the OS from disk; then it begins a much longer boot process in the OS. The OS eventually will load a program, change to unprivileged user mode, and jump to the start of the program.

6.7 EVOLUTION OF ARM ARCHITECTURE

The ARM1 processor was first developed by Acorn Computer in Britain for the BBC Micro computers in 1985 as an upgrade to the 6502 microprocessor used in many personal computers of the era. It was followed within the year by the ARM2, which went into production in the Acorn Archimedes computer. ARM was an acronym for *Acorn RISC Machine*. The product implemented Version 2 of the ARM instruction set (ARMv2). The address bus was only 26 bits, and the upper 6 bits of the 32-bit PC were used to hold status bits. The architecture included almost all of the instructions described in this chapter, including data-processing, most loads and stores, branches, and multiplies.

As of ARMv7, the CPSR is called the Application Program Status Register (APSR).

ARM soon extended the address bus to a full 32 bits, moving the status bits into a dedicated Current Program Status Register (CPSR). ARMv4, introduced in 1993, added halfword loads and stores and provided both signed and unsigned halfword and byte loads. This is the core of the modern ARM instruction set, and is what we have covered in this chapter.

The ARM instruction set has seen many enhancements described in subsequent sections. The highly successful ARM7TDMI processor in 1995 introduced the 16-bit Thumb instruction set in ARMv4T to improve code density. ARMv5TE added digital signal processing (DSP) and optional floating-point instructions. ARMv6 added multimedia instructions and enhanced the Thumb instruction set. ARMv7 improved the floating-point and multimedia instructions, renaming them Advanced SIMD. ARMv8 introduced a completely new 64-bit architecture. Various other system programming instructions have been introduced as the architecture has evolved.

6.7.1 Thumb Instruction Set

Thumb instructions are 16 bits long to achieve higher code density; they are identical to regular ARM instructions but generally have limitations, including that they:

- Access only the bottom eight registers
- Reuse a register as both a source and destination
- Support shorter immediates
- Lack conditional execution
- Always write the status flags

Almost all ARM instructions have Thumb equivalents. Because the instructions are less powerful, more are required to write an equivalent program. However, the instructions are half as long, giving overall Thumb code size of about 65% of the ARM equivalent. The Thumb instruction set is valuable not only to reduce the size and cost of code storage memory, but also to allow for an inexpensive 16-bit bus to instruction memory and to reduce the power consumed by fetching instructions from the memory.

ARM processors have an instruction set state register, ISETSTATE, that includes a T bit to indicate whether the processor is in normal mode (T = 0) or Thumb mode (T = 1). This mode determines how instructions should be fetched and interpreted. The `BX` and `BLX` branch instructions toggle the T bit to enter or exit Thumb mode.

Thumb instruction encoding is more complex and irregular than ARM instructions to pack as much useful information as possible into 16-bit halfwords. Figure 6.33 shows encodings for common Thumb instructions. The upper bits specify the type of instruction. Data-processing instructions typically specify two registers, one of which is both the first source and the destination. They always write the status flags. Adds, subtracts, and shifts can specify a short immediate. Conditional branches specify a 4-bit condition code and a short offset, whereas unconditional branches allow a longer offset. Note that `BX` takes a 4-bit register identifier so that it can access the link register LR. Special forms of `LDR`, `STR`, `ADD`, and `SUB` are defined to operate relative to the stack pointer SP (to access the stack frame during function calls). Another special form of `LDR` loads relative to the PC (to access a literal pool). Forms of `ADD` and `MOV` can access all 16 registers. `BL` always requires two halfwords to specify a 22-bit destination.

ARM subsequently refined the Thumb instruction set and added a number of 32-bit Thumb-2 instructions to boost performance of common operations and to allow any program to be written in Thumb mode.

The irregular Thumb instruction set encoding and variable-length instructions (1 or 2 halfwords) are characteristic of 16-bit processor architectures that must pack a large amount of information into a short instruction word. The irregularity complicates instruction decoding.

