2

Instructions: Language of the Computer

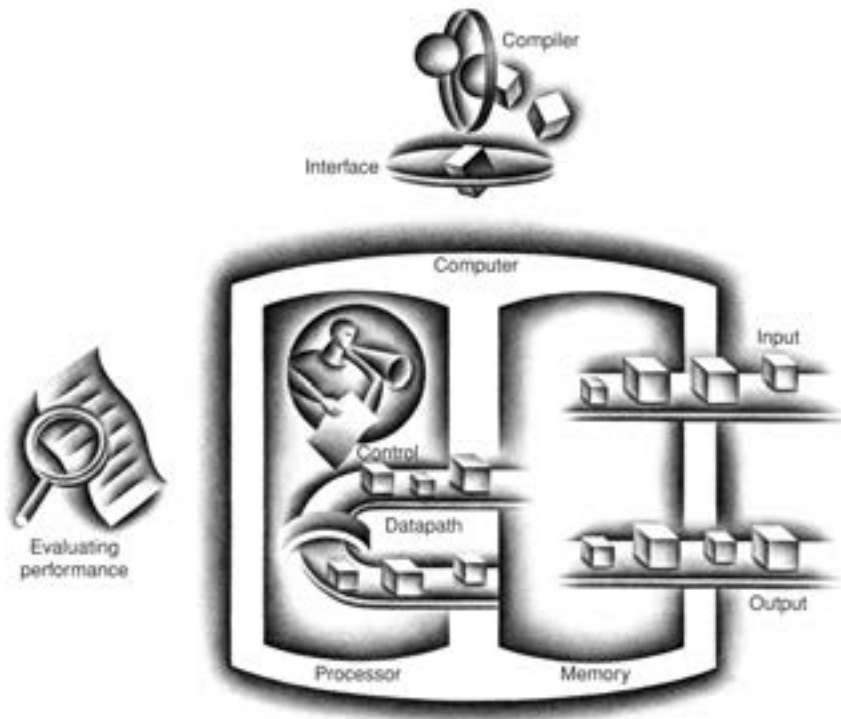*I speak Spanish to God, Italian to women, French to men, and German to my horse.*

Charles V, King of France
1337–1380

The Five Classic Components of a Computer

2.1   Introduction

To command a computer's hardware, you must speak its language. The words of a computer's language are called *instructions*, and its vocabulary is called an **instruction set**. In this chapter, you will see the instruction set of a real computer, both in the form written by people and in the form read by the computer. We introduce instructions in a top-down fashion. Starting from a notation that looks like a restricted programming language, we refine it step-by-step until you see the real language of a real computer. Chapter 3 continues our downward descent, unveiling the hardware for arithmetic and the representation of floating-point numbers.

instruction set The vocabulary of commands understood by a given architecture.

You might think that the languages of computers would be as diverse as those of people, but in reality computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others. This similarity occurs because all computers are constructed from hardware technologies based on similar underlying principles and because there are a few basic operations that all computers must provide. Moreover, computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and power. This goal is time honored; the following quote was written before you could buy a computer, and it is as true today as it was in 1947:

> It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations. . . . The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.
>
> Burks, Goldstine, and von Neumann, 1947

The "simplicity of the equipment" is as valuable a consideration for today's computers as it was for those of the 1950s. The goal of this chapter is to teach an instruction set that follows this advice, showing both how it is represented in hardware and the relationship between high-level programming languages and this more primitive one. Our examples are in the C programming language; ◉ **Section 2.15** on the CD shows how these would change for an object-oriented language like Java.

By learning how to represent instructions, you will also discover the secret of computing: the **stored-program concept**. Moreover, you will exercise your "foreign language" skills by writing programs in the language of the computer and running them on the simulator that comes with this book. You will also see the impact of programming languages and compiler optimization on performance. We conclude with a look at the historical evolution of instruction sets and an overview of other computer dialects.

The chosen instruction set comes is ARM, which is the most popular 32-bit instruction set in the world, as 4 billion were shipped in 2008. Later, we will take a quick look at two other popular instruction sets. MIPS is quite similar to ARM, with making up in elegance what it lacks in popularity. The other example, the Intel x86, is inside almost all of the 300 million PCs made in 2008.

We reveal the ARM instruction set a piece at a time, giving the rationale along with the computer structures. This top-down, step-by-step tutorial weaves the components with their explanations, making the computer's language more palatable. Figure 2.1 gives a sneak preview of the instruction set covered in this chapter.

stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored-program computer.

2.2 Operations of the Computer Hardware

Every computer must be able to perform arithmetic. The ARM assembly language notation

```
ADD a, b, c
```

instructs a computer to add the two variables b and c and to put their sum in a.

This notation is rigid in that each ARM arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables b, c, d, and e into variable a. (In this section we are being deliberately vague about what a "variable" is; in the next section we'll explain in detail.)

The following sequence of instructions adds the four variables:

```
ADD a, b, c      ; The sum of b and c is placed in a.
ADD a, a, d      ; The sum of b, c, and d is now in a.
ADD a, a, e      ; The sum of b, c, d, and e is now in a.
```

Thus, it takes three instructions to sum the four variables.

The words to the right of the sharp symbol (;) on each line above are *comments* for the human reader; the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

There must certainly be instructions for performing the fundamental arithmetic operations.

Burks, Goldstine, and von Neumann, 1947

ARM operands

Name	Example	Comments
16 registers	`r0, r1, r2,...,r11, r12, sp, lr, pc`	Fast locations for data. In ARM, data must be in registers to perform arithmetic, register
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. ARM uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

ARM assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`ADD  r1,r2,r3`	r1 = r2 − r3	3 register operands
	subtract	`SUB  r1,r2,r3`	r1 = r2 + r3	3 register operands
Data transfer	load register	`LDR   r1, [r2,#20]`	r1 = Memory[r2 + 20]	Word from memory to register
	store register	`STR   r1, [r2,#20]`	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	`LDRH  r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	`LDRHS  r1, [r2,#20]`	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	`STRH  r1, [r2,#20]`	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	`LDRB  r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	`LDRBS  r1, [r2,#20]`	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	`STRB  r1, [r2,#20]`	Memory[r2 + 20] = r1	Byte from register to memory
	swap	`SWP  r1, [r2,#20]`	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	`MOV  r1, r2`	r1 = r2	Copy value into register
Logical	and	`AND  r1, r2, r3`	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	`ORR  r1, r2, r3`	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	`MVN  r1, r2`	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	`LSL  r1, r2, #10`	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	`LSR  r1, r2, #10`	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	`CMP r1, r2`	cond. flag = r1 − r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	`BEQ  25`	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	`B  2500`	go to PC + 8 + 10000	Branch
	branch and link	`BL  2500`	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

FIGURE 2.1   ARM assembly language revealed in this chapter. This information is also found in Column 1 of the ARM Reference Data Card at the front of this book.

The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of four underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

Compiling Two C Assignment Statements into ARM

This segment of a C program contains the five variables a, b, c, d, and e. Since Java evolved from C, this example and the next few work for either high-level programming language:

EXAMPLE

```
a = b + c;
d = a - e;
```

The translation from C to ARM assembly language instructions is performed by the *compiler*. Show the ARM code produced by a compiler.

An ARM instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two ARM assembly language instructions:

ANSWER

```
ADD a, b, c
SUB d, a, e
```

Compiling a Complex C Assignment into ARM

A somewhat complex statement contains the five variables f, g, h, i, and j:

EXAMPLE

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

What might a C compiler produce?

The compiler must break this statement into several assembly instructions, since only one operation is performed per ARM instruction. The first ARM instruction calculates the sum of g and h. We must place the result somewhere, so the compiler creates a temporary variable, called t0:

ANSWER

```
ADD t0,g,h ; temporary variable t0 contains g + h
```

Although the next operation is subtract, we need to calculate the sum of i and j before we can subtract. Thus, the second instruction places the sum of i and j in another temporary variable created by the compiler, called t1:

```
ADD t1,i,j  ; temporary variable t1 contains i + j
```

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable f, completing the compiled code:

```
SUB f,t0,t1 ; f gets t0 - t1, which is (g + h) - (i + j)
```

Check Yourself

For a given function, which programming language likely takes the most lines of code? Put the three representations below in order.

1. Java

2. C

3. ARM assembly language

Elaboration: To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called *Java bytecodes* (see ⊙ Section 2.15 on the CD), which is quite different from the ARM instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like ARM. Because this compilation is normally done much later than for C programs, such Java compilers are often called *Just In Time* (JIT) compilers. Section 2.12 shows how JITs are used later than C compilers in the start-up process, and Section 2.13 shows the performance consequences of compiling versus interpreting Java programs.

2.3   Operands of the Computer Hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called *registers*. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the ARM architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name **word** in the ARM architecture.

word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the ARM architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 16 to 32 on current computers. (See ⊙ Section 2.20 on the CD for the history of the number of registers.) Thus, continuing in our top-down, stepwise evolution of the symbolic representation of the ARM language, in this section we have added the restriction that the three

operands of ARM arithmetic instructions must each be chosen from one of the 16 32-bit registers.

The reason for the limit of 16 registers may be found in the second of our four underlying design principles of hardware technology:

Design Principle 2: Smaller is faster.

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as "smaller is faster" are not absolutes; 15 registers may not be faster than 16. Yet, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 16 is the number of bits it would take in the instruction format, as Section 2.5 demonstrates "Energy is a major concern today, so a third reason for using fewer registers is to conserve energy."

Chapter 4 shows the central role that registers play in hardware construction; as we shall see in this chapter, effective use of registers is critical to program performance. We use the convention $r0, r1, \ldots, r15$ to refer to registers $0, 1, \ldots, r15$.

Compiling a C Assignment Using Registers

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

EXAMPLE

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

The variables f, g, h, i, and j are assigned to the registers r0, r1, r2, r3, and r4, respectively. What is the compiled ARM code?

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, r5 and r6, which correspond to the temporary variables above:

ANSWER

```
ADD r5,r0,r1 ; register r5 contains g + h
ADD r6,r2,r3 ; register r6 contains i + j
SUB r4,r5,r6 ; r4 gets r5 - r6, which is (g + h)-(i + j)
```

Memory Operands

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These complex data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures?
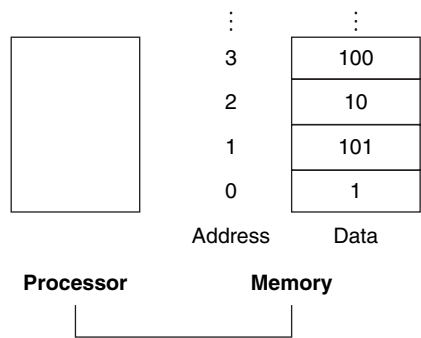
FIGURE 2.2   Memory addresses and contents of memory at those locations. If these elements were words, these addresses would be incorrect, since ARM actually uses byte addressing, with each word representing four bytes. Figure 2.3 shows the memory addressing for sequential word addresses.

Recall the five components of a computer introduced in Chapter 1 and repeated on page 75. The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in ARM instructions; thus, ARM must include instructions that transfer data between memory and registers. Such instructions are called **data transfer instructions**. To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in Figure 2.2, the address of the third data element is 2, and the value of Memory[2] is 10.

The data transfer instruction that copies data from memory to a register is traditionally called *load*. The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The actual ARM name for this instruction is LDR, standing for *load word into register*.

data transfer instruction A command that moves data between memory and registers.

address A value used to delineate the location of a specific data element within a memory array.

Compiling an Assignment When an Operand Is in Memory

EXAMPLE

Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers r1 and r2 and uses r5 as a temporary register as before. Let's also assume that the starting address, or *base address*, of the array is in r3. Compile this C assignment statement:

```
g = h + A[8];
```

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer A[8] to a register. The address of this array element is the sum of the base of the array A, found in register r3, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
LDR  r5,[r3,#8] ; Temporary reg r5 gets A[8]
```

(On the next page we'll make a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in r5 (which equals A[8]) since it is in a register. The instruction must add h (contained in r2) to A[8] (r5) and put the sum in the register corresponding to g (associated with r1):

```
ADD  r1,r2,r5 ; g = h + A[8]
```

The constant in a data transfer instruction (8) is called the *offset,* and the register added to form the address (r3) is called the *base register.*

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Hardware/ Software Interface

Since 8-bit *bytes* are useful in many programs, most architectures address individual bytes. Therefore, the address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4. For example, Figure 2.3 shows the actual ARM addresses for the words in Figure 2.2; the byte address of the third word is 8.
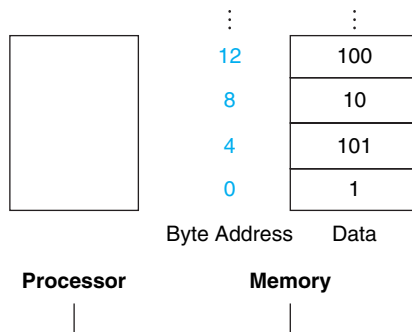

FIGURE 2.3 Actual ARM memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since ARM addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

alignment restriction
A requirement that data be aligned in memory on natural boundaries.

In ARM, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**, and many architectures have it. (Chapter 4 suggests why alignment leads to faster data transfers.)

Computers divide into those that use the address of the leftmost or "big end" byte as the word address versus those that use the rightmost or "little end" byte. ARM is in the *little-endian* camp.

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register* r3 *must be 4 × 8, or 32,* so that the load address will select A[8] and not A[8/4]. (See the related pitfall on page 171 of Section 2.18.)

The instruction complementary to load is traditionally called *store;* it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then offset to select the array element, and finally the base register. Once again, the ARM address is specified in part by a constant and in part by the contents of a register. The actual ARM name is STR, standing for *store word from a register.*

Compiling Using Load and Store

EXAMPLE

Assume variable h is associated with register r2 and the base address of the array A is in r3. What is the ARM assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more ARM instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select A[8], and the ADD instruction places the sum in r5:

```
LDR    r5,[r3,#32]  ; Temporary reg r5 gets A[8]
ADD    r5,r2,r5     ; Temporary reg r5 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4 × 12) as the offset and register r3 as the base register.

```
STR    r5,[r3,#48]  ; Stores h + A[8] back into A[12]
```

Load word and store word are the instructions that copy words between memory and registers in the ARM architecture. Other brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in Section 2.17.

Many programs have more variables than computers have registers. Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory. The process of putting less commonly used variables (or those needed later) into memory is called *spilling* registers.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This is indeed the case; data accesses are faster if data is in registers instead of memory.

Moreover, data is more useful when in a register. An ARM arithmetic instruction can read two registers, operate on them, and write the result. An ARM data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both faster to access and simpler to use. Accessing registers also uses less energy than accessing memory. To achieve highest performance and conserve energy, compilers must use registers efficiently.

Hardware/ Software Interface

Constant or Immediate Operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the ARM arithmetic instructions have a constant as an operand when running the SPEC2006 benchmarks.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register r3, we could use the code

```
    LDR r5, [r1,#AddrConstant4]    ; r5 = constant 4
    ADD r3,r3,r5                   ; r3 = r3 + r5 (r5 == 4)
```

assuming that r1 + AddrConstant4 is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer the option having one operand of the arithmetic instructions be a constant, called an immediate operand. To add 4 to register `r3`, we just write

```
ADD    r3,r3,#4              ; r3 = r3 + 4
```

The sharp or hash symbol (#) means the following number is a constant.

Immediate operands illustrate the third hardware design principle, first mentioned in the Fallacies and Pitfalls of Chapter 1:

Design Principle 3: Make the common case fast.

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

Check Yourself

Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?

1. Very fast: They increase as fast as Moore's law, which predicts doubling the number of transistors on a chip every 18 months.

2. Very slow: Since programs are usually distributed in the language of the computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable.

Elaboration: The ARM offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in Section 2.13.

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the *index register*. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

2.4 Signed and Unsigned Numbers

First, let's quickly review how a computer represents numbers. Humans are taught to think in base 10, but numbers may be represented in any base. For example, 123 base 10 = 1111011 base 2.

Numbers are kept in computer hardware as a series of high and low electronic signals, and so they are considered base 2 numbers. (Just as base 10 numbers are called *decimal* numbers, base 2 numbers are called *binary* numbers.)

A single digit of a binary number is thus the "atom" of computing, since all information is composed of **binary digits** or *bits*. This fundamental building block can be one of two values, which can be thought of as several alternatives: high or low, on or off, true or false, or 1 or 0.

Generalizing the point, in any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^i$$

where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: simply use the power of the base for that bit. We subscript decimal numbers with *ten* and binary numbers with *two*. For example,

1011_{two}

represents

$$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{ten}$$
$$= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{ten}$$
$$= 8 + 0 + 2 + 1_{ten}$$
$$= 11_{ten}$$

We number the bits 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a ARM word and the placement of the number 1011_{two}:

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 0 1 1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 31).

The ARM word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4,294,967,295_{ten}$):

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
0000 0000 0000 0000 0000 0000 0000 0010two = 2ten
. . .                    . . .
1111 1111 1111 1111 1111 1111 1111 1101two = 4,294,967,293ten
1111 1111 1111 1111 1111 1111 1111 1110two = 4,294,967,294ten
1111 1111 1111 1111 1111 1111 1111 1111two = 4,294,967,295ten
```

binary digit Also called **binary bit**. One of the two numbers in base 2, 0 or 1, that are the components of information.

least significant bit The rightmost bit in an ARM word.

most significant bit The leftmost bit in an ARM word.

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here xi means the ith bit of x):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the programming language, the operating system, and the program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude.*

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and a negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude representation was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

```
0000 0000 0000 0000 0000 0000 0000 0000two  =  0ten
0000 0000 0000 0000 0000 0000 0000 0001two  =  1ten
0000 0000 0000 0000 0000 0000 0000 0010two  =  2ten
. . .                                  . . .
0111 1111 1111 1111 1111 1111 1111 1101two  =  2,147,483,645ten
0111 1111 1111 1111 1111 1111 1111 1110two  =  2,147,483,646ten
0111 1111 1111 1111 1111 1111 1111 1111two  =  2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0000two  =  -2,147,483,648ten
1000 0000 0000 0000 0000 0000 0000 0001two  =  -2,147,483,647ten
1000 0000 0000 0000 0000 0000 0000 0010two  =  -2,147,483,646ten
. . .                                  . . .
```

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$$

The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{ten}$ ($2^{31} - 1$), use the same representation as before. The following bit pattern ($1000\ldots0000_{two}$) represents the most negative number $-2{,}147{,}483{,}648_{ten}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2{,}147{,}483{,}647_{ten}$ ($1000\ldots0001_{two}$) down to -1_{ten} ($1111\ldots1111_{two}$).

Two's complement does have one negative number, $-2{,}147{,}483{,}648_{ten}$, that has no corresponding positive number. Such imbalance was also a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with the number 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^1) + (x0 \times 2^0)$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

Binary to Decimal Conversion

What is the decimal value of this 32-bit two's complement number?

$$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_{two}$$

EXAMPLE

Substituting the number's bit values into the formula above:

ANSWER

$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$$
$$= -2^{31}\ \ +\ \ 2^{30}\ \ +\ \ 2^{29}\ \ +\ldots+\ \ 2^2\ \ +\ \ 0\ \ +\ \ 0$$
$$= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}$$
$$= -4_{ten}$$

We'll see a shortcut to simplify conversion from negative to positive soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers.

Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware/ Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`). Some C style guides even recommend declaring the former as `signed int` to keep the distinction clear.

Let's examine two useful shortcuts when working with two's complement numbers. The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111\ldots111_{two}$, which represents -1. Since $x + \bar{x} = -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$.

EXAMPLE

ANSWER

Negation Shortcut

Negate 2_{ten}, and then check the result by negating -2_{ten}.

$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

Negating this number by inverting the bits and adding one,

$$
\begin{array}{rl}
 & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\
+ & 1_{two} \\
\hline
= & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\
= & -2_{ten}
\end{array}
$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$
\begin{array}{rl}
& 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\
+ & \hspace{9cm} 1_{two} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\
= & 2_{ten}
\end{array}
$$

Our next shortcut tells us how to convert a binary number represented in *n* bits to a number represented with more than *n* bits. For example, the immediate field in the load, store, branch, ADD, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{ten}$ (-2^{15}) to $32{,}767_{ten}$ $(2^{15} - 1)$. To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension.*

Sign Extension Shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

EXAMPLE

ANSWER

The 16-bit binary version of the number 2 is

$$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$$0000\ 0000\ 0000\ 0010_{two}$$

becomes

$$1111\ 1111\ 1111\ 1101_{two}$$
$$+\qquad\qquad\qquad 1_{two}$$
$$=\ 1111\ 1111\ 1111\ 1110_{two}$$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}\ =\ {-2}_{ten}$$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement.

Check Yourself   What is the decimal value of this 64-bit two's complement number?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1000_{two}$$

1) -4_{ten}

2) -8_{ten}

3) -16_{ten}

4) $18{,}446{,}744{,}073{,}709{,}551{,}609_{ten}$

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its negative is 2^n; hence, the complement or negation of a two's complement number x is $2^n - x$.

A third alternative representation to two's complement and sign and magnitude is called one's complement. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several early scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00 \ldots 00_{two}$ is positive 0 and $11 \ldots 11_{two}$ is negative 0. The most negative number, $10 \ldots 000_{two}$, represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point in Chapter 3 is to represent the most negative value by $00 \ldots 000_{two}$ and the most positive value by $11 \ldots 11_{two}$, with 0 typically having the value $10 \ldots 00_{two}$. This is called a biased notation, since it biases the number such that the number plus the bias has a nonnegative representation.

one's complement
A notation that represents the most negative value by $10 \ldots 000_{two}$ and the most positive value by $01 \ldots 11_{two}$, leaving an equal number of negatives and positives but ending up with two zeros, one positive ($00 \ldots 00_{two}$) and one negative ($11 \ldots 11_{two}$). The term is also used to mean the inversion of every bit in a pattern: 0 to 1 and 1 to 0.

biased notation
A notation that represents the most negative value by $00 \ldots 000_{two}$ and the most positive value by $11 \ldots 11_{two}$, with 0 typically having the value $10 \ldots 00_{two}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

Elaboration: For signed decimal numbers, we used "−" to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal (see Figure 2.4) bit strings can encode the sign; hence we do not normally use "+" or "−" with binary or hexadecimal notation.

2.5 Representing Instructions in the Computer

We are now ready to explain the difference between the way humans instruct computers and the way computers see instructions.

Instructions are kept in the computer as a series of high and low electronic signals and may be represented as numbers. In fact, each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction.

Translating ARM Assembly Instructions into Machine Instructions

Let's do the next step in the refinement of the ARM language as an example. We'll show the real ARM language version of the instruction represented symbolically as