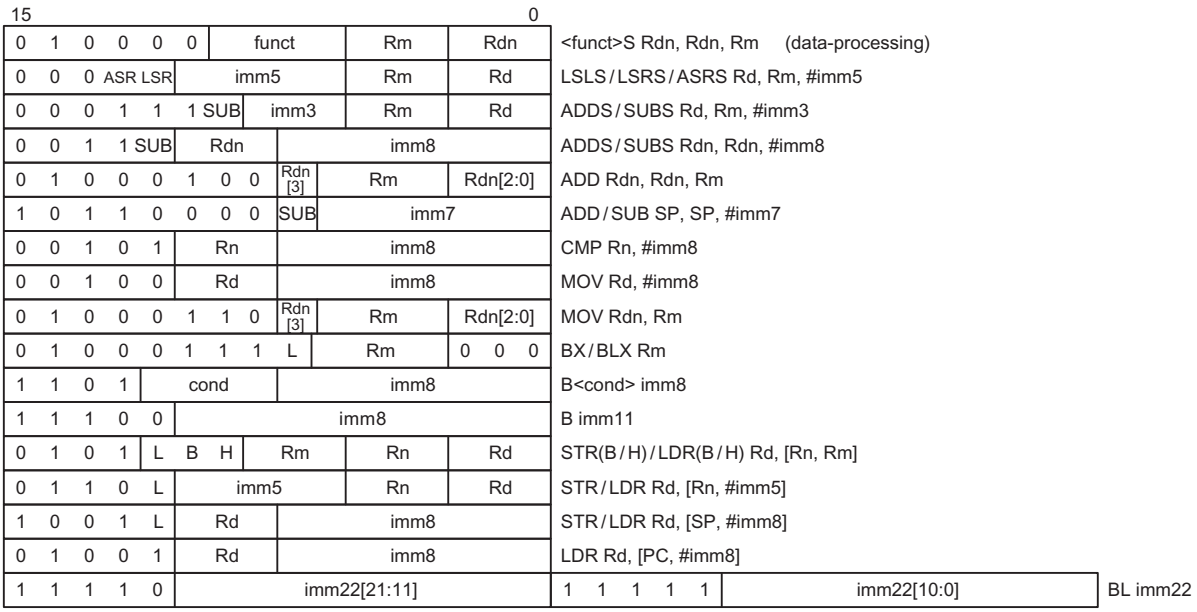


Figure 6.33 Thumb instruction encoding examples

Thumb-2 instructions are identified by their most significant 5 bits being 11101, 11110, or 11111. The processor then fetches a second halfword containing the remainder of the instruction. The Cortex-M series of processors operates exclusively in Thumb state.

6.7.2 DSP Instructions

The Fast Fourier Transform (FFT), the most common DSP algorithm, is both complicated and performance-critical. The DSP instructions in computer architectures are intended to perform efficient FFTs, especially on 16-bit fractional data.

The basic multiply instructions, listed in Appendix B, are part of ARMv4. ARMv5TE added the saturating math instructions and packed and fractional multiplies to support DSP algorithms.

Digital signal processors (DSPs) are designed to efficiently handle signal processing algorithms such as the Fast Fourier Transform (FFT) and Finite/Infinite Impulse Response filters (FIR/IIR). Common applications include audio and video encoding and decoding, motor control, and speech recognition. ARM provides a number of DSP instructions for these purposes. DSP instructions include multiply, add, and multiply-accumulate (MAC)—multiply and add the result to a running sum: sum = sum + src1 × src2. MAC is a distinguishing feature separating DSP instruction sets from regular instruction sets. It is very commonly used in DSP algorithms and doubles the performance relative to separate multiply and add instructions. However, MAC requires specifying an extra register to hold the running sum.

DSP instructions often operate on short (16-bit) data representing samples read from a sensor by an analog-to-digital converter. However, the intermediate results are held to greater precision (e.g., 32 or 64 bits)

Table 6.15 DSP data types

Type	Sign Bit	Integer Bits	Fractional Bits
short	1	15	0
unsigned short	0	16	0
long	1	31	0
unsigned long	0	32	0
long long	1	63	0
unsigned long long	0	64	0
Q15	1	0	15
Q31	1	0	31

or saturated to prevent overflow. In *saturated arithmetic*, results larger than the most positive number are treated as the most positive, and results smaller than the most negative are treated as the most negative. For example, in 32-bit arithmetic, results greater than $2^{31} - 1$ saturate at $2^{31} - 1$, and results less than -2^{31} saturate at -2^{31}. Common DSP data types are given in Table 6.15. Two's complement numbers are indicated as having one sign bit. The 16-, 32-, and 64-bit types are also known as *half*, *single*, and *double* precision, not to be confused with single and double-precision floating-point numbers. For efficiency, two half-precision numbers are packed in a single 32-bit word.

The *integer* types come in signed and unsigned flavors with the sign bit in the msb. *Fractional* types (Q15 and Q31) represent a signed fractional number; for example, Q31 spans the range $[-1, 1-2^{-31}]$ with a step of 2^{-31} between consecutive numbers. These types are not defined in the C standard but are supported by some libraries. Q31 can be converted to Q15 by truncation or rounding. In truncation, the Q15 result is just the upper half. In rounding, 0x00008000 is added to the Q31 value and then the result is truncated. When a computation involves many steps, rounding is useful because it avoids accumulating multiple small truncation errors into a significant error.

ARM added a *Q* flag to the status registers to indicate that overflow or saturation has occurred in DSP instructions. For applications where accuracy is critical, the program can clear the *Q* flag before a computation, do the computation in single-precision, and check the *Q* flag afterward. If it is set, overflow occurred and the computation can be repeated in double precision if necessary.

Saturated arithmetic is an important way to gracefully degrade accuracy in DSP algorithms. Commonly, single-precision arithmetic is sufficient to handle most inputs, but pathological cases can overflow the single-precision range. An overflow causes an abrupt sign change to a radically wrong answer, which may appear to the user as a click in an audio stream or a strangely colored pixel in a video stream. Going to double-precision arithmetic prevents overflow but degrades performance and increases power consumption in the typical case. Saturated arithmetic clips the overflow at the maximum or minimum value, which is usually close to the desired value and causes little inaccuracy.

Addition and subtraction are performed identically no matter which format is used. However, multiplication depends on the type. For example, with 16-bit numbers, the number 0xFFFF is interpreted as 65535 for unsigned short, −1 for short, and -2^{-15} for Q15 numbers. Hence, 0xFFFF × 0xFFFF has a very different value for each representation (4,294,836,225; 1; and 2^{-30}, respectively). This leads to different instructions for signed and unsigned multiplication.

A Q15 number A can be viewed as $a \times 2^{-15}$, where a is its interpretation in the range $[-2^{15}, 2^{15}-1]$ as a signed 16-bit number. Hence, the product of two Q15 numbers is:

$$A \times B = a \times b \times 2^{-30} = 2 \times a \times b \times 2^{-31}$$

This means that to multiply two Q15 numbers and get a Q31 result, do ordinary signed multiplication and then double the product. The product can then be truncated or rounded to put it back into Q15 format if necessary.