```
ADD r5,r1,r2
```

first as a combination of decimal numbers and then of binary numbers.

EXAMPLE

ANSWER

The decimal representation is

14	0	0	4	0	1	5	2

Each of these segments of an instruction is called a *field*. The fourth field (containing 4 in this case) tells the ARM computer that this instruction performs addition. The sixth field gives the number of the register that is the first source operand of the addition operation (1=r1), and the last field gives the other source operand for the addition (2 = r2). The seventh field contains the number of the register that is to receive the sum (5 = r5). Thus, this instruction adds (field 4 = 4) register r1 (field 6 = 1) to register r2 (field 8 = 2), and places the sum in register r5 (field 7 = 5); we'll reveal the purpose of the remaining four fields later.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

1110	00	0	0100	0	0001	0101	000000000010
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

instruction format
A form of representation of an instruction composed of fields of binary numbers.

machine language
Binary representation used for communication within a computer system.

hexadecimal
Numbers in base 16.

This layout of the instruction is called the **instruction format**. As you can see from counting the number of bits, this ARM instruction takes exactly 32 bits—the same size as a data word. In keeping with our design principle that simplicity favors regularity, all ARM instructions are 32 bits long.

To distinguish it from assembly language, we call the numeric version of instructions **machine language** and a sequence of such instructions *machine code*.

It would appear that you would now be reading and writing long, tedious strings of binary numbers. We avoid that tedium by using a higher base than binary that converts easily into binary. Since almost all computer data sizes are multiples of 4, **hexadecimal** (base 16) numbers are popular. Since base 16 is a power of 2, we can trivially convert by replacing each group of four binary digits by a single hexadecimal digit, and vice versa. Figure 2.4 converts between hexadecimal and binary.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

FIGURE 2.4 The hexadecimal-binary conversion table. Just replace one hexadecimal digit by the corresponding four binary digits, and vice versa. If the length of the binary number is not a multiple of 4, go from right to left.

Because we frequently deal with different number bases, to avoid confusion we will subscript decimal numbers with *ten*, binary numbers with *two*, and hexadecimal numbers with *hex*. (If there is no subscript, the default is base 10.) By the way, C and Java use the notation 0x*nnnn* for hexadecimal numbers.
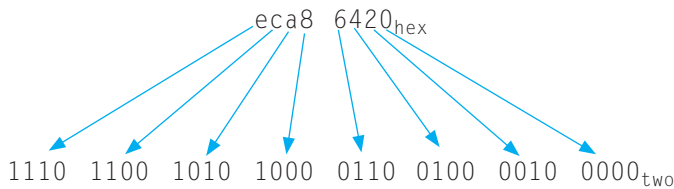
EXAMPLE

Binary to Hexadecimal and Back

Convert the following hexadecimal and binary numbers into the other base:

eca8 6420$_{hex}$

0001 0011 0101 0111 1001 1011 1101 1111 $_{two}$

ANSWER

Using Figure 2.4, the answer is just a table lookup one way:

eca8 6420$_{hex}$

1110 1100 1010 1000 0110 0100 0010 0000$_{two}$

And then the other direction:

0001 0011 0101 0111 1001 1011 1101 1111$_{two}$

1357 9bdf$_{hex}$
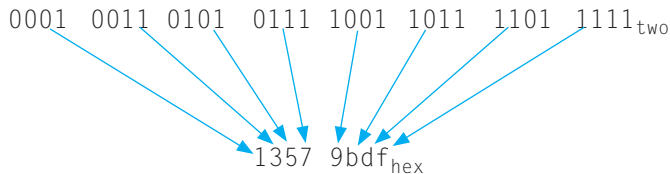
ARM Fields

ARM fields are given names to make them easier to discuss:

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

Here is the meaning of each name of the fields in ARM instructions:

- *Opcode:* Basic operation of the instruction, traditionally called the **opcode**.
- *Rd:* The register destination operand. It gets the result of the operation.
- *Rn:* The first register source operand.

opcode The field that denotes the operation and format of an instruction.

- *Operand2:* The second source operand.

- *I:* Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate. (Section 2.10 goes into details on ARM immediates, but for now we'll just assume its just a plain constant.)

- *S:* Set Condition Code. Described in Section 2.7, this field is related to conditional branch instructions.

- *Cond:* Condition. Described in Section 2.7, this field is related to conditional branch instructions.

- *F:* Instruction Format. This field allows ARM to different instruction formats when needed.

Let's look at the add word instruction from page 83 that has a constant operand:

```
ADD     r3,r3,#4        ; r3 = r3 + 4
```

As you might expect, the constant 4 is placed in the Operand2 field and the I field is set to 1. We make a small change the binary version from before.

14	0	1	4	0	3	3	4

The Opcode field is still 4 so that this instruction performs addition. The Rn and Rd fields gives the numbers of the register that is the first source operand (3) and the register to receive the sum (3). Thus, this instruction adds 4 to r3 and place the sum in r3.

Now let's try load word instruction from page 82:

```
LDR     r5,[r3, #32]    ; Temporary reg r5 gets A[8]
```

Loads and stores use a different instruction format from above, with just 6 fields:

Cond	F	Opcode	Rn	Rd	Offset12
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

To tell ARM that the format is different, the F field now as 1, meaning that this is a data transfer instruction format. The opcode field has 24, showing that this instruction does load word. The rest of the fields are straightword: Rn field has 3 for the base register, the Offset12 field has 32 as the offset to add to the base register, and the Rd field has 5 for the *destination* register, which receives the result of the load:

14	1	24	3	5	32
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

Let's call the first option the *data processing* (DP) instruction format and the second the *data transfer* (*DT*) instruction format.

Although multiple formats complicate the hardware, we can reduce the complexity by keeping the formats similar. For example, the first two fields and the last three fields of the two formats are the same size and four of them have the same names; the length of the Opcode field in DT format is equal to the sum of the lengths of three fields of the DP format.

Figure 2.5 shows the numbers used in each field for the ARM instructions covered here.

Instruction	Format	Cond	F	I	op	S	Rn	Rd	Operand2
ADD	DP	14	0	0	4_{ten}	0	reg	reg	reg
SUB (subtract)	DP	14	0	0	$2s_{ten}$	0	reg	reg	reg
ADD (immediate)	DP	14	0	1	4_{ten}	0	reg	reg	constant
LDR (load word)	DT	14	1	n.a.	24_{ten}	n.a.	reg	reg	address
STR (store word)	DT	14	1	n.a.	25_{ten}	n.a.	reg	reg	address

FIGURE 2.5 ARM instruction encoding. In the table above, "reg" means a register number between 0 and 15, "constant" means a 12-bit constant, "address" means a 12-bit address. "n.a." (not applicable) means this field does not appear in this format, and Op stands for opcode.

Translating ARM Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If r3 has the base of the array A and r2 corresponds to h, the assignment statement

  A[30] = h + A[30];

is compiled into

```
LDR    r5,[r3,#120]  ; Temporary reg r5 gets A[30]
ADD    r5,r2,r5      ; Temporary reg r5 gets h + A[30]
STR    r5,[r3,#120]  ; Stores h + A[30] back into A[30]
```

What is the ARM machine language code for these three instructions?

EXAMPLE

ANSWER

For convenience, let's first represent the machine language instructions using decimal numbers. From Figure 2.5, we can determine the three machine language instructions:

| Cond | F | opcode | | | Rn | Rd | Offset12 |
		I	opcode	S			Operand2
14	1		24		3	5	120
14	0	0	4	0	2	5	5
14	1		25		3	5	120

The LDR instruction is identified by 24 (see Figure 2.5) in the third field (opcode). The base register 3 is specified in the fourth field (Rn), and the destination register 5 is specified in the sixth field (Rd). The offset to select A[30] ($120 = 30 \times 4$) is found in the final field (offset12).

The ADD instruction that follows is specified with 4 in the fourth field (opcode). The three register operands (2, 5, and 5) are found in the sixth, seventh, and eighth fields.

The STR instruction is identified with 25 in the third field. The rest of this final instruction is identical to the LDR instruction.

Since $120_{ten} = 0000\ 1111\ 0000_{two}$, the binary equivalent to the decimal form is:

| Cond | F | opcode | | | Rn | Rd | Off |
		I	opcode	S			Operand2
1110	1		11000		0011	0101	0000 1111 0000
1110	0	0	100	0	0010	0101	0000 0000 0101
1110	1		11001		0011	0101	0000 1111 0000

Note the similarity of the binary representations of the first and last instructions. The only difference is in the last bit of the opcode.

Figure 2.6 summarizes the portions of ARM machine language described in this section. As we shall see in Chapter 4, the similarity of the binary representations of related instructions simplifies hardware design. These similarities are another example of regularity in the ARM architecture.

ARM machine language

Name	Format	Example								Comments
ADD	DP	14	0	0	4	0	2	1	3	ADD r1,r2,r3
SUB	DP	14	0	0	2	0	2	1	3	SUB r1,r2,r3
LDR	DT	14	1	24			2	1	100	LDR r1,100(r2)
STR	DT	14	1	25			2	1	100	STR r1,100(r2)
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format

FIGURE 2.6 ARM architecture revealed through Section 2.5. The two ARM instruction formats so far are DP and DT. The last 16 bits have the same sized fields: both contain an *Rn* field, giving one of the sources; and an *Rd* field, specifying the destination register.

Today's computers are built on two key principles:

1. Instructions are represented as numbers.

2. Programs are stored in memory to be read or written, just like numbers.

These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. Figure 2.7 shows the power of the concept; specifically, memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code.

One consequence of instructions as numbers is that programs are often shipped as files of binary numbers. The commercial implication is that computers can inherit ready-made software provided they are compatible with an existing instruction set. Such "binary compatibility" often leads industry to align around a small number of instruction set architectures.

The **BIG** Picture

What ARM instruction does this represent?

Check Yourself

Cond	F	I	opcode	S	Rn	Rd	Operand2
14	0	0	4	0	0	1	2

Choose from one of the four options below.

1. ADD r0, r1, r2
2. ADD r1, r0, r2
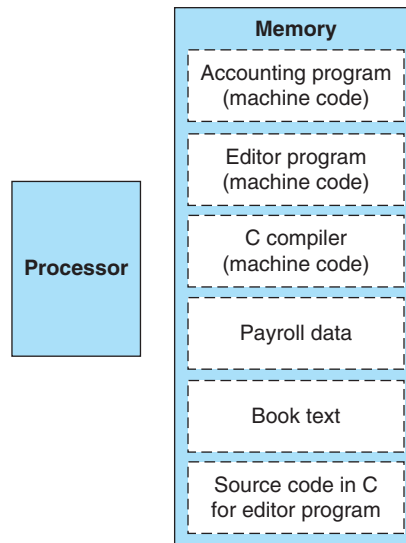3. ADD r2, r1, r0
4. SUB r2, r0, r1

FIGURE 2.7   The stored-program concept. Stored programs allow a computer that performs accounting to become, in the blink of an eye, a computer that helps an author write a book. The switch happens simply by loading memory with programs and data and then telling the computer to begin executing at a given location in memory. Treating instructions in the same way as data greatly simplifies both the memory hardware and the software of computer systems. Specifically, the memory technology needed for data can also be used for programs, and programs like compilers, for instance, can translate code written in a notation far more convenient for humans into code that the computer can understand.

"Contrariwise,"
continued Tweedledee,
"if it was so, it might
be; and if it were so,
it would be; but as it
isn't, it ain't. That's
logic."

Lewis Carroll, *Alice's*
Adventures in
Wonderland, 1865

2.6   Logical Operations

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.8 shows logical operations in C, Java, and ARM.

Logical operations	C operators	Java operators	ARM instructions
Bit-by-bit AND	&	&	AND
Bit-by-bit OR	|	|	ORR
Bit-by-bit NOT	~	~	MVN
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR

FIGURE 2.8 C and Java logical operators and their corresponding ARM instructions. ARM implements NOT using a NOR with one operand being zero.

The first class of such operations Another useful operation that isolates fields is **AND**. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register r2 contains

0000 0000 0000 0000 0000 1101 1100 0000$_{two}$

and register r1 contains

0000 0000 0000 0000 0011 1100 0000 0000$_{two}$

then, after executing the ARM instruction

```
    AND r5,r1,r2        ; reg r5 = reg r1 & reg r2
```

the value of register r5 would be

0000 0000 0000 0000 0000 1100 0000 0000$_{two}$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask "conceals" some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers r1 and r2 are unchanged from the preceding example, the result of the ARM instruction

```
    ORR r5,r1,r2        ; reg r5 = reg r1 | reg r2
```

is this value in register r5:

0000 0000 0000 0000 0011 1101 1100 0000$_{two}$

AND A logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

OR A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

The third logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. If the register r1 is unchanged from the preceding example, the result of the ARM instruction Move Not (MVN):

```
MVN r5,r1           ; reg r5 = ~reg r1
```

is this value in register r5:

$$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{two}$$

Although not a logical instruction, a useful instruction related to MVN that we've not listed so far simply copies one register to another without changing it. The Move instruction (MOV) does just what you'd think:

```
MOV r6,r5           ; reg r6 = reg r5
```

Register r6 now has the value of the contents of r5.

Another class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register r0 contained

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

and the instruction to shift left by 4 was executed, the new value would be:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

Shift left logical provides a bonus benefit. Shifting left by i bits gives the same result as multiplying by 2^i, just as shifting a decimal number by i digits is equivalent to multiplying by 10^i. For example, the above LSL shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and $9 \times 16 = 144$, the value of the second bit pattern.

The dual of a shift left is a shift right. The actual name of the two ARM shift operations are called *logical shift left* (LSL) and *logical shift right* (LSR).

Although ARM has shift operations, they are not separate instructions. How can this be? Unlike any other microprocessor instruction set, ARM offers the ability to shift the second operand as part of any data processing instruction!

For example, this variation of the add instruction adds register r1 to register r2 shifted left by 2 bits and puts the result in register r5:

```
ADD r5,r1,r2, LSL #2     ; r5 = r1 + (r2 << 2)
```

In case you were wondering, ARM hardware is designed so that the adds with shifts are no slower than regular adds.

If you just wanted to shift register r5 right by 4 bits and place the result in r6, you could do that with a move instruction:

```
    MOV r6,r5, LSR #4   ; r6 = r5 >> 4
```

Although usually programmers want to shift by a constant, ARM allows shifting by the value found in a register. The following instruction shifts register r5 right by the amount in register r3 and places the result in r6.

```
    MOV r6,r5, LSR r3   ; r6 = r5 >> r3
```

Figure 2.8.5 shows that these shift operations are encoded in the 12 bit Operand2 field of the Data Processing instruction format. If the shift field is 0, the operation is logical shift left, and if it is 1 then the operation is logical shift right. Here are the machine language versions of the three instructions above with shifts operations:

Cond	F	I	opcode	S	Rn	Rd	Shift_imm		Shift		Rm
							Rs	0	Shift		Rm
14	0	0	4	0	2	5	2		0	0	5
14	0	0	13	0	0	6	4		1	0	5
14	0	0	13	0	0	6	3	0	1	1	5

Note that if these new fields are 0, then there operand is not shifted, so the encodings shown in the examples in prior sections work properly.

Figure 2.8 above shows the relationship between the C and Java operators and the ARM instructions.

11	8	7	6	5	4	3		0
shift_imm			Shift		0		Rm	
Rs		0	Shift		1		Rm	

FIGURE 2.8.5   Encoding of shift operations inside Operand2 field of Data Processing instruction format.

Elaboration: The full ARM instruction set also includes exclusive or (EOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. It also has Bit Clear (BIC), which sets to 0 any bit that is a 1 in the second operand. C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical operations in ARM: AND, ORR, LSL, and LSR.

Elaboration: ARM has more shift operations than LSL and LSR. Arithmetic shift right (ASR) replicates the sign bit during a shift; we'll see what they were hoping to do with ASR (but failed) in the fallacy in Section 3.8 of the next chapter. Instead of discarding the bits on a right shift, Rotate Right (ROR) brings them back into the vacated upper bits. As the name suggests, the bit pattern can be thought of as a ring that rotates in a register but are never lost.

Check Yourself

Which operations can isolate a field in a word?

1. AND

2. A shift left followed by a shift right

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. ...This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

2.7   Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. ARM assembly language includes many decision-making instructions, similar to an *if* statement with a *go to*. Let's start with an instruction that compares two values followed by an instruction that branches to L1 if the registers are equal

```
CMP register1, register2
BEQ L1
```

This pair of instructions means go to the statement labelled L1 if the value in register1 equals the value in register2. The mnemonic CMP stands for *compare* and BEQ stands for *branch if equal*. Another example is the instruction pair
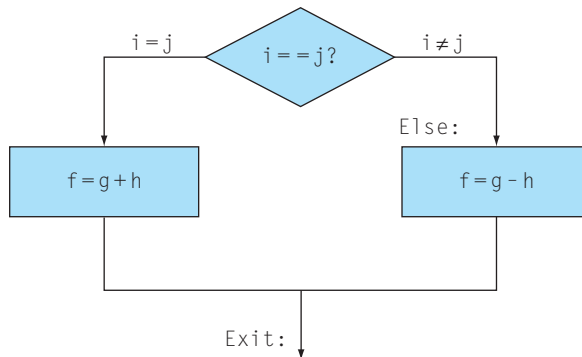
```
CMP register1, register2
BEQ L1
```

FIGURE 2.9 **Illustration of the options in the *if* statement above.** The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

This pair goes to the statement labelled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `BNE` stands for *branch if not equal*. BEQ and BNE are traditionally called **conditional branches**.

conditional branch
An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

Compiling *if-then-else* into Conditional Branches

In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers r0 through r4, what is the compiled ARM code for this C *if* statement?

EXAMPLE

```
if (i == j) f = g + h; else f = g - h;
```

Figure 2.9 is a flowchart of what the ARM code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (BEQ). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label Else is defined below) and so we use the branch if registers are *not* equal instruction (BNE):

ANSWER

```
CMP r3,r4
BNE, Else   ; go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
ADD r0,r1,r2        ; f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch.* This instruction says that the processor always follows the branch (the label Exit is defined below).

```
B Exit       ; go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label Else to this instruction. We also show the label Exit that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:SUB r0,r1,r2   ; f = g - h (skipped if i = j)
Exit:
```

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

Hardware/ Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a *while* Loop in C

Here is a traditional loop in C:

```
while (save[i] == k)
     i += 1;
```

Assume that i and k correspond to registers r3 and r5 and the base of the array save is in r6. What is the ARM assembly code corresponding to this C segment?