The rich assortment of multiply and multiply-accumulate instructions are summarized in Table 6.16. MACs require up to four registers: *RdHi*, *RdLo*, *Rn*, and *Rm*. For double-precision operations, *RdHi* and *RdLo* hold the most and least significant 32 bits, respectively. For example, `UMLAL RdLo, RdHi, Rn, Rm` computes {*RdHi*, *RdLo*} = {*RdHi*, *RdLo*} + *Rn* × *Rm*. Half-precision multiplies come in various flavors denoted in braces to choose the operands from the top or bottom half of the word, and in *dual* forms where both the top and bottom halves are multiplied. MACs involving half-precision inputs and a single-precision accumulator (`SMLA*`, `SMLAW*`, `SMUAD`, `SMUSD`, `SMLAD`, `SMLSD`) will set the *Q* flag if the accumulator overflows. The most significant word (MSW) multiplies also come in forms with an `R` suffix that round rather than truncate.

The DSP instructions also include saturated add (`QADD`) and subtract (`QSUB`) of 32-bit words that saturate the results instead of overflowing. They also include `QDADD` and `QDSUB`, which double the second operand before adding/subtracting it to/from the first with saturation; we will shortly find these valuable in fractional MACs. They set the *Q* flag if saturation occurs.

Finally, the DSP instructions include `LDRD` and `STRD` that load and store an even/odd pair of registers in a 64-bit memory double word. These instructions increase the efficiency of moving double-precision values between memory and registers.

Table 6.17 summarizes how to use the DSP instructions to multiply or MAC various types of data. The examples assume halfword data is in the bottom half of a register and that the top half is zero; use the T flavor of `SMUL` when the data is in the top instead. The result is stored in R2, or in {R3, R2} for double-precision. Fractional operations (Q15/Q31) double the result using saturated adds to prevent overflow when multiplying −1 × −1.

Table 6.16 Multiply and multiply-accumulate instructions

Instruction	Function	Description
Ordinary 32-bit multiplication works for both signed and unsigned		
MUL	32 = 32 × 32	Multiply
MLA	32 = 32 + 32 × 32	Multiply-accumulate
MLS	32 = 32 − 32 × 32	Multiply-subtract
unsigned long long = unsigned long × unsigned long		
UMULL	64 = 32 × 32	Unsigned multiply long
UMLAL	64 = 64 + 32 × 32	Unsigned multiply-accumulate long
UMAAL	64 = 32 + 32 × 32 + 32	Unsigned multiply-accumulate-add long
long long = long × long		
SMULL	64 = 32 × 32	Signed multiply long
SMLAL	64 = 64 + 32 × 32	Signed multiply-accumulate long
Packed arithmetic: short × short		
SMUL{BB/BT/TB/TT}	32 = 16 × 16	Signed multiply {bottom/top}
SMLA{BB/BT/TB/TT}	32 = 32 + 16 × 16	Signed multiply-accumulate {bottom/top}
SMLAL{BB/BT/TB/TT}	64 = 64 + 16 × 16	Signed multiply-accumulate long {bottom/top}
Fractional multiplication (Q31 / Q15)		
SMULW{B/T}	32 = (32 × 16) >> 16	Signed multiply word-halfword {bottom/top}
SMLAW{B/T}	32 = 32 + (32 × 16) >> 16	Signed multiply-add word-halfword {bottom/top}
SMMUL{R}	32 = (32 × 32) >> 32	Signed MSW multiply {round}
SMMLA{R}	32 = 32 + (32 × 32) >> 32	Signed MSW multiply-accumulate {round}
SMMLS{R}	32 = 32 − (32 × 32) >> 32	Signed MSW multiply-subtract {round}
long or long long = short × short + short × short		
SMUAD	32 = 16 × 16 + 16 × 16	Signed dual multiply-add
SMUSD	32 = 16 × 16 − 16 × 16	Signed dual multiply-subtract
SMLAD	32 = 32 + 16 × 16 + 16 × 16	Signed multiply-accumulate dual
SMLSD	32 = 32 + 16 × 16 − 16 × 16	Signed multiply-subtract dual
SMLALD	64 = 64 + 16 × 16 + 16 × 16	Signed multiply-accumulate long dual
SMLSLD	64 = 64 + 16 × 16 − 16 × 16	Signed multiply-subtract long dual

Table 6.17 Multiply and MAC code for various data types

First Operand (R0)	Second Operand (R1)	Product (R3/R2)	Multiply	MAC

short	short	short	`SMULBB R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`SMLABB R2, R0, R1`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

short	short	long	`SMULBB R2, R0, R1`	`SMLABB R2, R0, R1, R2`

short	short	long long	`MOV R2, #0`
			`MOV R3, #0`
			`SMLALBB R2, R3, R0, R1`
				`SMLALBB R2, R3, R0, R1`

long	short	long	`SMULWB R2, R0, R1`	`SMLAWB R2, R0, R1, R2`
long	long	long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
long	long	long long	`SMULL R2, R3, R0, R1`	`SMLAL R2, R3, R0, R1`

unsigned short	unsigned short	unsigned short	`MUL R2, R0, R1`
			`LDR R3, =0x0000FFFF`
			`AND R2, R3, R2`
				`MLA R2, R0, R1, R2`
				`LDR R3, =0x0000FFFF`
				`AND R2, R3, R2`

unsigned short	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned short	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long	`MUL R2, R0, R1`	`MLA R2, R0, R1, R2`
unsigned long	unsigned long	unsigned long long	`UMULL R2, R3, R0, R1`	`UMLAL R2, R3, R0, R1`