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. Fortunately, we can use the logical shift left operation, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 101 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: ADD r12,r6, r3, LSL #2    ; r12 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
LDR  r0,[r12,#0]    ; Temp reg r0 = save[i]
```

The next instruction pair performs the loop test, exiting if `save[i]` ≠ `k`:

```
CMP  r0,r5
BNE  Exit  ; go to Exit if save[i] ≠ k
```

The next instruction adds `1` to `i`:

```
ADD  r3,r3,#1       ; i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
B    Loop  ; go to Loop

Exit:
```

(See the exercises for an optimization of this sequence.)

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

Hardware/ Software Interface

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Thus, ARM has a large set of conditional branches. For example, LT, LE, GT, and GE branch if the

result of the compare is less than, less than or equal, greater than, or greater than or equal, respectively.

Conditional branches are actually based on *condition flags* or *condition codes* that can be set by the CMP instruction. Branches then test the condition flags. Condition flags form a special purpose register who values can be tested by a conditional branch instruction at any time after the compare instruction, not just by the following instruction.

Moreover, the condition flags can be set by many other instructions than compare; in such a case, the result of the operation is compared to 0. The S bit in Figure 2.5 is used to set the condition codes as part of data processing instructions. The programmer specifies setting the condition flags by appending an S to the instruction name. Thus, SUBS sets the condition flags with the result of the subtraction, while SUB leaves the condition codes unchanged.

Hardware/ Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

ARM offers more versions conditional branch to handle these alternatives: Less than unsigned is call LO for lower; less than or equal unsigned is called LS for lower or same; Great than unsigned is called HI for higher; Greater or equal unsigned is called HS for higher or same.

Signed versus Unsigned Comparison

Suppose register r0 has the binary number

    $1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111_{two}$

and that register r1 has the binary number

    $0000\ \ 0000\ \ 0000\ \ 0000\ \ 0000\ \ 0000\ \ 0000\ \ 0001_{two}$

EXAMPLE

and the following instruction is executed.

```
CMP   r0, r1
```

Which conditional branch is taken?

```
BLO   L1 ; unsigned branch
BLT   L2 ; signed branch
```

The value in register r0 represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register r1 represents 1_{ten} in either case. The branch on lower unsigned instruction (BLO) is *not* taken to L1, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$. However, the branch on less than instruction (BLT) is taken to L2, since $-1_{ten} < 1_{ten}$.

ANSWER

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \le x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: branch to IndexOutOfBounds if $r1 \ge r2$ or if r1 is negative.

EXAMPLE

The checking code just uses BHS to do both checks:

ANSWER

```
CMP r1,r2
BHS IndexOutOfBounds ;if r1>=r2 or r1<0, go to Error
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

jump address table
Also called **jump table**. A table of addresses of alternative instruction sequences.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program needs to jump using the address in the appropriate entry from the jump table.

ARM has a surprisingly easy way to handle such situations. Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed. For historical reasons, this register is almost always called the **program counter**, abbreviated *pc* in the ARM architecture, although a more sensible name would have been *instruction address register*. Register 15 is actually the program counter in ARM, so a `LDR` instruction with the destination register 15 means an unconditional branch to the address specified in memory. In fact, any instruction with register 15 as the destination register is an unconditional branch to the address at that value. In Section 2.8, we'll see how this trick is useful when returning from a procedure.

program counter (PC) The register containing the address of the instruction in the program being executed.

Hardware/ Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

Encoding Branch Instructions in ARM

A problem occurs when an instruction needs longer fields than those shown above in Section 2.6. For example, the longest field in the instruction formats above is just 12 bits, suggesting that branch field might be just 12 bits. That might limit programs to just 2^{12} or 4096 bytes!

Hence, we have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format. This leads us to the final hardware design principle:

Design Principle 4: Good design demands good compromises.

The compromise chosen by the ARM designers is to keep all instructions the same length, thereby requiring different kinds of instruction formats for different kinds of instructions. We saw the DP and DT formats above, which had many similarities. Branch has a third instruction type:

Cond	12	address
4 bits	4 bits	24 bits

Value	Meaning	Value	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

FIGURE 2.9.5 Encodings of Options for Cond field.

The Cond field encodes the many versions of the conditional branch mentioned above. Figure 2.9.5 shows those encodings.

You might think that the 24-bit address would extend the program limit to 2^{24} or 16 MB, which would be fine for many programs but constrain some large ones. An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

$$\text{Program counter} = \text{Register} + \text{Branch address}$$

This sum allows the program to be as large as 2^{32}, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the program counter (PC) contains the address of the current instruction, we can branch within $\pm 2^{24}$ words of the current instruction if we use the PC as the register to be added to the address. All loops and *if* statements are much smaller than 2^{24} words, so the PC is the ideal choice.

This form of branch addressing is called **PC-relative addressing**. As we shall see in Chapter 4, it is convenient for the hardware to increment the PC early. Hence, the ARM address is actually relative to the address of the instruction two after the branch (PC + 8) as opposed to the current instruction (PC).

Since all ARM instructions are 4 bytes long, ARM stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 24-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address.

PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.

Conditional Execution

Another unusual feature of ARM is that most instructions can be conditionally executed, not just branches. That is the purpose of the 4-bit Cond field found

in most ARM instruction formats. The assembly language programmer simply appends the desired condition to the instruction name, telling the computer to perform the operation only if the condition is true based on the last time the condition flags were set. That is, ADDEQ performs the addition only if the condition flags suggest the operands were equal for the operation that set the flags.

For example, the ARM code to perform the if statement in Figure 2.9 is reproduced below:

```
        CMP r3,r4
        BNE Else       ; go to Else if i ≠ j
        ADD r0,r1,r2   ; f = g + h (skipped if i ≠ j)
        B Exit         ; go to Exit
   Else:SUB r0,r1,r2   ; f = g - h (skipped if i = j)
   Exit:
```

The desired result can be achieved without any branches using conditional execution:

```
        CMP r3,r4
        ADDEQ r0,r1,r2 ; f = g + h (skipped if i ≠ j)
        SUBNE r0,r1,r2 ; f = g - h (skipped if i = j)
```

Figure 2.9.5 shows the encodings for conditional execution of all instructions, not just branches. Note that 14 means always execute. Thus, Figures 2.5 and 2.6 in Section 2.5 show the value 14 in the Cond field of the machine language instructions to indicate the version of instructions that are always executed.

Conditional execution provides a technique to execute instructions depending on a test without using conditional branch instructions. Chapter 4 shows that branches can lower the performance of pipelined computers, so removing branches can help performance even more than the reduction in instructions suggests.

Check Yourself   I. C has many statements for decisions and loops, while ARM has few. Which of the following do or do not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.

2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.

3. More decision statements mean fewer lines of code, which generally reduces coding time.

4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while ARM doesn't?

1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.

2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.

3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

2.8 Supporting Procedures in Computer Hardware

A **procedure** or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in Section 2.15 on the CD, but Java needs everything from a computer that C needs.

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a "need to know" basis, so the spy can't make assumptions about his employer.

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.

2. Transfer control to the procedure.

3. Acquire the storage resources needed for the procedure.

4. Perform the desired task.

5. Put the result value in a place where the calling program can access it.

6. Return control to the point of origin, since a procedure can be called from several points in a program.

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. ARM software follows the following convention for procedure calling in allocating its 16 registers:

- r0–r3: four argument registers in which to pass parameters

- lr: one link register containing the return address register to return to the point of origin

In addition to allocating these registers, ARM assembly language includes an instruction just for the procedures: it jumps to an address and simultaneously saves the address of the following instruction in register lr. The **Branch-and-Link instruction** (BL) is simply written

```
BL    ProcedureAddress
```

The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link," stored in register lr (register 14), is called the **return address**. The return address is needed because the same procedure could be called from several parts of the program.

To return, ARM just uses the move instruction to copy the link register into the PC, which causes an unconditional branch to the address specified in a register:

```
MOV  pc, lr
```

This instruction branches to the address stored in register lr—which is just what we want. Thus, the calling program, or **caller,** puts the parameter values in r0–r3 and uses BL X to jump to procedure X (sometimes named the **callee**). The callee then performs the calculations, places the results (if any) into r0 and r1, and returns control to the caller using MOV pc, lr. The BL instruction actually saves PC + 4 in register lr to link to the following instruction to set up the procedure return.

Using More Registers

Suppose a compiler needs more registers for a procedure than the four argument and two return value registers. Since we must cover our tracks after our mission is complete, any registers needed by the caller must be restored to the values that they contained *before* the procedure was invoked. This situation is an example in which we need to spill registers to memory, as mentioned in the *Hardware/Software Interface* section.

The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue. A stack needs a pointer to the most recently allocated address in the stack to show where the next procedure should place the registers to be spilled or where

Branch-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (lr or register 14 in ARM).

return address A link to the calling site that allows a procedure to return to the proper address; in ARM it is stored in register lr (register 14).

caller The program that instigates a procedure and provides the necessary parameter values.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

stack A data structure for spilling registers organized as a last-in-first-out queue.

old register values are found. The **stack pointer** is adjusted by one word for each register that is saved or restored. ARM software reserves register 13 for the stack pointer, giving it the obvious name sp. Stacks are so popular that they have their own buzzwords for transferring data to and from the stack: placing data onto the stack is called a **push,** and removing data from the stack is called a **pop.**

By historical precedent, stacks "grow" from higher addresses to lower addresses. This convention means that you push values onto the stack by subtracting from the stack pointer. Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In ARM, it is register sp (register 13).

push Add element to stack.

pop Remove element from stack.

Compiling a C Procedure That Doesn't Call Another Procedure

Let's turn the example on page 79 from Section 2.2 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
     int f;

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

What is the compiled ARM assembly code?

EXAMPLE

The parameter variables g, h, i, and j correspond to the argument registers r0, r1, r2, and r3, and f corresponds to r4. The compiled program starts with the label of the procedure:

ANSWER

```
leaf_example:
```

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 79, which uses two temporary registers. Thus, we need to save three registers: r4, r5, and r6. We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
SUB  sp, sp, #12      ; adjust stack to make room for 3 items
STR  r6, [sp,#8]      ; save register r6 for use afterwards
STR  r5, [sp,#4]      ; save register r5 for use afterwards
STR  r4, [sp,#0]      ; save register r4 for use afterwards
```
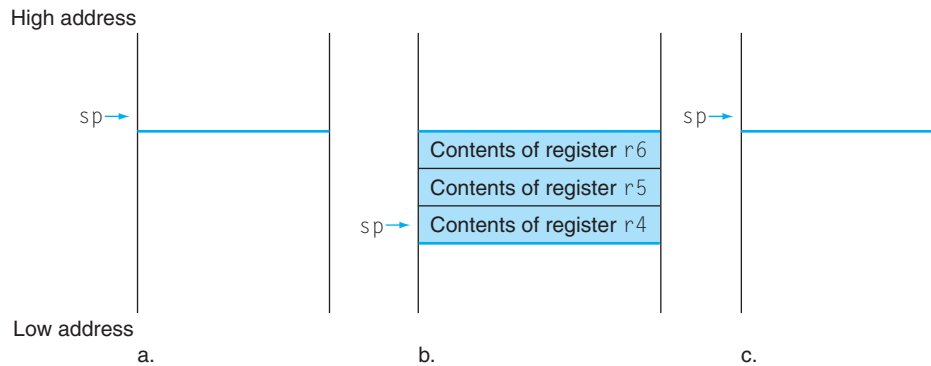
FIGURE 2.10   The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Figure 2.10 shows the stack before, during, and after the procedure call.

The next three statements correspond to the body of the procedure, which follows the example on page 79:

```
ADD r5,r0,r1 ; register r5 contains g + h
ADD r6,r2,r3 ; register r6 contains i + j
SUB r4,r5,r6 ; f gets r5 - r6, which is (g + h) - (i + j)
```

To return the value of f, we copy it into a return value register r0:

```
MOV r0,r4 ; returns f (r0 = r4)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
LDR   r4, [sp,#0] ; restore register r4 for caller
LDR   r5, [sp,#4] ; restore register r5 for caller
LDR   r6, [sp,#8] ; restore register r6 for caller
ADD sp,sp,#12     ; adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
MOV pc, lr ; jump back to calling routine
```

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, ARM software separates 12 of the registers into two groups:

- ■ r0–r3, r12: argument or scratch registers that are *not* preserved by the callee (called procedure) on a procedure call
- ■ r4–r11: eight variable registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, if we could rewrite the code to use r12 and reuse one of the r0 to r3, we can drop two stores and two loads from the code. We still must save and restore r4, since the callee must assume that the caller needs its value.

Nested Procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register r0 and then using BL A. Then suppose that procedure A calls procedure B via BL B with an argument of 7, also placed in r0. Since A hasn't finished its task yet, there is a conflict over the use of register r0. Similarly, there is a conflict over the return address in register lr, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the extra registers. The caller pushes any argument registers (r0–r3) that are needed after the call. The callee pushes the return address register lr and any variable registers (r4–r11) used by the callee. The stack pointer sp is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

Compiling a Recursive C Procedure, Showing Nested Procedure Linking

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
    if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

What is the ARM assembly code?

EXAMPLE

ANSWER

The parameter variable n corresponds to the argument register r0. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and r0:

```
fact:
    SUB   sp, sp, #8      ; adjust stack for 2 items
    STR   lr, [sp,#8]     ; save the return address
    STR   r0, [sp,#0]     ; save the argument n
```

The first time fact is called, STR saves an address in the program that called fact. The next two instructions test whether n is less than 1, going to L1 if n ≥ 1.

```
    CMP   r0,#1           ; compare n to 1
    BGE   L1             ; if n >= 1, go to L1
```

If n is less than 1, fact returns 1 by putting 1 into a value register: it moves 1 to r0. It then pops the two saved values off the stack and jumps to the return address:

```
    MOV   r0,#1           ; return 1
    ADD   sp,sp,#8        ; pop 2 items off stack
    MOV   pc,lr           ; return to the caller
```

Before popping two items off the stack, we could have loaded r0 and lr. Since r0 and lr don't change when n is less than 1, we skip those instructions.

If n is not less than 1, the argument n is decremented and then fact is called again with the decremented value:

```
L1:   SUB r0,r0,#1    ; n >= 1: argument gets (n - 1)
      BL  fact        ; call fact with (n - 1)
```

The next instruction is where fact returns. First we save the returned value and restore the old return address and old argument, along with the stack pointer:

```
MOV  r12,r0         ; save the return value
LDR  r0, [sp,#0]    ; return from BL: restore argument n
LDR  lr, [sp,#0]    ; restore the return address
ADD  sp, sp, #8     ; adjust stack pointer to pop 2
items
```

Next, the return value register r0 gets the product of old argument r0 and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:

```
    MUL r0,r0,r12     ; return n * fact (n - 1)
```

Finally, fact jumps again to the return address:

```
    MOV  pc,lr            ; return to the caller
```

Hardware/ Software Interface

A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*. Examples include integers and characters (see Section 2.9). C has two storage classes: *automatic* and *static*. Automatic variables are local to a procedure and are discarded when the procedure exits. Static variables exist across exits from and entries to procedures. C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*. The rest are automatic.

Figure 2.11 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above sp is preserved simply by making sure the callee does not write above sp; sp is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Preserved	Not preserved
Variable registers: r4-r11	Argument registers: r0-r3
Stack pointer register: sp	Intra-procedure-call scatch register: r12
Link register: lr	Stack below the stack pointer
Stack above the stack pointer	

FIGURE 2.11   What is and what is not preserved across a procedure call.

Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

Allocating Space for New Data on the Heap

In addition to automatic variables that are local to procedures, C programmers need space in memory for static variables and for dynamic data structures. Figure 2.13 shows the ARM convention for allocation of memory. The stack starts in the high end of memory and grows down. The first part of the low end of memory is reserved, followed by the home of the ARM machine code, traditionally called the **text segment**. Above the code is the *static data segment*, which is the place for constants and other static variables. Although arrays tend to be a fixed length

procedure frame Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

text segment The segment of a UNIX object file that contains the machine language code for routines in the source file.

High address

Low address

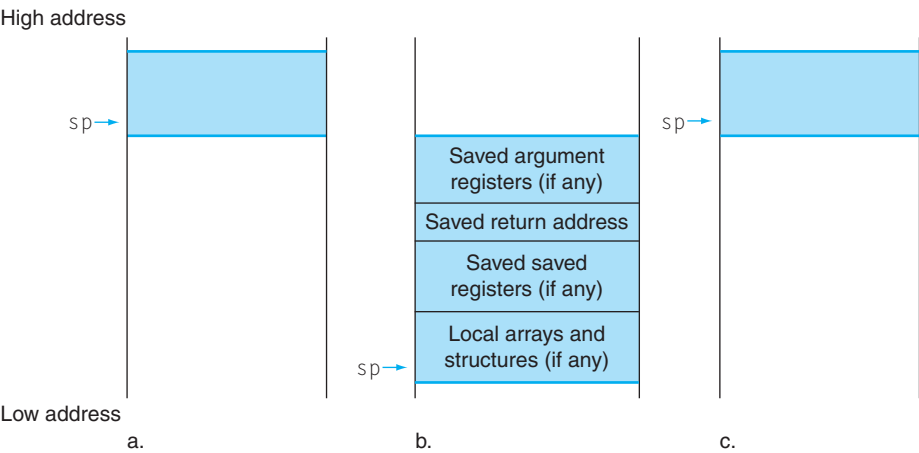   a.   b.   c.

FIGURE 2.12   Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The stack pointer (sp) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the stack pointer.
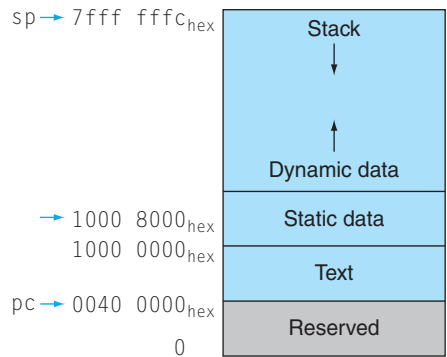
FIGURE 2.13   Typical ARM memory allocation for program and data. These addresses are only a software convention, and not part of the ARM architecture. The stack pointer is initialized to 7fff fffc$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000$_{hex}$. The static data starts at 1000 0000$_{hex}$. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap.

and thus are a good match to the static data segment, data structures like linked lists tend to grow and shrink during their lifetimes. The segment for such data structures is traditionally called the *heap,* and it is placed next in memory. Note that this allocation allows the stack and heap to grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.

   C allocates and frees space on the heap with explicit functions. malloc() allocates space on the heap and returns a pointer to it, and free() releases space

on the heap to which the pointer points. Memory allocation is controlled by programs in C, and it is the source of many common and difficult bugs. Forgetting to free space leads to a "memory leak," which eventually uses up so much memory that the operating system may crash. Freeing space too early leads to "dangling pointers," which can cause pointers to point to things that the program never intended. Java uses automatic memory allocation and garbage collection just to avoid such bugs.

Figure 2.14 summarizes the register conventions for the ARM assembly language.

Name	Register number	Usage	Preserved on call?
a1-a2	0–1	Argument / return result / scratch register	no
a3-a4	2–3	Argument / scratch register	no
v1-v8	4–11	Variables for local routine	yes
ip	12	Intra-procedure-call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link Register (Return address)	yes
pc	15	Program Counter	n.a.

FIGURE 2.14 ARM register conventions.

Elaboration: What if there are more than four parameters? The ARM convention is to place the extra parameters on the stack. The procedure then expects the first four parameters to be in registers r0 through r3 and the rest in memory, addressable via the stack pointer.

Elaboration: Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the over-head associated with procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
  if (n > 0)
    return sum(n - 1, acc + n);
  else
    return acc;
}
```

Consider the procedure call sum(3,0). This will result in recursive calls to sum (2,3), sum(1,5), and sum(0,6), and then the result 6 will be returned four times. This recursive call of sum is referred to as a *tail call*, and this example use of tail recursion can be implemented very efficiently (assume r0 = n and r1 = acc):

```
sum: CMP r0, #0          ; test if n <= 0
     BLE sum_exit         ; go to sum_exit if n <= 0
```

```
        ADD r1, r1, r0          ; add n to acc
        SUB r0, r0, #1          ; subtract 1 from n
        B sum                   ; go to sum
sum_exit:
        MOV  r0, r1             ; return value acc
        MOV  pc, lr             ; return to caller
```

Check Yourself

Which of the following statements about C and Java are generally true?

1. C programmers manage data explicitly, while it's automatic in Java.

2. C leads to more pointer bugs and memory leak bugs than does Java.

!(@|=> (wow open tab at bar is great)

Fourth line of the keyboard poem "Hatless Atlas," 1991 (some give names to ASCII characters: "!" is "wow," "(" is open, "|" is bar, and so on).

2.9 Communicating with People

Computers were invented to crunch numbers, but as soon as they became commercially viable they were used to process text. Most computers today offer 8-bit bytes to represent characters, with the American Standard Code for Information Interchange (ASCII) being the representation that nearly everyone follows. Figure 2.15 summarizes ASCII.

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	;	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124		
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the ARM Reference Data Card at the front of this book.

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

<div style="text-align: right">Hardware/
Software
Interface</div>

ASCII versus Binary Numbers

We could represent numbers as strings of ASCII digits instead of as integers. How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

EXAMPLE

One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be (10 × 8)/32 or 2.5. In addition to the expansion in storage, the hardware to ADD, subtract, multiply, and divide such decimal numbers is difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

ANSWER

A series of instructions can extract a byte from a word, so load word and store word are sufficient for transferring bytes as well as words. Because of the popularity of text in some programs, however, ARM provides instructions to move bytes. Load register byte (LDRB) loads a byte from memory, placing it in the rightmost 8 bits of a register. Store register byte (STRB) takes a byte from the rightmost 8 bits of a register and writes it to memory. Thus, we copy a byte with the sequence

```
LDRB r0,[sp,#0]          ; Read byte from source
STRB r0,[r10,#0]         ; Write byte to destination
```

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

<div style="text-align: right">Hardware/
Software
Interface</div>

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. ARM does offer two flavors of byte loads: *load register signed byte* (LDRSB) treats the byte as a signed number and thus sign-extends to fill the 24 leftmost bits of the register, while *load register byte* (LDRB) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, LDRB is used practically exclusively for byte loads.

Characters are normally combined into strings, which have a variable number of characters. There are three choices for representing a string: (1) the first position of the string is reserved to give the length of a string, (2) an accompanying variable has the length of the string (as in a structure), or (3) the last position of a string is indicated by a character used to mark the end of a string. C uses the third choice, terminating a string with a byte whose value is 0 (named null in ASCII). Thus, the string "Cal" is represented in C by the following 4 bytes, shown as decimal numbers: 67, 97, 108, 0. (As we shall see, Java uses the first option.)

Compiling a String Copy Procedure, Showing How to Use C Strings

EXAMPLE

The procedure strcpy copies string y to string x using the null byte termination convention of C:

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

What is the ARM assembly code?

ANSWER

Below is the basic ARM assembly code segment. Assume that base addresses for arrays x and y are found in r0 and r1, while i is in r4. strcpy adjusts the stack pointer and then saves the saved register r4 on the stack:

```
strcpy:
    SUB    sp, sp, #4    ; adjust stack for 1 more item
    STR    r4, [sp,#0]   ; save r4
```

To initialize i to 0, the next instruction sets r4 to 0 by adding 0 to 0 and placing that sum in r4:

```
        MOV     r4, #0 ; i = 0 + 0
```

This is the beginning of the loop. The address of y[i] is first formed by adding i to y[]:

```
  L1:   ADD     r2,r4,r1 ; address of y[i] in r2
```

Note that we don't have to multiply i by 4 since y is an array of *bytes* and not of words, as in prior examples.

To load the character in y[i], we use load byte unsigned, which puts the character into r3:

```
  LDRBS   r3, [r2,#0]  ; r3 = y[i] and set condition flags
```

A similar address calculation puts the address of x[i] in r12, and then the character in r3 is stored at that address.

```
        ADD     r12,r4,r0     ; address of x[i] in r12
        STRB    r3, [r12,#0] ; x[i] = y[i]
```

Next, we exit the loop if the character was 0. That is, we exit if it is the last character of the string:

```
        BEQ     L2        ; if y[i] == 0, go to L2
```

If not, we increment i and loop back:

```
        ADD             r4, r4, #1 ; i = i + 1
        B               L1         ; go to L1
```

If we don't loop back, it was the last character of the string; we restore r4 and the stack pointer, and then return.

```
  L2: LDR     r4, [sp,#0]     ; y[i] == 0: end of string.
                                  Restore old r4
        ADD     sp, sp, #4    ; pop 1 word off stack
        MOV     pc, lr        ; return
```

String copies usually use pointers instead of arrays in C to avoid the operations on i in the code above. Section 2.10 shows more sophisticated addressing options for loads and stores that would reduce the number of instructions in this loop. Also, see Section 2.14 for an explanation of arrays versus pointers.

Since the procedure strcpy above is a leaf procedure, the compiler could allocate i to a temporary register and avoid saving and restoring it. Hence, instead of thinking of the registers r0 to r3 as being just for arguments, we can think of them as registers that the callee should use whenever convenient. When a compiler finds a leaf procedure, it exhausts such registers before using registers it must save.

Characters and Strings in Java

Unicode is a universal encoding of the alphabets of most human languages. Figure 2.16 is a list of Unicode alphabets; there are almost as many *alphabets* in Unicode as there are useful *symbols* in ASCII. To be more inclusive, Java uses Unicode for characters. By default, it uses 16 bits to represent a character.

The ARM instruction set has explicit instructions to load and store such 16-bit quantities, called *halfwords.* Load register half (LDRH) loads a halfword from memory, placing it in the rightmost 16 bits of a register. Like load register byte, *load half* (LDRH) treats the halfword as a unsigned number and thus zero-extends to fill the 16 leftmost bits of the register, while *load register signed halfword* (LDRSH) works with signed integers. Thus, LDRH is the more popular of the two. Store half (STRH) takes a halfword from the rightmost 16 bits of a register and writes it to memory. We copy a halfword with the sequence

```
LDRH r0,[sp,#0]     ; Read halfword (16 bits) from source
STRH r0,[r12,#0]    ; Write halfword (16 bits) to destination
```

Strings are a standard Java class with special built-in support and predefined methods for concatenation, comparison, and conversion. Unlike C, Java includes a word that gives the length of the string, similar to Java arrays.

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

FIGURE 2.16   Example alphabets in Unicode. Unicode version 4.0 has more than 160 "blocks," which is their name for a collection of symbols. Each block is a multiple of 16. For example, Greek starts at 0370_{hex}, and Cyrillic at 0400_{hex}. The first three columns show 48 blocks that correspond to human languages in roughly Unicode numerical order. The last column has 16 blocks that are multilingual and are not in order. A 16-bit encoding, called UTF-16, is the default. A variable-length encoding, called UTF-8, keeps the ASCII subset as eight bits and uses 16–32 bits for the other characters. UTF-32 uses 32 bits per character. To learn more, see *www.unicode.org*.

Elaboration: ARM software tries to keep the stack aligned to word addresses, allowing the program to always use `LDR` and `STR` (which must be aligned) to access the stack. This convention means that a `char` variable allocated on the stack occupies 4 bytes, even though it needs less. However, a C string variable or an array of bytes *will* pack 4 bytes per word, and a Java string variable or array of shorts packs 2 halfwords per word.

I. Which of the following statements about characters and strings in C and Java are true?

1. A string in C takes about half the memory as the same string in Java.

2. Strings are just an informal name for single-dimension arrays of characters in C and Java.

3. Strings in C and Java use null (0) to mark the end of a string.

4. Operations on strings, like length, are faster in C than in Java.

II. Which type of variable that can contain $1,000,000,000_{ten}$ takes the most memory space?

1. `int` in C

2. `string` in C

3. `string` in Java

2.10 ARM Addressing for 32-Bit Immediates and More Complex Addressing Modes

Although keeping all ARM instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have a 32-bit constant or 32-bit address. This section starts with the how ARM builds ins support for certain 32-bit patterns from just 12 bits, and then shows the optimizations for addresses used in data transfers.

32-Bit Immediate Operands

Although constants are frequently short and fit into a narrow field, sometimes they are bigger. The ARM architects believed that some 32-bit constants were more popular than others, and so included a trick to allow ARM to specify some that they thought would be important. The 12-bit Operand2 field in the DP format actually is subdivided into 2 fields: a 8-bit constant field on the right and a 4-bit rotate right field. This latter field rotates the 8-bit constant to the right by twice the value in the rotate field.

Using unsigned numbers, this trick can represent any number such that

$$X * 2^{2i}$$

where X is between 0 and 255 and i is between 0 and 15. It can also represent a few other patterns when the 8-bit rotated constant straddles both the most significant and least significant bits:

$$T * 2^{30} + W, \ U * 2^{28} + V, \text{ and } W * 2^{26} + T$$

where T is between 0 and 3, U is 0 to 15, V is 0 to 15, and W is 0 and 63. Since ARM has the MVN instruction, it is also quick to produce the one's complement of any of the patterns above.

EXAMPLE

Loading a 32-Bit Constant

What is the ARM machine code to load this 32-bit constant into register `r0`?

```
0000 0000 1101 1001 0000 0000 0000 0000
```

ANSWER

First, we would find the pattern for the 8 non-zerobits (`1101 1001`) which is 217 in decimal. To move that pattern from the 8 least significant bits to bits 23 to 16, we need to rotate the 8-bit pattern to the right by 16 bits. Since ARM doubles the amount in the rotation field, we need to enter 8 in that field. The machine language MOV instruction (opcode=13) to put this 32-bit value in `r4` is:

Cond	F	I	Opcode	S	Rn	Rd	rotate-imm	mm_8
14	0	1	13	0	0	4	8	217
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	4 bits	8 bits

Hardware/ Software Interface

The assembler should map large constants into the right instruction format so that the programmer need not worry about it. If it doesn't match one of these patterns, the assembler should place the 32-bit constant in memory and then load it into the register. This useful feature can be generalized so that the symbolic representation of the ARM machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include (see Section 2.12).

Elaboration: ARMv6T2 added instructions that allow creation of any 32-bit constant in just two instructions. MOVT writes a 16-bit immediate into the top half of the destination register without affecting the bottom half. MOVW does the opposite: Writes the bottom half without affecting the top half. The ARM assembler picks the most efficient instruction sequence depending on the constant.

More Complex Addressing in Data Transfer Instructions

Multiple forms of addressing are generically called **addressing modes**. Consistent with the ARM philosophy of trying to reduce the number of instructions to execute a program—as exemplified by the shifting of arithmetic operands, conditional instruction execution, and rotating immediates—are including many addressing modes for data processing instructions.

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

The addressing mode in Section 2.6 is called *immediate offset*, where a constant address is added to a base register. There are seven more:

1. *Register Offset*. Instead of adding a constant to the base register, another register is added to the base register. This mode can help with an index into an array, where the array index is in one register and the base of the array is in another. Example: `LDR r2, [r0,r1]`.

2. *Scaled Register Offset*. Just as the second operand can be optionally shifted left or right in data processing instructions, this addressing mode allows the register to be shifted *before* it is added to the base register. This mode can be useful to turn an array index into a byte address by shifting it left by 2 bits. We'll see this mode used in Section 2.14. Example: `LDR r2, [r0,r1, LSL #2]`

3. *Immediate Pre-Indexed*. This and the following modes update the base register with the new address as part of the addressing mode. That is, on a load instruction, the content of the destination register changes based on the value fetched from memory and the base register changes to the address that was used to access memory. This mode can be useful when going sequentially through an array. There are two versions—one where the address is added to the base and one where the address is subtracted from the base—to allow the programmer to go through the array forwards or backwards. Note that in this mode the addition or subtraction occurs before the address is sent to memory. Example: `LDR r2, [r0, #4]!`

4. *Immediate Post-Indexed*. Just to cover all eventualities, ARM also has a mode just like immediate pre-indexed except the address in the base register is used to access memory first and then the constant is added or subtracted later. Depending on how you set up your program to access memory, either pre-indexed or post-indexed is desired, but not likely both. We'll see this mode used in Section 2.14. Example: `LDR r2, [r0], #4`

5. *Register Pre-Indexed*. The same Immediate Pre-Indexed, except you add or subtract a register instead of a constant. Example: `LDR r2, [r0,r1]!`

6. *Scaled Register Pre-Indexed*. The same Register Pre-Indexed, except you shift the register before adding or subtracting it. Example: `LDR r2, [r0,r1, LSL #2]!`

7. *Register Post-Indexed*. The same Immediate Post-Indexed, except you add or subtract a register instead of a constant. Example: `LDR r2, [r0],r1`

Some consider the operands in the data processing instructions to be addressing modes as well. Hence, we can add the following modes to the list:

1. *Immediate addressing,* where the operand is a constant within the instruction itself. Example: `ADD   r2, r0, #5`

2. *Register addressing,* where the operand is a register. Example: `ADD r2, r0, r1`

3. *Scaled register addressing,* where the register operand is shifted first. Example: `ADD r2, r0, r1, LSL #2`

4. *PC-relative addressing,* where the branch address is the sum of the PC and a constant in the instruction. Example: BEQ 1000

Since one of the 16 ARM registers is the program counter, the data transfer addressing modes can also offer PC-relative addressing, just as we saw for branches. One reason for using PC-relative addressing with loads would be to fetch 32-bit constants that could be placed in memory along with the program.

Figure 2.17 shows how operands are identified for each addressing mode. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate and register addressing.

Figure 2.18 shows the ARM instruction formats covered so far. Figure 2.1 on page 78 shows the ARM assembly language revealed in this chapter. The remaining portion of ARM instructions deals mainly with arithmetic and real numbers, which are covered in the next chapter.
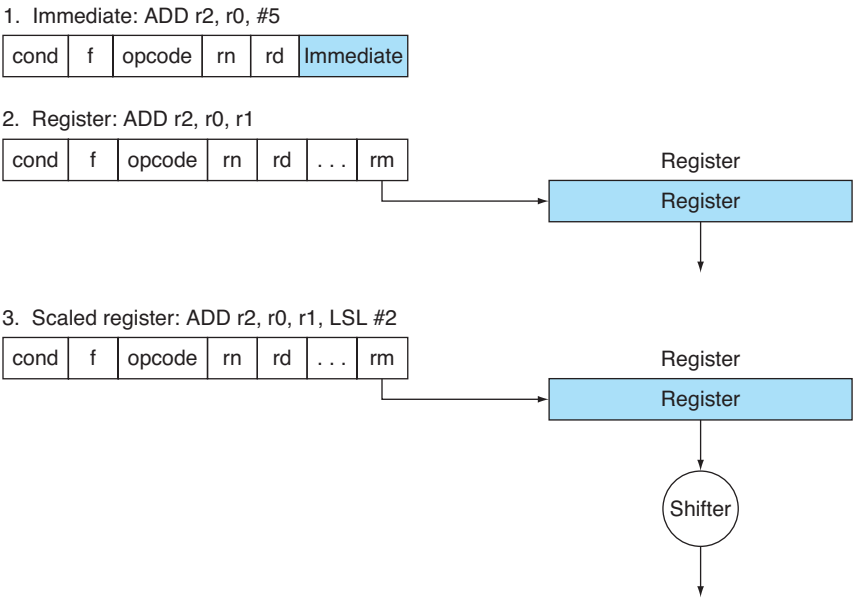
FIGURE 2.17   Illustration of the twelve ARM addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is in the instruction itself.

4. PC-relative: BEQ 1000
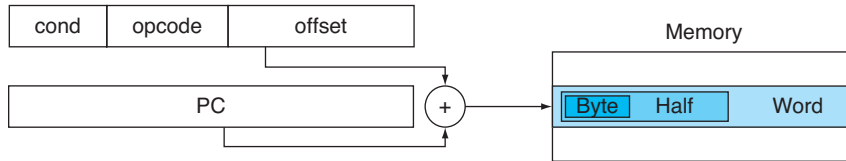
5. Immediate offset: LDR r2, [r0, #8]
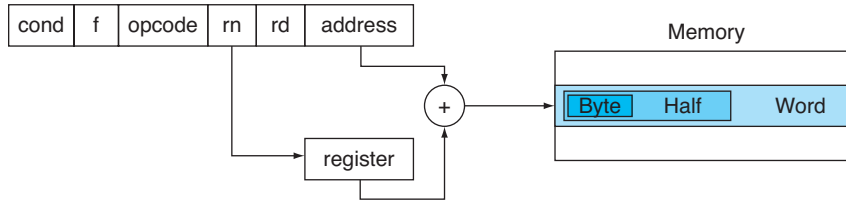
6. Register offset: LDR r2, [r0, r1]
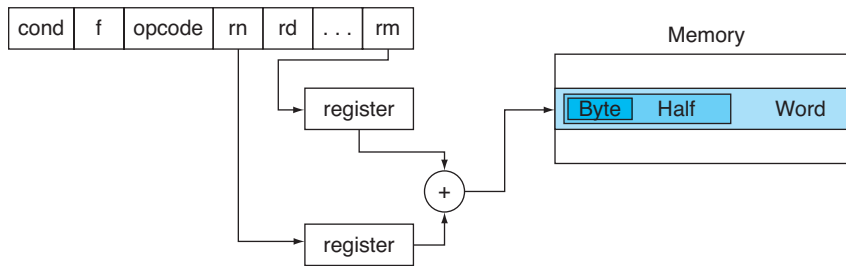
7. Scaled register offset: LDR r2, [r0, r1, LSL #2]
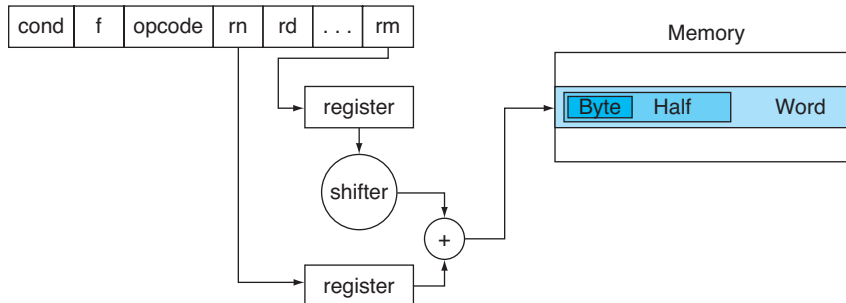
8. Immediate offset pre-indexed: LDR r2, [r0, #4]!
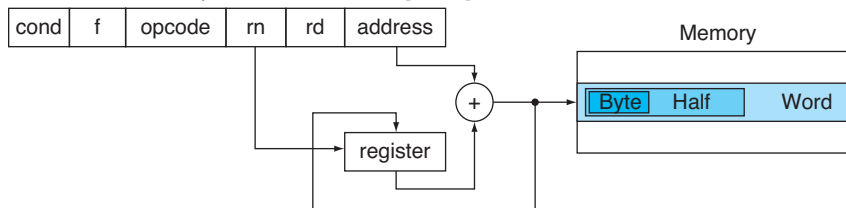
FIGURE 2.17   Illustration of the twelve ARM addressing modes. (*continued*)

9. Immediate offset post-indexed: LDR r2, [r0], #4
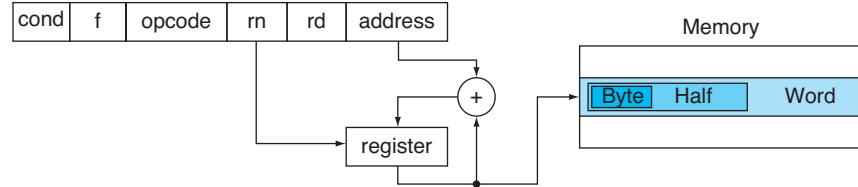
10. Register offset pre-indexed: LDR r2, [r0, r1]!
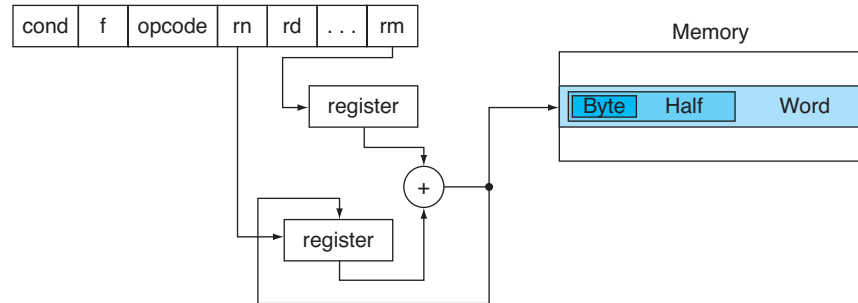
11. Scaled register offset pre-indexed: LDR r2, [r0, r1, LSL #2]!
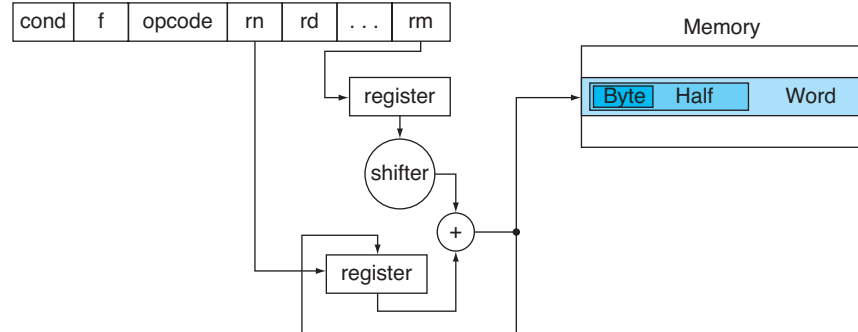
12. Register offset post-indexed: LDR r2, [r0], r1
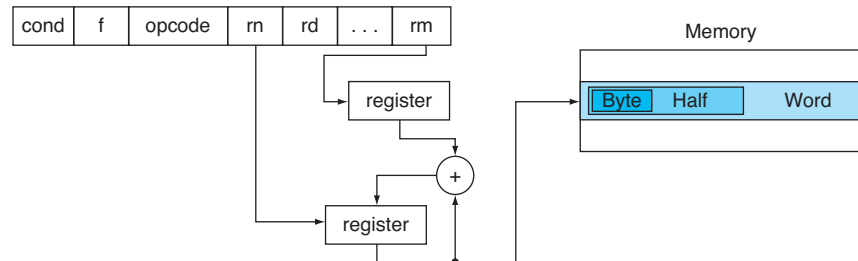
FIGURE 2.17   Illustration of the twelve ARM addressing modes. (*continued*)

Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F		Opcode		Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

FIGURE 2.18 ARM instruction formats.