Q15	Q15	Q15	`SMULBB R2, R0, R1`
			`QADD R2, R2, R2`
			`LSR R2, R2, #16`
				`SMLABB R2, R0, R1, R2`
				`SSAT R2, 16, R2`

Q15	Q15	Q31	`SMULBB R2, R0, R1` `QADD R2, R2, R2`	`SMULBB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q15	Q31	`SMULWB R2, R0, R1` `QADD R2, R2, R2`	`SMULWB R3, R0, R1` `QDADD R2, R2, R3`
Q31	Q31	Q31	`SMMUL R2, R0, R1` `QADD R2, R2, R2`	`SMMUL R3, R0, R1` `QDADD R2, R2, R3`

6.7.3 Floating-Point Instructions

Floating-point is more flexible than the fixed-point numbers favored in DSP and makes programming easier. Floating-point is widely used in graphics, scientific applications, and control algorithms. Floating-point arithmetic can be performed with a series of ordinary data-processing instructions but is faster and consumes less power using dedicated floating-point instructions and hardware.

The ARMv5 instruction set includes optional floating-point instructions. These instructions access at least 16 64-bit double-precision registers separate from the ordinary registers. These registers can also be treated as pairs of 32-bit single-precision registers. The registers are named D0–D15 as double-precision or S0–S31 as single-precision. For example, `VADD.F32 S2, S0, S1` and `VADD.F64 D2, D0, D1` perform single and double-precision floating-point adds, respectively. Floating-point instructions, listed in Table 6.18, are suffixed with `.F32` or `.F64` to indicate single- or double-precision floating-point.

Table 6.18 ARM floating-point instructions

Instruction	Function
`VABS Rd, Rm`	Rd = |Rm|
`VADD Rd, Rn, Rm`	Rd = Rn + Rm
`VCMP Rd, Rm`	Compare and set floating-point status flags
`VCVT Rd, Rm`	Convert between int and float
`VDIV Rd, Rn, Rm`	Rd = Rn / Rm
`VMLA Rd, Rn, Rm`	Rd = Rd + Rn * Rm
`VMLS Rd, Rn, Rm`	Rd = Rd − Rn * Rm
`VMOV Rd, Rm or #const`	Rd = Rm or constant
`VMUL Rd, Rn, Rm`	Rd = Rn * Rm
`VNEG Rd, Rm`	Rd = −Rm
`VNMLA Rd, Rn, Rm`	Rd = −(Rd + Rn * Rm)
`VNMLS Rd, Rn, Rm`	Rd = −(Rd − Rn * Rm)
`VNMUL Rd, Rn, Rm`	Rd = −Rn * Rm
`VSQRT Rd, Rm`	Rd = sqrt(Rm)
`VSUB Rd, Rn, Rm`	Rd = Rn − Rm

The MRC and MCR instructions are used to transfer data between the ordinary registers and the floating-point coprocessor registers.

ARM defines the Floating-Point Status and Control Register (FPSCR). Like the ordinary status register, it holds *N*, *Z*, *C*, and *V* flags for floating-point operations. It also specifies rounding modes, exceptions, and special conditions such as overflow, underflow, and divide-by-zero. The VMRS and VMSR instructions transfer information between a regular register and the FPSCR.

6.7.4 Power-Saving and Security Instructions

Battery-powered devices save power by spending most of their time in sleep mode. ARMv6K introduced instructions to support such power savings. The wait for interrupt (WFI) instruction allows the processor to enter a low-power state until an interrupt occurs. The system may generate interrupts based on user events (such as touching a screen) or on a periodic timer. The wait for event (WFE) instruction is similar but is helpful in multiprocessor systems (see Section 7.7.8) so that a processor can go to sleep until notified by another processor. It wakes up either during an interrupt or when another processor sends an event using the SEV instruction.

ARMv7 enhances the exception handling to support virtualization and security. In *virtualization*, multiple operating systems can run concurrently on the same processor, unaware of each other's existence. A hypervisor switches between the operating systems. The hypervisor operates at privilege level PL2. It is invoked with a hypervisor trap exception. With *security* extensions, the processor defines a secure state with limited means of entry and restricted access to secure portions of memory. Even if an attacker compromises the operating system, the secure kernel may resist tampering. For example, the secure kernel may be used to disable a stolen phone or to enforce digital rights management such that a user can't duplicate copyrighted content.

6.7.5 SIMD Instructions

The term SIMD (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. Moreover,

when the components from 16 adjacent pixels are packed into a 128-bit quadword, the processing can be performed 16 times faster. Similarly, coordinates in a 3-dimensional graphics space are generally represented with 32-bit (single-precision) floating-point numbers. Four of these coordinates can be packed into a 128-bit quadword.

Most modern architectures offer SIMD arithmetic operations with wide SIMD registers packing multiple narrower operands. For example, the ARMv7 Advanced SIMD instructions share the registers from the floating-point unit. Moreover, these registers can also be paired to act as eight 128-bit quad words Q0–Q7. The registers pack together several 8-, 16-, 32-, or 64-bit integer or floating-point values. The instructions are suffixed with `.I8`, `.I16`, `.I32`, `.I64`, `.F32`, or `.F64` to indicate how the registers should be treated.

Figure 6.34 shows the `VADD.I8 D2, D1, D0` vector add instruction operating on eight pairs of 8-bit integers packed into 64-bit double words. Similarly `VADD.I32 Q2, Q1, Q0` adds four pairs of 32-bit integers packed into 128-bit quad words and `VADD.F32, D2, D1, D0` adds two pairs of 32-bit single-precision floating-point numbers packed into 64-bit double words. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ must not affect the result of $a_1 + b_1$.