Although ARM has 32-bit addresses, many microprocessors also have 64-bit address extensions in addition to 32-bit addresses. These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in such a way that is able to move software compatibly upward to the next generation of architecture.

Hardware/ Software Interface

2.11 Parallelism and Instructions: Synchronization

Parallel execution is easier when tasks are independent, but often they need to cooperate. Cooperation usually means some tasks are writing new values that others must read. To know when a task is finished writing so that it is safe for another to read, the tasks need to synchronize. If they don't synchronize, there is a danger of a **data race**, where the results of the program can change depending on how events happen to occur.

For example, recall the analogy of the eight reporters writing a story on page 43 of Chapter 1. Suppose one reporter needs to read all the prior sections before writing a conclusion. Hence, he must know when the other reporters have finished their sections, so that he or she need not worry about them being changed afterwards. That is, they had better synchronize the writing and reading of each section so that the conclusion will be consistent with what is printed in the prior sections.

In computing, synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. In this section, we focus on the implementation of *lock* and *unlock* synchronization operations. Lock and unlock can be used straightforwardly to create regions where only a single processor can operate, called *mutual exclusion*, as well as to implement more complex synchronization mechanisms.

The critical ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to *atomically* read and modify a memory

data race Two memory accesses form a data race if they are from different threads to same location, at least one is a write, and they occur one after another.

location. That is, nothing else can interpose itself between the read and the write of the memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky.

Let's start with one such hardware primitive and show how it can be used to build a basic synchronization primitive. One typical operation for building synchronization operations is the *atomic exchange* or *atomic swap*, which interchanges a value in a register for a value in memory. ARM has such an instruction: SWP.

To see how to use this to build a basic synchronization primitive, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to 1, preventing any competing exchange in another processor from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: this race is broken, since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the hardware. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

Elaboration: Although it was presented for multiprocessor synchronization, atomic exchange is also useful for the operating system in dealing with multiple processes in a single processor. To make sure nothing interferes in a single processor, the store conditional also fails if the processor does a context switch between the two instructions (see Chapter 5).

Check Yourself

When do you use primitives like SWP?

1. When cooperating threads of a parallel program need to synchronize to get proper behaviour for reading and writing shared data

2. When cooperating processes on a uniprocessor need to synchronize for reading and writing shared data

2.12 Translating and Starting a Program

This section describes the four steps in transforming a C program in a file on disk into a program running on a computer. Figure 2.19 shows the translation hierarchy. Some systems combine these steps to reduce translation time, but these are the logical four phases that programs go through. This section follows this translation hierarchy.

Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level language programs take
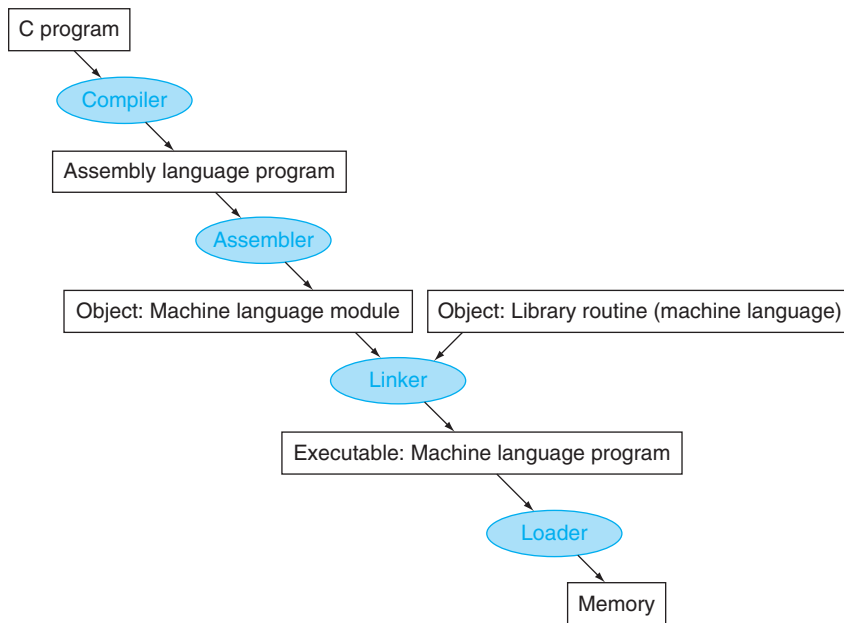
FIGURE 2.19   A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable files by default are called a.out. MS-DOS uses the suffixes .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

assembly language
A symbolic language that can be translated into binary machine language.

many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in **assembly language** because memories were small and compilers were inefficient. The 500,000-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as good as an assembly language expert, and sometimes even better for large programs.

Assembler

Since assembly language is an interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called **pseudoinstructions**.

pseudoinstruction
A common variation of assembly language instructions often treated as if it were an instruction in its own right.

For example, the ARM assembler accepts this instruction:

```
LDR r0, #constant
```

and the assembler determines which instructions to use to create the constant in the most efficient way possible. The default is loading a 32-bit constant from memory.

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. ARM assemblers use hexadecimal.

Such features are convenient, but the primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a **symbol table**. As you might expect, the table contains pairs of symbols and addresses.

symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.

- The *text segment* contains the machine language code.

- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both *static data,* which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program. See Figure 2.13.)

- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.

■ The *symbol table* contains the remaining labels that are not defined, such as external references.

■ The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines, because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a **link editor** or **linker**, which takes all the independently assembled machine language programs and "stitches" them together.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.

2. Determine the addresses of data and instruction labels.

3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions, jump instructions, and data addresses, so the job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses. Editing is the origin of the name "link editor," or linker for short. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that Figure 2.13 on page 120 shows the ARM convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data would be placed relative to other modules. When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.

The linker produces an **executable file** that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, that still have unresolved addresses and hence result in object files.

linker Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

executable file A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A "stripped executable" does not contain that information. Relocation information may be included for the loader.

EXAMPLE

Linking Object Files

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures A and B and the instructions that refer to the addresses of data words X and Y.

Object file header			
	Name	Procedure A	
	Text size	100$_{hex}$	
	Data size	20$_{hex}$	
Text segment	Address	Instruction	
	0	LDR r0, 0(r3)	
	4	BL 0	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	LDR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200$_{hex}$	
	Data size	30$_{hex}$	
Text segment	Address	Instruction	
	0	STR r1, 0(r3)	
	4	BL 0	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	STR	Y
	4	BL	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Procedure A needs to find the address for the variable labelled X to put in the load instruction and to find the address of procedure B to place in the BL instruction. Procedure B needs the address of the variable labelled Y for the store instruction and the address of procedure A for its BL instruction.

From Figure 2.13 on page 120, we know that the text segment starts at address $40\ 0000_{hex}$ and the data segment at $1000\ 0000_{hex}$. The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100_{hex} bytes and its data is 20_{hex} bytes, so the starting address for procedure B text is $40\ 0100_{hex}$, and its data starts at $1000\ 0020_{hex}$.

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	LDR r0, 8000_{hex}(r3)
	$0040\ 0004_{hex}$	BL 00 $00EC_{hex}$
	...	...
	$0040\ 0100_{hex}$	STR r1, 8020_{hex}(r3)
	$0040\ 0104_{hex}$	BL FF $FDDD_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have two types here:

1. The BLs use PC-relative addressing. The BL at address $40\ 0004_{hex}$ gets $40\ 0100_{hex}$ (the address of procedure B) - ($40\ 0004_{hex}$ + 8) = $00\ 00EC_{hex}$ in its address field. The BL at $40\ 0104_{hex}$ needs $40\ 0000_{hex}$ (the address of procedure A) – ($40\ 0104_{hex}$ + 8) = -112_{hex}. The two's complement representation is FF $FDDD_{hex}$.

2. The load and store addresses are harder because they are relative to a base register. This example uses the global pointer as the base register. Assume that register r4 is initialized to $1000\ 8000_{hex}$. To get the address $1000\ 0000_{hex}$ (the address of word X), we place 8000_{hex} in the address field of LDR at address $40\ 0000_{hex}$. Similarly, we place 8020_{hex} in the address field of STR at address $40\ 0100_{hex}$ to get the address $1000\ 0020_{hex}$ (the address of word Y).

Elaboration: Recall that ARM instructions are word aligned, so `BL` drops the right two bits to increase the instruction's address range. Thus, it uses 24 bits to create a 26-bit byte address. Hence, the actual address in the lower 24 bits of the `BL` instruction in this example is $00\ 002B_{hex}$, rather than $00\ 00EC_{hex}$.

Loader

loader A systems program that places an object program in main memory so that it is ready to execute.

Now that the executable file is on disk, the operating system reads it to memory and starts it. The **loader** follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.

2. Creates an address space large enough for the text and data.

3. Copies the instructions and data from the executable file into memory.

4. Copies the parameters (if any) to the main program onto the stack.

5. Initializes the machine registers and sets the stack pointer to the first free location.

6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

Dynamically Linked Libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

■ The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.

■ It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

dynamically linked libraries (DLLs) Library routines that are linked to a program during execution.

These disadvantages lead to **dynamically linked libraries (DLLs)**, where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the initial version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus only those that are called during

the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

Like many innovations in our field, this trick relies on a level of indirection. Figure 2.20 shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump.

The first time the library routine is called, the program calls the dummy entry and follows the indirect jump. It points to code that puts a number in a register to identify the desired library routine and then jumps to the dynamic linker/loader. The linker/loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine. It then jumps to it. When the

a. First call to DLL routine   b. Subsequent calls to DLL routine
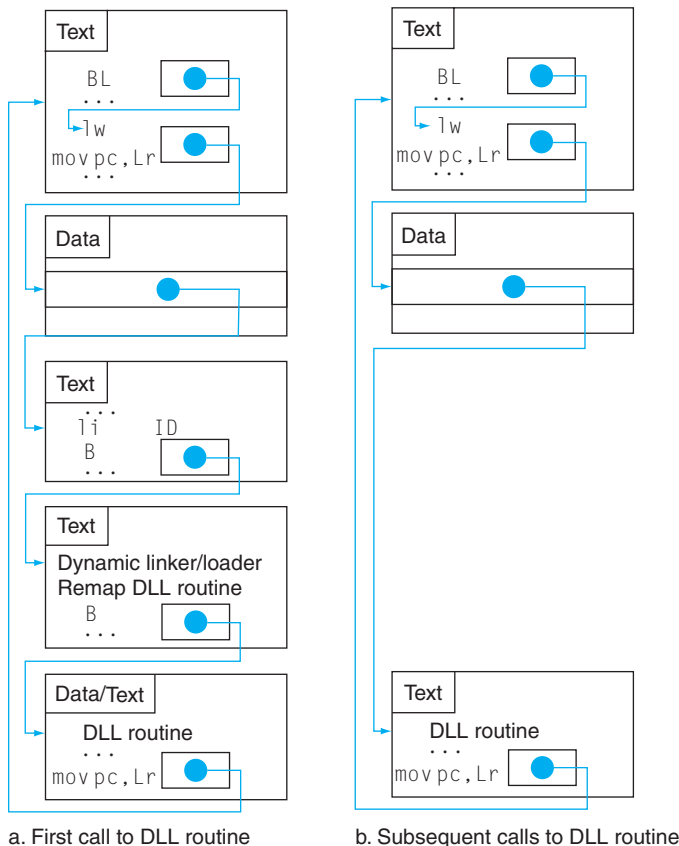
FIGURE 2.20 Dynamically linked library via lazy procedure linkage. (a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in Chapter 5, the operating system may avoid copying the desired routine by remapping it using virtual memory management.

routine completes, it returns to the original calling site. Thereafter, the call to the library routine jumps indirectly to the routine without the extra hops.

In summary, DLLs require extra space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect jump thereafter. Note that the return from the library pays no extra overhead. Microsoft's Windows relies extensively on dynamically linked libraries, and it is also the default when executing programs on UNIX systems today.

Starting a Java Program

The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a specific implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to run safely on any computer, even if it might slow execution time.

Figure 2.21 shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the **Java bytecode** instruction set (see ⊙ **Section 2.15** on the CD). This instruction set is designed to be close to the Java language so that this compilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

Java bytecode
Instruction from an instruction set designed to interpret Java programs.
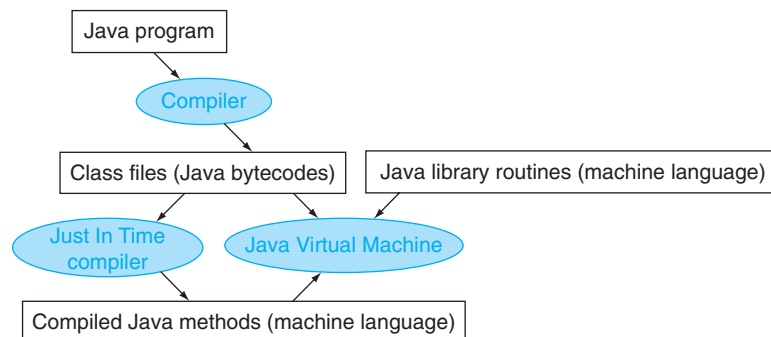
FIGURE 2.21   A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

A software interpreter, called a **Java Virtual Machine** (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the ARM simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most people could write and run Java programs shortly after Java was announced. Today, Java virtual machines are found in hundreds of millions of devices, in everything from cell phones to Internet browsers.

The downside of interpretation is lower performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java development was compilers that translated *while* the program was running. Such **Just In Time compilers (JIT)** typically profile the running program to find where the "hot" methods are and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

As computers get faster so that compilers can do more, and as researchers invent betters ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. ◎ **Section 2.15** on the CD goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

Which of the advantages of an interpreter over a translator do you think was most important for the designers of Java?

1. Ease of writing an interpreter

2. Better error messages

3. Smaller object code

4. Machine independence

Java Virtual Machine (JVM) The program that interprets Java bytecodes.

Just In Time compiler (JIT) The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

Check Yourself

2.13   A C Sort Example to Put It All Together

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the ARM code from two procedures written in C: one to swap array elements and one to sort them.

```
void swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

FIGURE 2.22   A C procedure that swaps two locations in memory. This subsection uses this procedure in a sorting example.

The Procedure swap

Let's start with the code for the procedure swap in Figure 2.22. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.

2. Produce code for the body of the procedure.

3. Preserve registers across the procedure invocation.

This section describes the swap procedure in these three pieces, concluding by putting all the pieces together.

Register Allocation for swap

As mentioned on pages 113–114, the ARM convention on parameter passing is to use registers r0, r1, r2, and r3. Since swap has just two parameters, v and k, they will be found in registers r0 and r1. The only other variable is temp, which we associate with register r2 since swap is a leaf procedure (see page 116–117). This register allocation corresponds to the variable declarations in the first part of the swap procedure in Figure 2.22. We'll also need two registers to hold temporary calculations. We'll use r3 and r12, since the caller does not expect them to be preserved across a procedure call. To make to assembly language easier to read, we'll use the assembler directive that let's us associate names with registers.

```
v       RN 0  ; 1st argument address of v
k       RN 1  ; 2nd argument index k
temp    RN 2  ; local variable
temp2   RN 3  ; temporary for v[k+1]
vkAddr  RN 12 ; to hold address of v[k]
```

Code for the Body of the Procedure swap

The remaining lines of C code in swap are

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Recall that the memory address for ARM refers to the *byte* address, and so words are really 4 bytes apart. Hence we need to multiply the index k by 4 before adding it to the address. *Forgetting that sequential word addresses differ by 4 instead of by 1 is a common mistake in assembly language programming.* Hence the first step is to get the address of v[k] by multiplying k by 4 via a shift left by 2:

```
ADD    vkAddr, v, k, LSL #2; reg vkAddr = v + (k * 4)
               ; reg vkAddr has the address of v[k]
```

Now we load v[k] using vkAddr, and then v[k+1] by adding 4 to vkAddr:

```
LDR    temp, [vkAddr, #0]  ; temp = v[k]
LDR    temp2, [vkAddr, #4] ; temp2 = v[k + 1]
                           ; refers to next element of v
```

Next we store temp and temp2 to the swapped addresses:

```
STR    temp2, [vkAddr, #0] ; v[k] = temp2
STR    temp, [vkAddr, #4]  ; v[k+1] = temp
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within swap. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The Full swap Procedure

We are now ready for the whole routine, which includes the procedure label and the return jump. To make it easier to follow, we identify in Figure 2.23 each block of code with its purpose in the procedure.

Procedure body
| ```
swap: ADD     vkAddr, v, k, LSL #2  ; reg vkAddr = v + (k * 4)
                                    ; reg vkAddr has the address of v[k]
      LDR     temp, [vkAddr, #0]    ; temp (temp) = v[k]
      LDR     temp2, [vkAddr, #4]   ; temp2 = v[k + 1]
                                    ; refers to next element of v
      STR     temp2, [vkAddr, #0]   ; v[k] = temp2
      STR     temp, [vkAddr, #4]    ; v[k+1] = temp
``` |

Procedure return
| ```
MOV    pc, lr                      ; return to calling routine
``` |

FIGURE 2.23   ARM assembly code of the procedure *swap* in Figure 2.22.

The Procedure sort

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the

swap procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. Figure 2.24 shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

FIGURE 2.24   A C procedure that performs a sort on the array v.

Register Allocation for sort

The two parameters of the procedure sort, v and n, are in the parameter registers r0 and r1, and we assign register r2 to local variable i and register r3 to j. We'll also need five more registers to hold values, which we'll assign to r12 and r4 to r7.