Advanced SIMD instructions begin with `V`. They include the following categories:

- Basic arithmetic functions also defined for floating-point
- Loads and stores of multiple elements, including deinterleaving and interleaving
- Bitwise logical operations
- Comparisons
- Many flavors of shifts, additions, and subtractions with and without saturation
- Many flavors of multiply and MAC
- Miscellaneous instructions

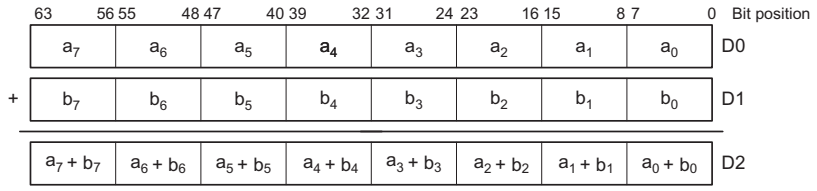


Figure 6.34 Packed arithmetic: eight simultaneous 8-bit additions

ARMv6 also defined a more limited set of SIMD instructions operating on the regular 32-bit registers. These include 8- and 16-bit addition and subtraction, and instructions to efficiently pack and unpack bytes and halfwords into a word. These instructions are useful to manipulate 16-bit data in DSP code.

6.7.6 64-bit Architecture

32-bit architectures allow a program to directly access at most 2^{32} bytes = 4 GB of memory. Large computer servers led the transition to 64-bit architectures that can access vast amounts of memory. Personal computers and then mobile devices followed. 64-bit architectures can sometimes be faster as well because they move more information with a single instruction.

Many architectures simply extend their general-purpose registers from 32 to 64 bits, but ARMv8 introduced a new instruction set as well to streamline idiosyncrasies. The classic instruction set lacks enough general-purpose registers for complex programs, forcing costly movement of data between registers and memory. Keeping the PC in R15 and SP in R13 also complicates the processor implementation, and programs often need a register containing the value 0.

The ARMv8 instructions are still 32 bits long and the instruction set looks very much like ARMv7, but with some problems cleaned up. In ARMv8, the register file is expanded to 31 64-bit registers (called X0–X30) and the PC and SP are no longer part of the general-purpose registers. X30 serves as the link register. Note that there is no X31 register; instead, it is called the zero register (ZR) and is hardwired to 0. Data-processing instructions can operate on 32- or 64-bit values, whereas loads and stores always use 64-bit addresses. To make room for the extra bits to specify source and destination registers, the condition field is removed from most instructions. However, branches can still be conditional. ARMv8 also streamlines exception handling, doubles the number of advanced SIMD registers, and adds instructions for AES and SHA cryptography. The instruction encodings are rather complex and do not classify into a handful of categories.

On reset, ARMv8 processors boot in 64-bit mode. The processor can drop into 32-bit mode by setting a bit in a system register and invoking an exception. It returns to 64-bit mode when the exception returns.

6.8 ANOTHER PERSPECTIVE: x86 ARCHITECTURE

Almost all personal computers today use x86 architecture microprocessors. x86, also called IA-32, is a 32-bit architecture originally developed by Intel. AMD also sells x86 compatible microprocessors.

The x86 architecture has a long and convoluted history dating back to 1978, when Intel announced the 16-bit 8086 microprocessor. IBM selected the 8086 and its cousin, the 8088, for IBM's first personal computers. In 1985, Intel introduced the 32-bit 80386 microprocessor, which was backward compatible with the 8086, so it could run software developed for earlier PCs. Processor architectures compatible with the 80386 are called x86 processors. The Pentium, Core, and Athlon processors are well known x86 processors.

Various groups at Intel and AMD over many years have shoehorned more instructions and capabilities into the antiquated architecture. The result is far less elegant than ARM. However, software compatibility is far more important than technical elegance, so x86 has been the *de facto* PC standard for more than two decades. More than 100 million x86 processors are sold every year. This huge market justifies more than $5 billion of research and development annually to continue improving the processors.

x86 is an example of a Complex Instruction Set Computer (CISC) architecture. In contrast to RISC architectures such as ARM, each CISC instruction can do more work. Programs for CISC architectures usually require fewer instructions. The instruction encodings were selected to be more compact, so as to save memory, when RAM was far more expensive than it is today; instructions are of variable length and are often less than 32 bits. The trade-off is that complicated instructions are more difficult to decode and tend to execute more slowly.

This section introduces the x86 architecture. The goal is not to make you into an x86 assembly language programmer, but rather to illustrate some of the similarities and differences between x86 and ARM. We think it is interesting to see how x86 works. However, none of the material in this section is needed to understand the rest of the book. Major differences between x86 and ARM are summarized in Table 6.19.