To enhance code legibility, we'll tell the assembler to rename the registers:

```
v         RN 0  ; 1st argument address of v
n         RN 1  ; 2nd argument index n
i         RN 2  ; local variable i
j         RN 3  ; local variable j
vjAddr    RN 12 ; to hold address of v[j]
vj        RN 4  ; to hold a copy of v[j]
vj1       RN 5  ; to hold a copy of v[j+1]
vcopy     RN 6  ; to hold a copy of v
ncopy     Rn 7  ; to hold a copy of n
```

Code for the Body of the Procedure sort

The procedure body consists of two nested *for* loops and a call to swap that includes parameters. Let's unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize i to 0, the first part of the *for* statement:

```
MOV    i, #0     ; i = 0
```

(Remember that `move` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer; see page 136.) It also takes just one instruction to increment i, the last part of the *for* statement:

```
ADD    i, i, #1         ; i += 1
```

The loop should be exited if i < n is *not* true or, said another way, should be exited if i ≥ n. This test takes two instructions:

```
for1tst:CMP i, n          ; if i ≥ n
     BGE   exit1          ; go to exit1 if i ≥ n
```

The bottom of the loop just jumps back to the loop test:

```
        B  for1tst       ; branch to test of outer loop
    exit1:
```

The skeleton code of the first *for* loop is then

```
        MOV  i, #0          ; i = 0
 for1tst:CMP i, n           ; if i ≥ n
        BGE  exit1          ; go to exit1 if i ≥ n
            ...
            (body of first for loop)
            ...
        ADD  i, i, #1       ; i += 1
        B    for1tst        ; branch to test of outer loop
  exit1:
```

Voila! (The exercises explore writing faster code for similar loops.)
   The second *for* loop looks like this in C:

```
    for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
    SUB    j, i, #1 ; j = i - 1
```

The decrement of j at the end of the loop is also one instruction:

```
    SUB    j, j, #1 ; j -= 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (j < 0):

```
    for2tst: CMP j, #0        ; if j < 0
             BLT  exit2       ; go to exit2 if j < 0
```

This branch will skip over the second condition test. If it doesn't skip, j ≥ 0.

The second test exits if v[j] > v[j + 1] is *not* true, or exits if v[j] ≤ v[j + 1]. First we create the address by multiplying j by 4 (since we need a byte address) and add it to the base address of v:

```
ADD    vjAddr, v, j, LSL #2 ; reg vjAddr = v + (j * 4)
```

Now we load v[j]:

```
LDR    vj, [vjAddr,s#0]     ; reg vj  = v[j]
```

Since we know that the second element is just the following word, we add 4 to the address in register vjAddr to get v[j + 1]:

```
LDR    vj1, [vjAddr,#4]     ; reg vj1  = v[j + 1]
```

The test of v[j] ≤ v[j + 1] is next, so the two instructions of the exit test are

```
CMP    vj, vj1              ; if vj ≤ vj1
BLE    exit2                ; go to exit2 if vj ≤ vj1
```

The bottom of the loop jumps back to the inner loop test:

```
B     for2tst ; branch to test of inner loop
```

Combining the pieces, the skeleton of the second *for* loop looks like this:

```
        SUB  j, i,#1                ; j = i - 1
for2tst:CMP  j, #0                  ; if j < 0
        BLT  exit2                  ; go to exit2 if j < 0
        ADD  vjAddr, v, j, LSL #2   ; reg vjAddr = v + (j * 4)
        LDR  vj, [vjAddr,#0]        ; reg vj   = v[j]
        LDR  vj1, [vjAddr,#4]       ; reg vj1  = v[j + 1]
        CMP  vj, vj1               ; if vj ≤ vj1
        BLE  exit2                  ; go to exit2 if vj ≤ vj1
            ...
            (body of second for loop)
            ...
        SUB  j, j, #1              ; j -= 1
        B    for2tst              ; branch to test of inner loop
    exit2:
```

The Procedure Call in sort

The next step is the body of the second *for* loop:

```
swap(v,j);
```

Calling swap is easy enough:

```
BL        swap
```

Passing Parameters in sort

The problem comes when we want to pass parameters because the sort procedure needs the values in registers v and n, yet the swap procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for sort into other registers earlier in the procedure, making registers v and n available for the call of swap. We first copy v and n into vcopy and ncopy during the procedure:

```
MOV    vcopy, v  ; copy parameter v into vcopy (save r0)
MOV    ncopy, n  ; copy parameter n into ncopy (save r1)
```

Then we pass the parameters to swap with these two instructions:

```
MOV    r0, vcopy ; first swap parameter is v
MOV    r1, j     ; second swap parameter is j
```

Preserving Registers in sort

The only remaining code is the saving and restoring of registers. Clearly, we must save the return address in register lr, since sort is a procedure and is called itself. The sort procedure also uses the saved registers i, j, vcopy, and ncopy, so they must be saved. The prologue of the sort procedure is then

```
SUB    sp,sp,#20     ; make room on stack for 5 registers
STR    lr, [sp, #16] ; save lr on stack
STR    ncopy, [sp, #12] ; save ncopy on stack
STR    vcopy, [sp, #8] ; save vcopy on stack
STR    j, [sp, #4]   ; save j on stack
STR    i, [sp, #0]   ; save i on stack
```

The tail of the procedure simply reverses all these instructions, then adds a MOV pc,lr to return.

The Full Procedure sort

Now we put all the pieces together in Figure 2.25. Once again, to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, nine lines of the sort procedure in C became 32 lines in the ARM assembly language.

Elaboration: One optimization that works with this example is *procedure inlining*. Instead of passing arguments in parameters and invoking the code with a BL instruction, the compiler would copy the code from the body of the swap procedure where the call to swap appears in the code. Inlining would avoid four instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into *lower* performance if it increased the cache miss rate; see Chapter 5.

Saving registers
| ```
sort:   SUB   sp,sp,#20              ; make room on stack for 5 registers
        STR   lr, [sp, #16]          ; save lr on stack
        STR   ncopy, [sp, #12]       ; save ncopy on stack
        STR   vcopy,  [sp, #8]       ; save vcopy on stack
        STR   j,  [sp, #4]           ; save j on stack
        STR   i,  [sp, #0]           ; save i on stack
``` |

Procedure body	
| Move parameters | ```
MOV   vcopy, v      ; copy parameter v into vcopy (save r0)
MOV   ncopy, n      ; copy parameter n into ncopy (save r1)
``` |
| Outer loop | ```
        MOV   i, #0       ; i = 0
for1tst: CMP    i, n       ; if i ≥ n
        BGE   exit1      ; go to exit1 if  i ≥ n
``` |
| Inner loop | ```
        SUB   j, i, #1       ; j = i - 1
for2tst: CMP   j, #0         ; if j < 0
        BLT   exit2 ; go to exit2 if j < 0
        ADD   vjAddr, v, j, LSL #2      ; reg vjAddr = v + (j * 4)
        LDR   vj, [vjAddr,#0]           ; reg vj  = v[j]
        LDR   vj1, [vjAddr,#4]          ; reg vj1  = v[j + 1]
        CMP   vj, vj1        ; if vj ≤ vj1
        BLE   exit2          ; go to exit2 if vj ≤ vj1
``` |
| Pass parameters and call | ```
MOV   r0, vcopy      ; first swap parameter is v
MOV   r1, j          ; second swap parameter is j
BL    swap           ; swap code shown in Figure 2.23
``` |
| Inner loop | ```
SUB   j, j, #1       ; j -= 1
B     for2tst        ; branch to test of inner loop
``` |
| Outer loop | ```
exit2:  ADD   i, i, #1       ; i += 1
        B     for1tst        ; branch to test of outer loop
``` |

Restoring registers
| ```
exit1:  LDR   i,  [sp, #0]       ; restore i from stack
        LDR   j,  [sp, #4]       ; restore j from stack
        LDR   vcopy,  [sp, #8]   ; restore vcopy from stack
        LDR   ncopy, [sp, #12]   ; restore ncopy from stack
        LDR   lr, [sp, #16]      ; restore lr from stack
        ADD   sp,sp,#20          ; restore stack pointer
``` |

Procedure return
| ```
        MOV   pc, lr                ; return to calling routine
``` |

FIGURE 2.25   ARM assembly version of procedure sort **in Figure 2.24.**

Elaboration: ARM includes instructions to save and restore registers at procedure call boundaries. Store Multiple (STM) and Load Multiple (LDM) can store and load up to 16 registers, which are specified by a bit mask in the instruction. Thus, the 5 stores and

the SUB could be replaced by the instruction STM sp!, {i, j, ncopy, vcopy, lr} since the stack pointer would be updated by the pre-index addressing mode.

Figure 2.26 shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI, and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

Figure 2.27 compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the JIT compiler makes Java 2.1 times *faster* than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (◎ **Section 2.15** on the CD gives more details on interpretation versus compilation of Java and the Java and ARM code for Bubble Sort.) The ratios aren't as close for Quicksort in Column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude a performance increases by when sorting 100,000 items. Even comparing interpreted Java in Column 5 to the C compiler at highest optimization in Column 4, Quicksort beats Bubble Sort by a factor of 50 (0.05×2468, or 123 times faster than the unoptimized C code versus 2.41 times faster).

Understanding Program Performance

Elaboration: The ARM compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement sp by 16 to make room for all four argument registers (16 bytes). One reason is that C provides a vararg option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare vararg, it copies the four argument registers onto the stack into the four reserved locations.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.26   Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

FIGURE 2.27   Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as Figure 2.26. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

2.14   Arrays versus Pointers

A challenge for any new C programmer is understanding pointers. Comparing assembly code that uses arrays and array indices to the assembly code that uses pointers offers insights about pointers. This section shows C and ARM assembly versions of two procedures to clear a sequence of words in memory: one using array indices and one using pointers. Figure 2.28 shows the two C procedures.

The purpose of this section is to show how pointers map into ARM instructions, and not to endorse a dated programming style. We'll see the impact of modern compiler optimization on these two procedures at the end of the section.

Array Version of Clear

Let's start with the array version, clear1, focusing on the body of the loop and ignoring the procedure linkage code. We assume that the two parameters array and size are found in the registers r0 and r1, and that i is allocated to register r2. We start by renaming registers.

```
array         RN 0     ; 1st argument address of array
n             RN 1     ; 2nd argument size (of array)
i             RN 2     ; local variable i
zero          RN 3     ; temporary to hold constant 0
```

The initialization of i, the first part of the *for* loop, is straightforward:

```
MOV    i,0      ; i = 0
```

We also need to put 0 into the temporary register so that we can write it to memory:

```
MOV    zero,0   ; zero = 0
```

```
clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i += 1)
      array[i] = 0;
}


clear2(int *array, int size)
{
  int *p;
  for (p = &array[0]; p < array[size]; p = p + 1)
      *p = 0;
}
```

FIGURE 2.28 Two C procedures for setting an array to all zeros. Clear1 uses indices, while clear2 uses pointers. The second procedure needs some explanation for those unfamiliar with C. The address of a variable is indicated by &, and the object pointed to by a pointer is indicated by *. The declarations declare that array and p are pointers to integers. The first part of the *for* loop in clear2 assigns the address of the first element of array to the pointer p. The second part of the *for* loop tests to see if the pointer is pointing beyond the last element of array. Incrementing a pointer by one, in the last part of the *for* loop, means moving the pointer to the next sequential object of its declared size. Since p is a pointer to integers, the compiler will generate ARM instructions to increment p by four, the number of bytes in a ARM integer. The assignment in the loop places 0 in the object pointed to by p.

To set array[i] to 0 we must first get its address. We can use scaled register offset addressing mode to multiply i by 4 to get the byte address and then add it to the index to get the address of array[i]:

```
    loop1:STR    zero, [array,i, LSL #2] ; array[i] = 0
```

This instruction is the end of the body of the loop, so the next step is to increment i:

```
        ADD    i,i,#1           ; i = i + 1
```

The loop test checks if i is less than size:

```
        CMP    i,size           ; i < size
        BLT    loop1            ; if (i < size) go to loop1
```

We have now seen all the pieces of the procedure. Here is the ARM code for clearing an array using indices:

```
        MOV    i,0              ; i = 0
        MOV    zero,0           ; zero = 0
  loop1:STR    zero, [array,i, LSL #2] ; array[i] = 0
        ADD    i,i,#1           ; i = i + 1
        CMP    i,size           ; i < size
        BLT    loop1            ; if (i < size) go to loop1
```

(This code works as long as `size` is greater than 0; ANSI C requires a test of size before the loop, but we'll skip that legality here.)

Pointer Version of Clear

The second procedure that uses pointers allocates the two parameters `array` and `size` to the registers `r0` and `r1` and allocates `p` to register `r2`, renaming the registers to do this:

```
array          RN 0    ; 1st argument address of array
n              RN 1    ; 2nd argument size (of array)
p              RN 2    ; local variable i
zero           RN 3    ; temporary to hold constant 0
arraySize      RN 12   ; address of array[size]
```

The code for the second procedure starts with assigning the pointer `p` to the address of the first element of the array and setting zero:

```
        MOV   p,array  ; p = address of array[0]
        MOV   zero,#0  ; zero = 0
```

The next code is the body of the *for* loop, which simply stores 0 into memory pointed to by `p`. We'll use the immediate post-indexed addressing mode to increment `p`. Incrementing a pointer by 1 means moving the pointer to the next sequential object in C. Since `p` is a pointer to integers, each of which uses 4 bytes, the compiler increments `p` by 4.

```
loop2: STR    zero,[p],#4     ; Memory[p] = 0; p = p + 4
```

The loop test is next. The first step is calculating the address of the last element of `array`. Start with multiplying `size` by 4 to get its byte address and then we add the product to the starting address of the array to get the address of the first word *after* the array:

```
        ADD  arraySize,array,size,LSL #2 ; arraySize = address
                                         ; of array[size]
```

The loop test is simply to see if `p` is less than the last element of `array`:

```
        CMP  p,arraySize   ; p < &array[size]
        BLT  loop2         ; if (p<&array[size]) go to loop2
```

With all the pieces completed, we can show a pointer version of the code to zero an array:

```
        MOV  p,array       ; p = address of array[0]
        MOV  zero,#0       ; zero = 0
```

```
loop2:STR zero,[p],#4      ; Memory[p] = 0; p = p + 4
      ADD  arraySize,array,size,LSL #2 ; arraySize = address
                                        ; of array[size]
      CMP  p,arraySize    ; p < &array[size]
      BLT  loop2          ; if (p<&array[size]) go to loop2
```

As in the first example, this code assumes size is greater than 0.

Note that this program calculates the address of the end of the array in every iteration of the loop, even though it does not change. A faster version of the code moves this calculation outside the loop:

```
      MOV    p,array        ; p = address of array[0]
      MOV    zero,#0        ; zero = 0
      ADD    arraySize,array,size,LSL #2 ; arraySize =
                             ; address of array[size]
  loop2:STR zero,[p],#4   ; Memory[p] = 0; p = p + 4
      CMP    p,arraySize   ; p < &array[size]
      BLT    loop2         ; if (p<&array[size]) go to loop2
```

Comparing the Two Versions of Clear

Comparing the two code sequences side by side illustrates the difference between array indices and pointers (the changes introduced by the pointer version are highlighted):

```
      MOV   i,#0     ; i = 0                              MOV   p,array    ; p = & array[0]
      MOV   zero,#0  ;zero = 0                            MOV   zero,#0    ;zero = 0
                                                          ADD   arraySize,array,size,LSL #2
loop1: STR zero, [array,i, LSL #2] ; array[i] = 0                          ; arraySize = &array[size]
      ADD   i,i,#1  ; i = i + 1                 loop2:    STR   zero,[p],#4  ; Memory[p] = 0,p = p + 4
      CMP   i,size  ; i < size                            CMP   p,arraySize; p<&array[size]
      BLT   loop1   ; if (i < size) go to loop1           BLT   loop2       ; if () go to loop2
```

The version on the left must have the "multiply" and add inside the loop because i is incremented and each address must be recalculated from the new index. However, the scaled register offset addressing mode hides that extra work. The memory pointer version on the right increments the pointer p directly via the post-indexed immediate addressing mode. The pointer version moves the end of array calculation outside the loop, thereby reducing the instructions executed per iteration from 4 to 3. This manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops). **Section 2.15** on the CD describes these two and many other optimizations.

Elaboration: As mentioned earlier, a C compiler would add a test to be sure that `size` is greater than 0. One way would be to add a jump just before the first instruction of the loop to the `CMP` instruction.

Understanding Program Performance

People used to be taught to use pointers in C to get greater efficiency than that available with arrays: "Use pointers, even if you can't understand the code." Modern optimizing compilers can produce code for the array version that is just as good. Most programmers today prefer that the compiler do the heavy lifting.

2.15 Advanced Material: Compiling C and Interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section is for readers interested in seeing how an **objected oriented language** like Java executes on a ARM architecture. It shows the Java bytecodes used for interpretation and the ARM code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java Virtual Machine and JIT compilers.

The rest of this section is on the CD.

object oriented language A programming language that is oriented around objects rather than actions, or data versus logic.

2.16 Real Stuff: MIPS Instructions

While not as popular as ARM, MIPS is an elegant instruction set that came out the same year as ARM and was inspired by similar philosophies. Figure 2.29 lists the similarities. The principle differences are that MIPS has more registers and ARM has more addressing modes.

There is a similar core of instruction sets for arithmetic-logical and data transfer instructions for MIPS and ARM, as Figure 2.30 shows.

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	26 (32 starting with ARMv3)	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

FIGURE 2.29   Similarities in ARM and MIPS instruction sets.

	Instruction name	ARM	MIPS
Register-register	Add	ADD	addu, addiu
	Add (trap if overflow)	ADDS; SWIVS	add
	Subtract	SUB	subu
	Subtract (trap if overflow)	SUBS; SWIVS	sub
	Multiply	MUL	mult, multu
	Divide	—	div, divu
	And	AND	and
	Or	ORR	or
	Xor	EOR	xor
	Load high part register	MOVT	lui
	Shift left logical	LSL[1]	sllv, sll
	Shift right logical	LSR[1]	srlv, srl
	Shift right arithmetic	ASR[1]	srav, sra
	Compare	CMP, CMN, TST, TEQ	slt/i, slt/iu
Data transfer	Load byte signed	LDRSB	lb
	Load byte unsigned	LDRB	lbu
	Load halfword signed	LDRSH	lh
	Load halfword unsigned	LDRH	lhu
	Load word	LDR	lw
	Store byte	STRB	sb
	Store halfword	STRH	sh
	Store word	STR	sw
	Read, write special registers	MRS, MSR	move
	Atomic Exchange	SWP, SWPB	ll;sc

FIGURE 2.30   ARM register-register and data transfer instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture or not synthesized in a few instructions. If there are several choices of instructions equivalent to the ARM core, they are separated by commas. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 1 are just a variation of a move instruction, such as LSR[1]. "Note that ARMv7M has divide instructions."

Data addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

FIGURE 2.31   Summary of data addressing modes in ARM vs. MIPS. MIPS has three basic addressing modes. The remaining six ARM addressing modes would require another instruction to calculate the address in MIPS.

Addressing Modes

Figure 2.31 shows the data addressing modes supported by ARM. MIPS has just three simple data addressing modes while ARM has nine, including fairly complex calculations. MIPS would require extra instructions to perform those address calculations.

Compare and Conditional Branch

Instead of using condition codes that are set as a side effect of an arithmetic or logical instruction code as in ARM, MIPS uses the contents of registers to evaluate conditional branches. Comparisons are done using the instruction Set on Less Than (slt) sets the contents of a register to 1 if the first operand is less than the second operand, and to 0 otherwise. The MIPS instructions Branch Equal (beq) and Branch Not Equal (bne) instructions compare the values of two registers before deciding to branch. The comparison to zero comes for free since on of MIPS 32 registers always has the value 0, called $zero.

Every ARM instruction has the option of executing conditionally, depending on the condition codes. The MIPS instructions do not have such a field.

Figure 2.32 shows the instruction formats for ARM and MIPS. The principal differences are the 4-bit conditional execution field in every instruction and the smaller register field, because ARM has half the number of registers.

The MIPS 16-bit immediate field is either zero extended to 32 bits or sign extended to 32 bits depending on the instruction, and doesn't have the unusual bit twidling features ARM's 12-bit immediate field.

Figure 2.33 shows a few of the ARM arithmetic-logical instructions not found in MIPS. Since ARM does not have a dedicated register for 0, it has separate opcodes to perform some operations that MIPS can do with $zero. MIPS instructions are generally a subset of the ARM instruction set, although MIPS does provide a Divide instruction (div). Figure 2.33.5 shows the MIPS instruction set used in Chapters 4 to 6.

FIGURE 2.32 Instruction formats for ARM and MIPS. The differences result from whether the architecture has 16 or 32 registers. and whether it has the 4-bit conditional execution field.

Name	Definition	ARM v.6	MIPS
Load immediate	Rd = Imm	MOV	addi, $0,
Not	Rd = ~(Rs1)	MVN	nor, $0,
Move	Rd = Rs1	MOV	or, $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ROR	
And not	Rd = Rs1 & ~(Rs2)	BIC	
Reverse subtract	Rd = Rs2 - Rs1	RSB, RSC	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	ADCS	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	SBCS	—

FIGURE 2.33 ARM arithmetic/logical instructions not found in MIPS.

MIPS operands

Name	Example	Comments

32 registers	`$s0-$s7, $t0-$t9, $zero,`
	`$a0-$a3, $v0-$v1, $gp, $fp,`
	`$sp, $ra, $at`
		Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.

2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add  $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub  $s1,$s2,$s3`	$s1 = $s2 − $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw  $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw  $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh  $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh  $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb  $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb  $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll  $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc  $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and  $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or   $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor  $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt  $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j   2500`	go to 10000	Jump to target address
	jump register	`jr  $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

MIPS assembly language.

2.17   Real Stuff: x86 Instructions

Designers of instruction sets sometimes provide more powerful operations than those found in ARM and MIPS. The goal is generally to reduce the number of instructions executed by a program. The danger is that this reduction can occur at the cost of simplicity, increasing the time a program takes to execute because the instructions are slower. This slowness may be the result of a slower clock cycle time or of requiring more clock cycles than a simpler sequence.

The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions. Section 2.18 demonstrates the pitfalls of complexity.

Evolution of the Intel x86

ARM and MIPS were the vision of single small groups in 1985; the pieces of these architectures fit nicely together, and the whole architecture can be described succinctly. Such is not the case for the x86; it is the product of several independent groups who evolved the architecture over 30 years, adding new features to the original instruction set as someone might add clothing to a packed bag. Here are important x86 milestones.

- **1978**: The Intel 8086 architecture was announced as an assembly language–compatible extension of the then successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Unlike ARM and MIPS, the registers have dedicated uses, and hence the 8086 is not considered a **general-purpose register** architecture.

- **1980**: The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Instead of using registers, it relies on a stack (see 🔘 **Section 2.20** and Section 3.7).

- **1982**: The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory-mapping and protection model (see Chapter 5), and by adding a few instructions to round out the instruction set and to manipulate the protection model.

- **1985**: The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 5). Like the 80286, the 80386 has a mode to execute 8086 programs without change.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

- **1989–95**: The subsequent 80486 in 1989, Pentium in 1992, and Pentium Pro in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing (Chapter 7) and a conditional move instruction.

- **1997**: After the Pentium and Pentium Pro were shipping, Intel announced that it would expand the Pentium and the Pentium Pro architectures with MMX (Multi Media Extensions). This new set of 57 instructions uses the floating-point stack to accelerate multimedia and communication applications. MMX instructions typically operate on multiple short data elements at a time, in the tradition of single instruction, multiple data (SIMD) architectures (see Chapter 7). Pentium II did not introduce any new instructions.

- **1999**: Intel added another 70 instructions, labelled SSE (Streaming SIMD Extensions) as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE includes cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

- **2001**: Intel added yet another 144 instructions, this time labelled SSE2. The new data type is double precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable more multimedia operations, it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers like those found in other computers. This change boosted the floating-point performance of the Pentium 4, the first microprocessor to include SSE2 instructions.

- **2003**: A company other than Intel enhanced the x86 architecture this time. AMD announced a set of architectural extensions to increase the address space from 32 to 64 bits. Similar to the transition from a 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to 16 and increases the number of 128-bit SSE registers to 16. The primary ISA change comes from adding a new mode called *long mode* that redefines the execution of all x86 instructions with 64-bit addresses and data. To address the larger number of registers, it adds a new prefix to instructions. Depending how you count, long mode also adds four to ten new instructions and drops 27 old ones. PC-relative data addressing is another extension. AMD64 still has a mode that is identical to x86 (*legacy mode*) plus a mode that restricts user programs to x86 but allows operating systems to use AMD64 (*compatibility mode*). These modes allow a more graceful transition to 64-bit addressing than the HP/Intel IA-64 architecture.

■ **2004**: Intel capitulates and embraces AMD64, relabeling it Extended Memory 64 Technology (EM64T). The major difference is that Intel added a 128-bit atomic compare and swap instruction, which probably should have been included in AMD64. At the same time, Intel announced another generation of media extensions. SSE3 adds 13 instructions to support complex arithmetic, graphics operations on arrays of structures, video encoding, floating-point conversion, and thread synchronization (see Section 2.11). AMD will offer SSE3 in subsequent chips and it will almost certainly add the missing atomic swap instruction to AMD64 to maintain binary compatibility with Intel.

■ **2006**: Intel announces 54 new instructions as part of the SSE4 instruction set extensions. These extensions perform tweaks like sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, and so on. They also added support for virtual machines (see Chapter 5).

■ **2007**: AMD announces 170 instructions as part of SSE5, including 46 instructions of the base instruction set that adds three operand instructions like ARM and MIPS.

■ **2008**: Intel announces the Advanced Vector Extension that expands the SSE register width from 128 to 256 bits, thereby redefining about 250 instructions and adding 128 new instructions.

This history illustrates the impact of the "golden handcuffs" of compatibility on the x86, as the existing software base at each step was too important to jeopardize with significant architectural changes. If you looked over the life of the x86, on average the architecture has been extended by one instruction per month!

Whatever the artistic failures of the x86, keep in mind that there are more instances of this architectural family on desktop computers than of any other architecture, increasing by more than 250 million per year. Nevertheless, this checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

Brace yourself for what you are about to see! Do *not* try to read this section with the care you would need to write x86 programs; the goal instead is to give you familiarity with the strengths and weaknesses of the world's most popular desktop architecture.

Rather than show the entire 16-bit and 32-bit instruction set, in this section we concentrate on the 32-bit subset that originated with the 80386, as this portion of the architecture is what is used today. We start our explanation with the registers and addressing modes, move on to the integer operations, and conclude with an examination of instruction encoding.

x86 Registers and Data Addressing Modes

The registers of the 80386 show the evolution of the instruction set (Figure 2.34). The 80386 extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an *E* to their name to indicate the 32-bit version. We'll refer to them generically as

Name		Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
	CS	Code segment pointer
	SS	Stack segment pointer (top of stack)
	DS	Data segment pointer 0
	ES	Data segment pointer 1
	FS	Data segment pointer 2
	GS	Data segment pointer 3
EIP		Instruction pointer (PC)
EFLAGS		Condition codes

(The top registers span bits 31 to 0.)

FIGURE 2.34   The 80386 register set. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers.

GPRs (general-purpose registers). The 80386 contains only eight GPRs. This means MIPS programs can use four times as many and ARM twice as many.

Figure 2.35 shows the arithmetic, logical, and data transfer instructions are two-operand instructions. There are two important differences here. The x86 arithmetic and logical instructions must have one operand act as both a source and a destination; ARM and MIPS allow separate registers for source and destination. This restriction puts more pressure on the limited registers, since one source register must be modified. The second important difference is that one of the operands can be in memory. Thus, virtually any instruction may have one operand in memory, unlike ARM and MIPS.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

FIGURE 2.35   Instruction types for the arithmetic, logical, and data transfer instructions.
The x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode.
Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure 2.34
(not EIP or EFLAGS).

Mode	Description	Register restrictions
Register indirect	Address is in a register.	Not ESP or EBP
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP

FIGURE 2.36   x86 32-bit addressing modes with register restrictions. The Base plus Scaled
Index addressing mode, found in ARM or but not MIPS, is included to avoid the multiplies by 4 (scale fac-
tor of 2) to turn an index in a register into a byte address (see Figures 2.23 and 2.25). A scale factor of 1 is
used for 16-bit data, and a scale factor of 3 for 64-bit data. A scale factor of 0 means the address is not scaled.
(Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are
essentially identical and we combine them here.)

Data memory-addressing modes, described in detail below, offer two sizes of
addresses within the instruction. These so-called *displacements* can be 8 bits or 32 bits.

Although a memory operand can use any addressing mode, there are restric-
tions on which *registers* can be used in a mode. Figure 2.36 shows the x86 address-
ing modes and which GPRs cannot be used with each mode.

x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (*word*) data types. The
80386 adds 32-bit addresses and data (*double words*) in the x86. (AMD64 adds
64-bit addresses and data, called *quad words*; we'll stick to the 80386 in this sec-
tion.) The data type distinctions apply to register operations as well as memory
accesses.

Almost every operation works on both 8-bit data and on one longer data size. That size is determined by the mode and is either 16 bits or 32 bits.

Clearly, some programs want to operate on data of all three sizes, so the 80386 architects provided a convenient way to specify each version without expanding code size significantly. They decided that either 16-bit or 32-bit data dominates most programs, and so it made sense to be able to set a default large size. This default data size is set by a bit in the code segment register. To override the default data size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behaviour. The three original prefixes override the default segment register, lock the bus to support synchronization (see Section 2.11), or repeat the following instruction until the register ECX counts down to 0. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop

2. Arithmetic and logic instructions, including test, integer, and decimal arithmetic operations

3. Control flow, including conditional branches, unconditional jumps, calls, and returns

4. String instructions, including string move and string compare

The first two categories are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location. Figure 2.37 shows some typical x86 instructions and their functions.

Instruction	Function
`je name`	`if equal(condition code) {EIP=name};` `EIP-128 <= name < EIP+128`
`jmp name`	`EIP=name`
`call name`	`SP=SP-4; M[SP]=EIP+5; EIP=name;`
`movw EBX,[EDI+45]`	`EBX=M[EDI+45]`
`push ESI`	`SP=SP-4; M[SP]=ESI`
`pop EDI`	`EDI=M[SP]; SP=SP+4`
`add EAX,;6765`	`EAX= EAX+6765`
`test EDX,;42`	Set condition code (flags) with EDX and 42
`movsl`	`M[EDI]=M[ESI];` `EDI=EDI+4; ESI=ESI+4`

FIGURE 2.37   Some typical x86 instructions and their functions. A list of frequent operations appears in Figure 2.38. The `CALL` saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

Conditional branches on the x86 are based on *condition codes* or *flags*, like ARM. Unlike ARM, where condition codes are set or not depending on the S bit, condition codes are set as an implicit side effect of most x86 instructions. Branches then test the condition codes. PC-relative branch addresses must be specified in the number of bytes, since unlike ARM and MIPS, x86 instructions are not all 4 bytes in length.

String instructions are part of the 8080 ancestry of the x86 and are not commonly executed in most programs. They are often slower than equivalent software routines (see the fallacy on page 170).

Figure 2.38 lists some of the integer x86 instructions. Many of the instructions are available in both byte and word formats.

Instruction	Meaning
Control	**Conditional and unconditional branches**
jnz, jz	Jump if condition to EIP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
jmp	Unconditional jump—8-bit or 16-bit offset
call	Subroutine call—16-bit offset; return address pushed onto stack
ret	Pops return address from stack and jumps to it
loop	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
move	Move between two registers or between register and memory
push, pop	Push source operand on stack; pop operand from stack top to a register
les	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
add, sub	Add source to destination; subtract source from destination; register-memory format
cmp	Compare source and destination; register-memory format
shl, shr, rcr	Shift left; shift logical right; rotate right with carry condition code as fill
cbw	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
test	Logical AND of source and destination sets condition codes
inc, dec	Increment destination, decrement destination
or, xor	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
movs	Copies from string source to destination by incrementing ESI and EDI; may be repeated
lods	Loads a byte, word, or doubleword of a string into the EAX register

FIGURE 2.38   Some typical operations on the x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 80386 is complex, with many different instruction formats. Instructions for the 80386 may vary from 1 byte, when there are no operands, up to 15 bytes.

Figure 2.39 shows the instruction format for several of the example instructions in Figure 2.37. The opcode byte usually contains a bit saying whether the operand is 8 bits or 32 bits. For some instructions, the opcode may include the addressing

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV   EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

FIGURE 2.39 Typical x86 instruction formats. Figure 2.40 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 17 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

mode and the register; this is true in many instructions that have the form "register = register op immediate." Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m," which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The base plus scaled index mode uses a second postbyte, labeled "sc, index, base."

Figure 2.40 shows the encoding of the two postbyte address specifiers for both 16-bit and 32-bit mode. Unfortunately, to understand fully which registers and which addressing modes are available, you need to see the encoding of all address-ing modes and sometimes even the encoding of the instructions.

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+disp8	(sib)+disp8	SI+disp16	(sib)+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

FIGURE 2.40   The encoding of the first address specifier of the x86: mod, reg, r/m. The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16-bit mode (8086) or 32-bit mode (80386). The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 add-ing an 8-bit displacement and mod = 2 adding a 16-bit or 32-bit displacement, depending on the address mode. The exceptions are 1) r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; 2) r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and 3) r/m = 4 in 32-bit mode when mod does not equal 3, where (sib) means use the scaled index mode shown in Figure 2.36. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

x86 Conclusion

Intel had a 16-bit microprocessor two years before its competitors' more elegant architectures, such as the Motorola 68000, and this head start led to the selection of the 8086 as the CPU for the IBM PC. Intel engineers generally acknowledge that the x86 is more difficult to build than computers like ARM and MIPS, but the large market means AMD and Intel can afford more resources to help overcome the added complexity. What the x86 lacks in style, it makes up for in quantity, making it beautiful from the right perspective.

Its saving grace is that the most frequently used x86 architectural compo-nents are not too difficult to implement, as AMD and Intel have demonstrated by rapidly improving performance of integer programs since 1978. To get that performance, compilers must avoid the portions of the architecture that are hard to implement fast.

2.18   Fallacies and Pitfalls

Fallacy: More powerful instructions mean higher performance.

Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction. One prefix can repeat the following instruction until a counter counts down to 0. Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

An alternative method, which uses the standard instructions found in all computers, is to load the data into the registers and then store the registers back to memory. This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times faster. A third version, which uses the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times faster than the complex move instruction.

Fallacy: Write in assembly language to obtain the highest performance.

At one time compilers for programming languages produced naïve instruction sequences; the increasing sophistication of compilers means the gap between compiled code and code produced by hand is closing fast. In fact, to compete with current compilers, the assembly language programmer needs to understand the concepts in Chapters 4 and 5 thoroughly (processor pipelining and memory hierarchy).

This battle between compilers and assembly language coders is one situation in which humans are losing ground. For example, C offers the programmer a chance to give a hint to the compiler about which variables to keep in registers versus spilled to memory. When compilers were poor at register allocation, such hints were vital to performance. In fact, some old C textbooks spent a fair amount of time giving examples that effectively use register hints. Today's C compilers generally ignore such hints, because the compiler does a better job at allocation than the programmer does.

Even *if* writing by hand resulted in faster code, the dangers of writing in assembly language are the longer time spent coding and debugging, the loss in portability, and the difficulty of maintaining such code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language than in C or Java. Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines. Writing in higher-level language instead of assembly language not only allows future compilers to tailor the code to future machines, it also makes the software easier to maintain and allows the program to run on more brands of computers.

Fallacy: The importance of commercial binary compatibility means successful instruction sets don't change.

FIGURE 2.41   Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

While backwards binary compatibility is sacrosanct, Figure 2.41 shows that the x86 architecture has grown dramatically. The average is more than one instruction per month over its 30-year lifetime!

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by one.

Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes. Forewarned is forearmed!

Pitfall: Using a pointer to an automatic variable outside its defining procedure.

A common mistake in dealing with pointers is to pass a result from a procedure that includes a pointer to an array that is local to that procedure. Following the stack discipline in Figure 2.12, the memory that contains the local array will be reused as soon as the procedure returns. Pointers to automatic variables can lead to chaos.

2.19   Concluding Remarks

Less is more.

Robert Browning,
Andrea del Sarto, 1855

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid environmental scientists, financial

advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. As illustrated in this chapter, four design principles guide the authors of instruction sets in making that delicate balance:

1. *Simplicity favors regularity.* Regularity motivates many features of the ARM instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in each instruction format.

2. *Smaller is faster.* The desire for speed is the reason that ARM has 16 registers rather than many more.

3. *Make the common case fast.* Examples of making the common ARM case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

4. *Good design demands good compromises.* One ARM example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. Figure 2.42 lists the ARM instructions we have covered so far, both real and pseudoinstructions.

Each category of ARM instructions is associated with constructs that appear in programming languages:

■ The arithmetic instructions correspond to the operations found in assignment statements.

■ Data transfer instructions are most likely to occur when dealing with data structures like arrays or structures.

■ The conditional branches are used in *if* statements and in loops.

■ The unconditional jumps are used in procedure calls and returns and for *case/switch* statements.

These instructions are not born equal; the popularity of the few dominates the many. For example, Figure 2.43 shows the popularity of each class of instructions for SPEC2006. The varying popularity of instructions plays an important role in the chapters about, datapath, control, and pipelining.

After we explain computer arithmetic in Chapter 3, we reveal the rest of the ARM instruction set architecture.

ARM instructions	Name	Format
add	ADD	DP
subtract	SUB	DP
load register	LDR	DT
store register	STR	DT
load register halfword	LDRH	DT
load register halfword signed	LDRHS	DT
store register halfword	STRH	DT
load register byte	LDRB	DT
load register byte signed	LDRBS	DT
store register byte	STRB	DT
swap	SWP	DT
mov	MOV	DP
and	AND	DP
or	ORR	DP
not	MVN	DP
logical shift left (optional operation)	LSL	DP
logical shift right (optional operation)	LSR	DP
compare	CMP	DP
branch on X: EQ, NE, LT, LE, GT, GE LO, LS, HI, HS, VS, VC, MI, PL	Bx	BR
branch (always)	B	BR
branch and link	BL	BR

FIGURE 2.42 The ARM instruction set covered so far. ⊙ Appendixes B1, B2 **and** B3 **describe the full ARM architecture.** Figure 2.1 shows more details of the ARM architecture revealed in this chapter.

Instruction class	ARM examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	ADD, SUB, MOV	Operations in assignment statements	16%	48%
Data transfer	LDR, STR, LDRB, LDRSB, LDRH, LDRSH, STRB, STRH	References to data structures, such as arrays	35%	36%
Logical	AND, ORR, MNV, LSL, LSR	Operations in assignment statements	12%	4%
Conditional branch	B_, CMP	*If* statements and loops	34%	8%
Jump	B, BL	Procedure calls, returns, and *case/switch* statements	2%	0%

FIGURE 2.43 ARM instruction classes, examples, correspondence to high-level program language constructs, and percentage of ARM instructions executed by category for the average SPEC2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual ARM instructions executed. Extrapolated from measurements of MIPS programs.

2.20 Historical Perspective and Further Reading

This section surveys the history of instruction set architectures (ISAs) over time, and we give a short history of programming languages and compilers. ISAs include accumulator architectures, general-purpose register architectures, stack architectures, and a brief history of ARM, MIPS, and the x86. We also review the controversial subjects of high-level-language computer architectures and reduced instruction set computer architectures. The history of programming languages includes Fortran, Lisp, Algol, C, Cobol, Pascal, Simula, Smalltalk, C++, and Java, and the history of compilers includes the key milestones and the pioneers who achieved them. The rest of this section is on the CD.

2.21 Exercises

Contributed by John Oliver of Cal Poly, San Luis Obispo, with contributions from Nicole Kaiyan (University of Adelaide) and Milos Prvulovic (Georgia Tech)

A link for an ARM simulator is provided on the CD and is helpful for these exercises. Although the simulator accepts pseudoinstructions, try not to use pseudoinstructions for any exercises that ask you to produce ARM code. Your goal should be to learn the real ARM instruction set, and if you are asked to count instructions, your count should reflect the actual instructions that will be executed and not the pseudoinstructions.

There are some cases where pseudoinstructions must be used (for example, the `la` instruction when an actual value is not known at assembly time). In many cases, they are quite convenient and result in more readable code (for example, the `li` and `move` instructions). If you choose to use pseudoinstructions for these reasons, please add a sentence or two to your solution stating which pseudoinstructions you have used and why.

Exercise 2.1

The following problems deal with translating from C to ARM. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	`f = g + h + i + j;`
b.	`f = g + (h + 5);`

2.1.1 [5] <2.2> For the C statements above, what is the corresponding ARM assembly code? Use a minimal number of ARM assembly instructions.

2.1.2 [5] <2.2> For the C statements above, how many ARM assembly instructions are needed to perform the C statement?

2.1.3 [5] <2.2> If the variables f, g, h, i, and j have values 1, 2, 3, 4, and 5, respectively, what is the end value of f?

The following problems deal with translating from ARM to C. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	`ADD  f, g, h`
b.	`ADD  f, f, #1` `ADD  f, g, h`

2.1.4 [5] <2.2> For the ARM statements above, what is a corresponding C statement?

2.1.5 [5] <2.2> If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

Exercise 2.2

The following problems deal with translating from C to ARM. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	`f = f + f + i;`
b.	`f = g + (j + 2);`

2.2.1 [5] <2.2> For the C statements above, what is the corresponding ARM assembly code? Use a minimal number of ARM assembly instructions.

2.2.2 [5] <2.2> For the C statements above, how many ARM assembly instructions are needed to perform the C statement?

2.2.3 [5] <2.2> If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

The following problems deal with translating from ARM to C. For the following exercise, assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	`ADD  f, f, h`
b.	`RSB  f, f, #0` `ADD  f, f, 1`

2.2.4 [5] <2.2> For the ARM statements above, what is a corresponding C statement?

2.2.5 [5] <2.2> If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

Exercise 2.3

The following problems deal with translating from C to ARM. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	f = f + g + h + i + j + 2;
b.	f = g - (f + 5);

2.3.1 [5] <2.2> For the C statements above, what is the corresponding ARM assembly code? Use a minimal number of ARM assembly instructions.

2.3.2 [5] <2.2> For the C statements above, how many ARM assembly instructions are needed to perform the C statement?

2.3.3 [5] <2.2> If the variables f, g, h, i, and j have values 1, 2, 3, 4, and 5, respectively, what is the end value of f?

The following problems deal with translating from ARM to C. Assume that the variables g, h, i, and j are given and could be considered 32-bit integers as declared in a C program.

a.	ADD f, -g, h
b.	ADD h, f, #1 SUB f, g, h

2.3.4 [5] <2.2> For the ARM statements above, what is a corresponding C statement?