Table 6.19 Major differences between ARM and x86

Feature	ARM	x86
# of registers	15 general purpose	8, some restrictions on purpose
# of operands	3–4 (2–3 sources, 1 destination)	2 (1 source, 1 source/destination)
operand location	registers or immediates	registers, immediates, or memory
operand size	32 bits	8, 16, or 32 bits
condition flags	yes	yes
instruction types	simple	simple and complicated
instruction encoding	fixed, 4 bytes	variable, 1–15 bytes

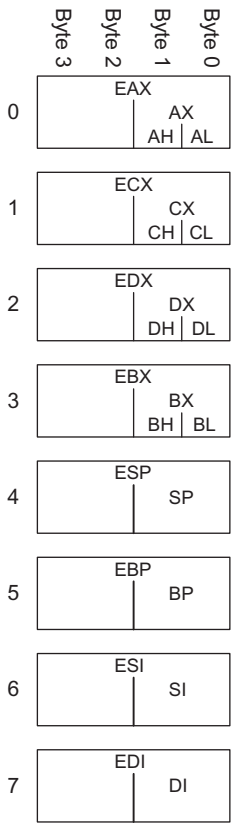


Figure 6.35 x86 registers

6.8.1 x86 Registers

The 8086 microprocessor provided eight 16-bit registers. It could separately access the upper and lower eight bits of some of these registers. When the 32-bit 80386 was introduced, the registers were extended to 32 bits. These registers are called EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. For backward compatibility, the bottom 16 bits and some of the bottom 8-bit portions are also usable, as shown in Figure 6.35.

The eight registers are almost, but not quite, general purpose. Certain instructions cannot use certain registers. Other instructions always put their results in certain registers. Like SP in ARM, ESP is normally reserved for the stack pointer.

The x86 program counter is called the EIP (the *extended instruction pointer*). Like the ARM PC, it advances from one instruction to the next or can be changed with branch and function call instructions.

6.8.2 x86 Operands

ARM instructions always act on registers or immediates. Explicit load and store instructions are needed to move data between memory and the registers. In contrast, x86 instructions may operate on registers, immediates, or memory. This partially compensates for the small set of registers.

ARM instructions generally specify three operands: two sources and one destination. x86 instructions specify only two operands. The first is a source. The second is both a source and the destination. Hence, x86 instructions always overwrite one of their sources with the result. Table 6.20 lists the combinations of operand locations in x86. All combinations are possible except memory to memory.

Table 6.20 Operand locations

Source/ Destination	Source	Example	Meaning
register	register	`add EAX, EBX`	`EAX <– EAX + EBX`
register	immediate	`add EAX, 42`	`EAX <– EAX + 42`
register	memory	`add EAX, [20]`	`EAX <– EAX + Mem[20]`
memory	register	`add [20], EAX`	`Mem[20] <– Mem[20] + EAX`
memory	immediate	`add [20], 42`	`Mem[20] <– Mem[20] + 42`

Table 6.21 Memory addressing modes

Example	Meaning	Comment
`add EAX, [20]`	`EAX <– EAX + Mem[20]`	displacement
`add EAX, [ESP]`	`EAX <– EAX + Mem[ESP]`	base addressing
`add EAX, [EDX+40]`	`EAX <– EAX + Mem[EDX+40]`	base + displacement
`add EAX, [60+EDI*4]`	`EAX <– EAX + Mem[60+EDI*4]`	displacement + scaled index
`add EAX, [EDX+80+EDI*2]`	`EAX <– EAX + Mem[EDX+80+EDI*2]`	base + displacement + scaled index

Table 6.22 Instructions acting on 8-, 16-, or 32-bit data

Example	Meaning	Data Size
`add AH, BL`	`AH <– AH + BL`	8-bit
`add AX, –1`	`AX <– AX + 0xFFFF`	16-bit
`add EAX, EDX`	`EAX <– EAX + EDX`	32-bit

Like ARM, x86 has a 32-bit memory space that is byte-addressable. However, x86 supports a wider variety of memory indexing modes. Memory locations are specified with any combination of a *base register*, *displacement*, and a *scaled index register*. Table 6.21 illustrates these combinations. The displacement can be an 8-, 16-, or 32-bit value. The scale multiplying the index register can be 1, 2, 4, or 8. The base + displacement mode is equivalent to the ARM base addressing mode for loads and stores. Like ARM, x86 also provides a scaled index. In x86, the scaled index provides an easy way to access arrays or structures of 2-, 4-, or 8-byte elements without having to issue a sequence of instructions to generate the address.

While ARM always acts on 32-bit words, x86 instructions can operate on 8-, 16-, or 32-bit data. Table 6.22 illustrates these variations.

6.8.3 Status Flags

x86, like many CISC architectures, uses condition flags (also called *status flags*) to make decisions about branches and to keep track of carries and arithmetic overflow. x86 uses a 32-bit register, called EFLAGS, that stores the status flags. Some of the bits of the EFLAGS register are given in Table 6.23. Other bits are used by the operating system.

ARM's use of condition flags sets it apart from other RISC architectures.

Table 6.23 Selected EFLAGS

Name	Meaning
CF (Carry Flag)	Carry out generated by last arithmetic operation. Indicates overflow in unsigned arithmetic. Also used for propagating the carry between words in multiple-precision arithmetic
ZF (Zero Flag)	Result of last operation was zero
SF (Sign Flag)	Result of last operation was negative (msb = 1)
OF (Overflow Flag)	Overflow of two's complement arithmetic

The architectural state of an x86 processor includes EFLAGS as well as the eight registers and the EIP.

6.8.4 x86 Instructions

x86 has a larger set of instructions than ARM. Table 6.24 describes some of the general purpose instructions. x86 also has instructions for floating-point arithmetic and for arithmetic on multiple short data elements packed into a longer word. D indicates the destination (a register or memory location), and S indicates the source (a register, memory location, or immediate).

Note that some instructions always act on specific registers. For example, 32×32-bit multiplication always takes one of the sources from EAX and always puts the 64-bit result in EDX and EAX. LOOP always stores the loop counter in ECX. PUSH, POP, CALL, and RET use the stack pointer, ESP.

Conditional jumps check the flags and branch if the appropriate condition is met. They come in many flavors. For example, JZ jumps if the zero flag (*ZF*) is 1. JNZ jumps if the zero flag is 0. Like ARM, the jumps usually follow an instruction, such as the compare instruction (CMP), that sets the flags. Table 6.25 lists some of the conditional jumps and how they depend on the flags set by a prior compare operation.

6.8.5 x86 Instruction Encoding

The x86 instruction encodings are truly messy, a legacy of decades of piecemeal changes. Unlike ARMv4, whose instructions are uniformly 32 bits, x86 instructions vary from 1 to 15 bytes, as shown in Figure 6.36.[1]

[1] It is possible to construct 17-byte instructions if all the optional fields are used. However, x86 places a 15-byte limit on the length of legal instructions.

Table 6.24 Selected x86 instructions

Instruction	Meaning	Function
ADD/SUB	add/subtract	D = D + S / D = D − S
ADDC	add with carry	D = D + S + CF
INC/DEC	increment/decrement	D = D + 1 / D = D − 1
CMP	compare	Set flags based on D − S
NEG	negate	D = − D
AND/OR/XOR	logical AND/OR/XOR	D = D op S
NOT	logical NOT	$D = \overline{D}$
IMUL/MUL	signed/unsigned multiply	EDX:EAX = EAX × D
IDIV/DIV	signed/unsigned divide	EDX:EAX/D EAX = Quotient; EDX = Remainder
SAR/SHR	arithmetic/logical shift right	D = D >>> S / D = D >> S
SAL/SHL	left shift	D = D << S
ROR/ROL	rotate right/left	Rotate D by S
RCR/RCL	rotate right/left with carry	Rotate CF and D by S
BT	bit test	CF = D[S] (the S*th* bit of D)
BTR/BTS	bit test and reset/set	CF = D[S]; D[S] = 0 / 1
TEST	set flags based on masked bits	Set flags based on D AND S
MOV	move	D = S
PUSH	push onto stack	ESP = ESP −4; Mem[ESP] = S
POP	pop off stack	D = MEM[ESP]; ESP = ESP + 4
CLC, STC	clear/set carry flag	CF = 0 / 1
JMP	unconditional jump	relative jump: EIP = EIP + S absolute jump: EIP = S
Jcc	conditional jump	if (flag) EIP = EIP + S
LOOP	loop	ECX = ECX −1 if (ECX ≠ 0) EIP = EIP + imm
CALL	function call	ESP = ESP −4; MEM[ESP] = EIP; EIP = S
RET	function return	EIP = MEM[ESP]; ESP = ESP + 4

Table 6.25 Selected branch conditions

Instruction	Meaning	Function after `CMP D, S`
`JZ/JE`	jump if `ZF = 1`	jump if `D = S`
`JNZ/JNE`	jump if `ZF = 0`	jump if `D ≠ S`
`JGE`	jump if `SF = OF`	jump if `D ≥ S`
`JG`	jump if `SF = OF` and `ZF = 0`	jump if `D >S`
`JLE`	jump if `SF ≠ OF` or `ZF = 1`	jump if `D ≤ S`
`JL`	jump if `SF ≠ OF`	jump if `D <S`
`JC/JB`	jump if `CF = 1`	
`JNC`	jump if `CF = 0`	
`JO`	jump if `OF = 1`	
`JNO`	jump if `OF = 0`	
`JS`	jump if `SF = 1`	
`JNS`	jump if `SF = 0`	

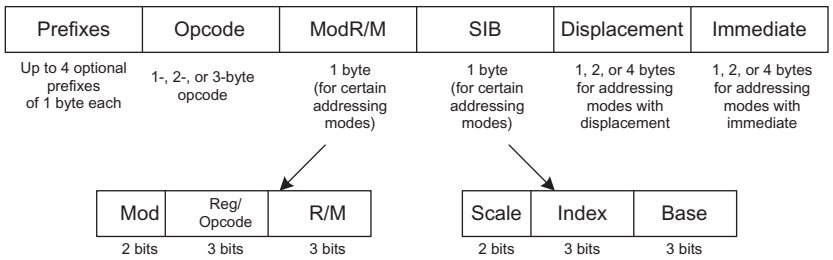


Figure 6.36 x86 instruction encodings

The *opcode* may be 1, 2, or 3 bytes. It is followed by four optional fields: *ModR/M*, *SIB*, *Displacement*, and *Immediate*. *ModR/M* specifies an addressing mode. *SIB* specifies the scale, index, and base registers in certain addressing modes. *Displacement* indicates a 1-, 2-, or 4-byte displacement in certain addressing modes. And *Immediate* is a 1-, 2-, or 4-byte constant for instructions using an immediate as the source operand. Moreover, an instruction can be preceded by up to four optional byte-long prefixes that modify its behavior.

The *ModR/M* byte uses the 2-bit *Mod* and 3-bit *R/M* field to specify the addressing mode for one of the operands. The operand can come from

one of the eight registers, or from one of 24 memory addressing modes. Due to artifacts in the encodings, the ESP and EBP registers are not available for use as the base or index register in certain addressing modes. The *Reg* field specifies the register used as the other operand. For certain instructions that do not require a second operand, the *Reg* field is used to specify three more bits of the *opcode*.

In addressing modes using a scaled index register, the *SIB* byte specifies the index register and the scale (1, 2, 4, or 8). If both a base and index are used, the *SIB* byte also specifies the base register.

ARM fully specifies the instruction in the *cond*, *op*, and *funct* fields of the instruction. x86 uses a variable number of bits to specify different instructions. It uses fewer bits to specify more common instructions, decreasing the average length of the instructions. Some instructions even have multiple opcodes. For example, `add AL, imm8` performs an 8-bit add of an immediate to AL. It is represented with the 1-byte opcode, 0x04, followed by a 1-byte immediate. The A register (AL, AX, or EAX) is called the *accumulator*. On the other hand, `add D, imm8` performs an 8-bit add of an immediate to an arbitrary destination, *D* (memory or a register). It is represented with the 1-byte *opcode* 0x80 followed by one or more bytes specifying *D*, followed by a 1-byte immediate. Many instructions have shortened encodings when the destination is the accumulator.

In the original 8086, the *opcode* specified whether the instruction acted on 8- or 16-bit operands. When the 80386 introduced 32-bit operands, no new opcodes were available to specify the 32-bit form. Instead, the same opcode was used for both 16- and 32-bit forms. An additional bit in the *code segment descriptor* used by the OS specifies which form the processor should choose. The bit is set to 0 for backward compatibility with 8086 programs, defaulting the *opcode* to 16-bit operands. It is set to 1 for programs to default to 32-bit operands. Moreover, the programmer can specify prefixes to change the form for a particular instruction. If the *prefix* 0x66 appears before the *opcode*, the alternative size operand is used (16 bits in 32-bit mode, or 32 bits in 16-bit mode).

6.8.6 Other x86 Peculiarities

The 80286 introduced *segmentation* to divide memory into segments of up to 64 KB in length. When the OS enables segmentation, addresses are computed relative to the beginning of the segment. The processor checks for addresses that go beyond the end of the segment and indicates an error, thus preventing programs from accessing memory outside their own segment. Segmentation proved to be a hassle for programmers and is not used in modern versions of the Windows operating system.

Intel and Hewlett-Packard jointly developed a new 64-bit architecture called IA-64 in the mid 1990's. It was designed from a clean slate, bypassing the convoluted history of x86, taking advantage of 20 years of new research in computer architecture, and providing a 64-bit address space. However, IA-64 has yet to become a market success. Most computers needing the large address space now use the 64-bit extensions of x86.

x86 contains string instructions that act on entire strings of bytes or words. The operations include moving, comparing, or scanning for a specific value. In modern processors, these instructions are usually slower than performing the equivalent operation with a series of simpler instructions, so they are best avoided.

As mentioned earlier, the 0x66 *prefix* is used to choose between 16- and 32-bit operand sizes. Other prefixes include ones used to lock the bus (to control access to shared variables in a multiprocessor system), to predict whether a branch will be taken or not, and to repeat the instruction during a string move.

The bane of any architecture is to run out of memory capacity. With 32-bit addresses, x86 can access 4 GB of memory. This was far more than the largest computers had in 1985, but by the early 2000's it had become limiting. In 2003, AMD extended the address space and register sizes to 64 bits, calling the enhanced architecture AMD64. AMD64 has a compatibility mode that allows it to run 32-bit programs unmodified while the OS takes advantage of the bigger address space. In 2004, Intel gave in and adopted the 64-bit extensions, renaming them Extended Memory 64 Technology (EM64T). With 64-bit addresses, computers can access 16 exabytes (16 billion GB) of memory.

For those curious about more details of the x86 architecture, the x86 Intel Architecture Software Developer's Manual is freely available on Intel's Web site.

6.8.7 The Big Picture

ARM strikes a balance between simple instructions and dense code by including features such as condition flags and shifted register operands. Thease features make ARM code more compact than other RISC architectures.

This section has given a taste of some of the differences between the ARM RISC architecture and the x86 CISC architecture. x86 tends to have shorter programs, because a complex instruction is equivalent to a series of simple ARM instructions and because the instructions are encoded to minimize memory use. However, the x86 architecture is a hodgepodge of features accumulated over the years, some of which are no longer useful but must be kept for compatibility with old programs. It has too few registers, and the instructions are difficult to decode. Merely explaining the instruction set is difficult. Despite all these failings, x86 is firmly entrenched as the dominant computer architecture for PCs, because the value of software compatibility is so great and because the huge market justifies the effort required to build fast x86 microprocessors.

6.9 SUMMARY

To command a computer, you must speak its language. A computer architecture defines how to command a processor. Many different computer architectures are in widespread commercial use today, but once

you understand one, learning others is much easier. The key questions to ask when approaching a new architecture are:

- What is the data word length?
- What are the registers?
- How is memory organized?
- What are the instructions?

ARM is a 32-bit architecture because it operates on 32-bit data. The ARM architecture has 16 registers which include 15 general-purpose registers and the PC. In principle, any of the general-purpose registers can be used in any code. However, by convention, certain registers are reserved for certain purposes for ease of programming and so that functions written by different programmers can communicate easily. For example, R14 (the link register LR) holds the return address after a `BL` instruction, and R0–R3 hold the arguments of a function. ARM has a byte-addressable memory system with 32-bit addresses. Instructions are 32 bits long and are word-aligned for efficient access. This chapter discussed the most commonly used ARM instructions.

The power of defining a computer architecture is that a program written for any given architecture can run on many different implementations of that architecture. For example, programs written for the Intel Pentium processor in 1993 will generally still run (and run much faster) on the Intel Xeon or AMD Phenom processors in 2015.

In the first part of this book, we learned about the circuit and logic levels of abstraction. In this chapter, we jumped up to the architecture level. In the next chapter, we study microarchitecture, the arrangement of digital building blocks that implement a processor architecture. Microarchitecture is the link between hardware and software engineering. And, we believe it is one of the most exciting topics in all of engineering: You will learn to build your own microprocessor!