2.3.5 [5] <2.2> If the variables f, g, h, and i have values 1, 2, 3, and 4, respectively, what is the end value of f?

Exercise 2.4

The following problems deal with translating from C to ARM. Assume that the variables f, g, h, i, and j are assigned to registers r0, r1, r2, r3, and r4, respectively.

Assume that the base address of the arrays A and B are in registers r6 and r7, respectively.

a.	`f = g + h + B[4];`
b.	`f = g - A[B[4]];`

2.4.1 [10] <2.2, 2.3> For the C statements above, what is the corresponding ARM assembly code?

2.4.2 [5] <2.2, 2.3> For the C statements above, how many ARM assembly instructions are needed to perform the C statement?

2.4.3 [5] <2.2, 2.3> For the C statements above, how many different registers are needed to carry out the C statement?

The following problems deal with translating from ARM to C. Assume that the variables f, g, h, i, and j are assigned to registers r0, r1, r2, r3, and r4, respectively. Assume that the base address of the arrays A and B are in registers r6 and r7, respectively.

a.	`ADD r0, r0, r1`
	`add r0, r0, r2`
	`add r0, r0, r3`
	`add r0, r0, r4`

b.	`LDR r0, 4[r6, #0x4]`

2.4.4 [10] <2.2, 2.3> For the ARM assembly instructions above, what is the corresponding C statement?

2.4.5 [5] <2.2, 2.3> For the ARM assembly instructions above, rewrite the assembly code to minimize the number of ARM instructions (if possible) needed to carry out the same function.

2.4.6 [5] <2.2, 2.3> How many registers are needed to carry out the ARM assembly as written above? If you could rewrite the code above, what is the minimal number of registers needed?

Exercise 2.5

In the following problems, we will be investigating memory operations in the context of a ARM processor. The table below shows the values of an array stored in memory.

a.	Address	Data
	12	1
	8	6
	4	4
	0	2
b.	Address	Data
	16	1
	12	2
	8	3
	4	4
	0	5

2.5.1 [10] <2.2, 2.3> For the memory locations in the table above, write C code to sort the data from lowest-to-highest, placing the lowest value in the smallest memory location shown in the figure. Assume that the data shown represents the C variable called `Array`, which is an array of type `int`. Assume that this particular machine is a byte-addressable machine and a word consists of 4 bytes.

2.5.2 [10] <2.2, 2.3> For the memory locations in the table above, write ARM code to sort the data from lowest-to-highest, placing the lowest value in the smallest memory location. Use a minimum number of ARM instructions. Assume the base address of `Array` is stored in register `r6`.

2.5.3 [5] <2.2, 2.3> To sort the array above, how many instructions are required for the ARM code? If you are not allowed to use the immediate field in `lw` and `sw` instructions, how many ARM instructions do you need?

The following problems explore the translation of hexadecimal numbers to other number formats.

a.	`0x12345678`
b.	`0xbeadf00d`

2.5.4 [5] <2.3> Translate the hexadecimal numbers above into decimal.

2.5.5 [5] <2.3> Show how the data in the table would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Exercise 2.6

The following problems deal with translating from C to ARM. Assume that the variables f, g, h, i, and j are assigned to registers r0, r1, r2, r3, and r4, respectively. Assume that the base address of the arrays A and B are in registers r6 and r7, respectively.

a.	`f = -g + h + B[1];`
b.	`f = A[B[g]+1];`

2.6.1 [10] <2.2, 2.3> For the C statements above, what is the corresponding ARM assembly code?

2.6.2 [5] <2.2, 2.3> For the C statements above, how many ARM assembly instructions are needed to perform the C statement?

2.6.3 [5] <2.2, 2.3> For the C statements above, how many registers are needed to carry out the C statement using ARM assembly code?

The following problems deal with translating from ARM to C. Assume that the variables f, g, h, i, and j are assigned to registers r0, r1, r2, r3, and r4, respectively. Assume that the base address of the arrays A and B are in registers r6 and r7, respectively.

a.	``` ADD r0, r0, r1 ADD r0, r3, r2 ADD r0, r0, r3 ```
b.	``` ADD r6, r6, #-20 ;(SUB r6, r6, #20) ADD r6, r6, r1 LDR r0, [r6, #8] ```

2.6.4 [5] <2.2, 2.3> For the ARM assembly instructions above, what is the corresponding C statement?

2.6.5 [5] <2.2, 2.3> For the ARM assembly above, assume that the registers r0, r1, r2, r3, contain the values 10, 20, 30, and 40, respectively. Also, assume that register r6 contains the value 256, and that memory contains the following values:

Address	Value
256	100
260	200
264	300

Find the value of r0 at the end of the assembly code.

2.6.6 [10] <2.3, 2.5> For each ARM instruction, show the value of the *opcode, Rd, Rn, operand2* and *I* fields. Remember to distinguish between DP-type and DT-type instructions.

Exercise 2.7

The following problems explore number conversions from signed and unsigned binary number to decimal numbers.

a.	1010 1101 0001 0000 0000 0000 0000 0010$_{two}$
b.	1111 1111 1111 1111 1011 0011 0101 0011$_{two}$

2.7.1 [5] <2.4> For the patterns above, what base 10 number does it represent, assuming that it is a two's complement integer?

2.7.2 [5] <2.4> For the patterns above, what base 10 number does it represent, assuming that it is an unsigned integer?

2.7.3 [5] <2.4> For the patterns above, what hexadecimal number does it represent?

The following problems explore number conversions from decimal to signed and unsigned binary numbers.

a.	2147483647$_{ten}$
b.	1000$_{ten}$

2.7.4 [5] <2.4> For the base ten numbers above, convert to 2's complement binary.

2.7.5 [5] <2.4> For the base ten numbers above, convert to 2's complement hexadecimal.

2.7.6 [5] <2.4> For the base ten numbers above, convert the negated values from the table to 2's complement hexadecimal.

Exercise 2.8

The following problems deal with sign extension and overflow. Registers r0 and r1 hold the values as shown in the table below. You will be asked to perform a ARM operation on these registers and show the result.

a.	r0 = 70000000$_{sixteen}$, r1 = 0x0FFFFFFF$_{sixteen}$
b.	r0 = 0x40000000$_{sixteen}$, r1 = 0x40000000$_{sixteen}$

2.8.1 [5] <2.4> For the contents of registers r0 and r1 as specified above, what is the value of r4 for the following assembly code:

```
ADD r4, r0, r1
```

Is the result in r4 the desired result, or has there been overflow?

2.8.2 [5] <2.4> For the contents of registers r0 and r1 as specified above, what is the value of r0 for the following assembly code:

```
SUB r4, r0, r1
```

Is the result in r4 the desired result, or has there been overflow?

2.8.3 [5] <2.4> For the contents of registers r0 and r1 as specified above, what is the value of r4 for the following assembly code:

```
ADD r4, r0, r1
ADD r4, r0, r0
```

Is the result in r4 the desired result, or has there been overflow?

In the following problems, you will perform various ARM operations on a pair of registers, r0 and r1. Given the values of r0 and r1 in each of the questions below, state if there will be overflow.

a.	ADD r0, r0, r1
b.	SUB r0, r0, r1 sub r0, r0, r1

2.8.4 [5] <2.4> Assume that register r0 = 0x70000000 and r1 = 0x10000000. For the table above, will there be overflow?

2.8.5 [5] <2.4> Assume that register r0 = 0x40000000 and r1 = 0x20000000. For the table above, will there be overflow?

2.8.6 [5] <2.4> Assume that register r0 = 0x8FFFFFFF and r1 = 0xD0000000. For the table above, will there be overflow?

Exercise 2.9

The table below contains various values for register r1. You will be asked to evaluate if there would be overflow for a given operation.

a.	2147483647_{ten}
b.	$0xD0000000_{sixteen}$

2.9.1 [5] <2.4> Assume that register r0 = 0x70000000 and r1 has the value as given in the table. If the instruction: ADD r0, r0, r1 is executed, will there be overflow?

2.9.2 [5] <2.4> Assume that register r0 = 0x80000000 and r1 has the value as given in the table. If the instruction: SUB r0, r0, r1 is executed, will there be overflow?

2.9.3 [5] <2.4> Assume that register r0 = 0x7FFFFFFF and r1 has the value as given in the table. If the instruction: SUB r0, r0, r1 is executed, will there be overflow?

The table below contains various values for register r1. You will be asked to evaluate if there would be overflow for a given operation.

a.	1010 1101 0001 0000 0000 0000 0000 0010$_{two}$
b.	1111 1111 1111 1111 1011 0011 0101 0011$_{two}$

2.9.4 [5] <2.4> Assume that register r0 = 0x70000000 and r1 has the value as given in the table. If the instruction: ADD r0, r0, r1 is executed, will there be overflow?

2.9.5 [5] <2.4> Assume that register r0 = 0x70000000 and r1 has the value as given in the table. If the instruction: ADD r0, r0, r1 is executed, what is the result in hex?

2.9.6 [5] <2.4> Assume that register r0 = 0x70000000 and r1 has the value as given in the table. If the instruction: ADD r0, r0, r1 is executed, what is the result in base ten?

Exercise 2.10

In the following problems, the data table contains bits that represent the opcode of an instruction. You will be asked to translate the entries into assembly code and determine what format of ARM instruction the bits represent.

a.	1010 1110 0000 1011 0000 0000 0000 0100$_{two}$
b.	1000 1101 0000 1000 0000 0000 0100 0000$_{two}$

2.10.1 [5] <2.5> For the binary entries above, what instruction do they represent?

2.10.2 [5] <2.5> What type (DP-type, DT-type) instruction do the binary entries above represent?

2.10.3 [5] <2.4, 2.5> If the binary entries above were data bits, what number would they represent in hexadecimal?

In the following problems, the data table contains ARM instructions. You will be asked to translate the entries into the bits of the opcode and determine what is the ARM instruction format.

a.	`ADD r0, r0, r5`
b.	`LDR r1, [r3, #4]`

2.10.4 [5] <2.4, 2.5> For the instructions above, show the hexadecimal representation of these instructions.

2.10.5 [5] <2.5> What type (DP-type, DT-type) instruction do the instructions above represent?

2.10.6 [5] <2.5> What is the hexadecimal representation of the *opcode*, Rd, and Rn fields in this instruction? For DP-type instruction, what is the hexadecimal representation of the Rd, I and operand2 fields?

Exercise 2.11

In the following problems, the data table contains bits that represent the opcode of an instruction. You will be asked to translate the entries into assembly code and determine what format of ARM instruction the bits represent.

a.	`0xE0842005`
b.	`0xE0423001`

2.11.1 [5] <2.4, 2.5> What binary number does the above hexadecimal number represent?

2.11.2 [5] <2.4, 2.5> What decimal number does the above hexadecimal number represent?

2.11.3 [5] <2.5> What instruction does the above hexadecimal number represent?

In the following problems, the data table contains the values of various fields of ARM instructions. You will be asked to determine what the instruction is, and find the ARM format for the instruction.

a.	Cond=14, F=1, opcode=25, Rn=6, Rd=5, operand2/offset=0
b.	Cond=14, F=0, opcode=2, Rn=6, Rd=5, operand2/offset=0

2.11.4 [5] <2.5> What type (DP-type, DT-type) instruction do the instructions above represent?

2.11.5 [5] <2.5> What is the ARM assembly instruction described above?

2.11.6 [5] <2.4, 2.5> What is the binary representation of the instructions above?

Exercise 2.12

In the following problems, the data table contains various modifications that could be made to the ARM instruction set architecture. You will investigate the impact of these changes on the instruction format of the ARM architecture.

a.	8 registers & 10-bit immediate constant
b.	10 bit offset

2.12.1 [5] <2.5> If the instruction set of the ARM processor is modified, the instruction format must also be changed. For each of the suggested changes above, show the size of the bit fields of an DP-type format instruction. What is the total number of bits needed for each instruction?

2.12.2 [5] <2.5> If the instruction set of the ARM processor is modified, the instruction format must also be changed. For each of the suggested changes above, show the size of the bit fields of an DT-type format instruction. What is the total number of bits needed for each instruction?

2.12.3 [5] <2.5, 2.10> Why could the suggested change in the table above decrease the size of a ARM assembly program? Why could the suggested change in the table above increase the size of a ARM assembly program?

In the following problems, the data table contains hexadecimal values. You will be asked to determine what ARM instruction the value represents, and find the ARM instruction format.

a.	0xE2801000
b.	0xE1801020

2.12.4 [5] <2.5> For the entries above, what is the value of the number in decimal?

2.12.5 [5] <2.5> For the hexadecimal entries above, what instruction do they represent?

2.12.6 [5] <2.4, 2.5> What type (DP-type, DT-type) instruction do the binary entries above represent? What is the value of the *opcode* field and the *Rd* field?

Exercise 2.13

In the following problems, the data table contains the values for registers r3 and r4. You will be asked to perform several ARM logical operations on these registers.

a.	r3 = 0x55555555, r4 = 0x12345678
b.	r3 = 0xBEADFEED, r4 = 0xDEADFADE

2.13.1 [5] <2.6> For the lines above, what is the value of r5 for the following sequence of instructions:

```
OR r5, r4, r3, LSL #4
```

2.13.2 [5] <2.6> For the values in the table above, what is the value of r2 for the following sequence of instructions:

r5

```
MVN r3, #1
AND r5, r3, r4, LSL #4
```

2.13.3 [5] <2.6> For the lines above, what is the value of r5 for the following sequence of instructions:

```
MOV r5, 0xFFEF
AND r5, r5, r3, LSR #3
```

In the following exercise, the data table contains various ARM logical operations. You will be asked to find the result of these operations given values for registers r0 and r1.

a.	ORR r2, r1, r0, LSL #1
b.	AND r2, r1, r0, LSR #1

2.13.4 [5] <2.6> Assume that r0 = 0x0000A5A5 and r1 = 00005A5A. What is the value of r2 after the two instructions in the table?

2.13.5 [5] <2.6> Assume that r0 = 0xA5A50000 and r1 = A5A50000. What is the value of r2 after the two instructions in the table?

2.13.6 [5] <2.6> Assume that r0 = 0xA5A5FFFF and r1 = A5A5FFFF. What is the value of r2 after the two instructions in the table?

Exercise 2.14

The following figure shows the placement of a bit field in register r0.

In the following problems, you will be asked to write ARM instructions to extract the bits "Field" from register r0 and place them into register r1 at the location indicated in the following table.

2.14.1 [20] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 22 and j = 5 and places the field into r1 in the format shown in the data table.

2.14.2 [5] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 4 and j = 0 and places the field into r1 in the format shown in the data table.

2.14.3 [5] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 31 and j = 28 and places the field into r1 in the format shown in the data table.

In the following problems, you will be asked to write ARM instructions to extract the bits "Field" from register r0 shown in the figure and place them into register $t1 at the location indicated in the following table. The bits shown as "XXX" are to remain unchanged.

2.14.4 [20] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 17 and j = 11 and places the field into r1 in the format shown in the data table.

2.14.5 [5] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 5 and j = 0 and places the field into r1 in the format shown in the data table.

2.14.6 [5] <2.6> Find the shortest sequence of ARM instructions that extracts a field from r0 for the constant values i = 31 and j = 29 and places the field into r1 in the format shown in the data table.

Exercise 2.15

For these problems, the table holds some logical operations that are not included in the ARM instruction set. How can these instructions be implemented?

a.	`ANDN r1, r2, r3`	`// bit-wise AND of r2, !r3`
b.	`XNOR r1, r2, r3`	`// bit-wise exclusive-NOR`

2.15.1 [5] <2.6> The logical instructions above are not included in the ARM instruction set, but are described above. If the value of r2 = 0x00FFA5A5 and the value of r3 = 0xFFFF003C, what is the result in r1?

2.15.2 [10] <2.6> The logical instructions above are not included in the ARM instruction set, but can be synthesized using one or more ARM assembly instructions. Provide a minimal set of ARM instructions that may be used in place of the instructions in the table above.

2.15.3 [5] <2.6> For your sequence of instructions in 2.15.2, show the bit-level representation of each instruction.

Various C-level logical statements are shown in the table below. In this exercise, you will be asked to evaluate the statements and implement these C statements using ARM assembly instructions.

a.	`A = B & C[0];`
b.	`A = A ? B : C[0]`

2.15.4 [5] <2.6> The table above shows different C statements that use logical operators. If the memory location at C[0] contains the integer value 0x00001234, and the initial integer value of A and B are 0x00000000 and 0x00002222, what is the result value of A?

2.15.5 [5] <2.6> For the C statements in the table above, write a minimal sequence of ARM assembly instructions that does the identical operation.

2.15.6 [5] <2.6> For your sequence of instructions in 2.15.5, show the bit-level representation of each instruction.

Exercise 2.16

For these problems, the table holds various binary values for register r0. Given the value of r0, you will be asked to evaluate the outcome of different branches.

a.	1010 1101 0001 0000 0000 0000 0000 0010$_{two}$
b.	1111 1111 1111 1111 1111 1111 1111 1111$_{two}$

2.16.1 [5] <2.7> Suppose that register r0 contains a value from above and r1 has the value

```
    0011 1111 1111 1000 0000 0000 0000 0000₂wo
```

What is the value of r2 after the following instructions?

```
        MOV r2,#0
        CMP r0,r1
        BGE ELSE
        B   DONE
ELSE: MOV r2, #2
DONE:
```

2.16.2 [5] <2.7> Suppose that register r0 contains a value from above and r1 has the value

```
    0011 1111 1111 1000 0000 0000 0000 0000₂wo
```

What is the value of r2 after the following instructions?

```
        MOV r2,#0
        CMP r0,r1
        BLO ELSE
        B   DONE
ELSE: MOV r2, #2
DONE:
```

2.16.3 [5] <2.7> Rewrite the above code using conditional instructions of ARM.

For these problems, the table holds various binary values for register r0. Given the value of r0, you will be asked to evaluate the outcome of different branches.

a.	`0x00001000`
b.	`0x20001400`

2.16.4 [5] <2.7> Suppose that register `r0` contains a value from above. What is the value of `r2` after the following instructions?

```
      MOV  r2, #0
      CMP  r0, r0
      BLT    ELSE
      B      DONE
ELSE: ADD r2, r2, #2
DONE:
```

2.16.5 [5] <2.6, 2.7> Suppose that register `r0` contains a value from above. What is the value of `r2` after the following instructions?

```
      MOV  r2, #0
      CMP  r0, r0
      BHI    ELSE
      B    DONE
ELSE: ADD r2, r2, #2
DONE:
```

Exercise 2.17

For these problems, several instructions that are not included in the ARM instruction set are shown.

a.	`ABS r2, r3`	`# r2 = |r3|`
b.	`SGT r1, r2, r3`	`# R[r1] = (R[r2] > R[r3]) ? 1:0`

2.17.1 [5] <2.7> The table above contains some instructions not included in the ARM instruction set and the description of each instruction. Why are these instructions not included in the ARM instruction set.

2.17.2 [5] <2.7> The table above contains some instructions not included in the ARM instruction set and the description of each instruction. If these instructions were to be implemented in the ARM instruction set, what is the most appropriate instruction format?

2.17.3 [5] <2.7> For each instruction in the table above, find the shortest sequence of ARM instructions that performs the same operation.

For these problems, the table holds ARM assembly code fragments. You will be asked to evaluate each of the code fragments, familiarizing you with the different ARM branch instructions.

| **a.** | ```
LOOP:  CMP    r1, #0
       BLT    ELSE
       B      DONE
ELSE:  ADD    r3, r3, #2
       SUB    r1, r1, #1
       B      LOOP
DONE:
``` |
| **b.** | ```
LOOP:  MOV    r2, 0xA
LOOP2: ADD    r4, r4, #2
       SUB     r2, r2, #1
       CMP      r1, #0
       BNE    LOOP2
       SUB    r1, r1, #1
       BNE    LOOP
DONE:
``` |

2.17.4 [5] <2.7> For the loops written in ARM assembly above, assume that the register r1 is initialized to the value 10. What is the value in register r2 assuming the r2 is initially zero?

2.17.5 [5] <2.7> For each of the loops above, write the equivalent C code routine. Assume that the registers r3, r4, r1, and r2 are integers A, B, i, and temp, respectively.

2.17.6 [5] <2.7> For the loops written in ARM assembly above, assume that the register r1 is initialized to the value N. How many ARM instructions are executed?

Exercise 2.18

For these problems, the table holds some C code. You will be asked to evaluate these C code statements in ARM assembly code.

| **a.** | ```
for(i=0; i<10; i++)
    a += b;
``` |
| **b.** | ```
while (a < 10){
    D[a] = b + a;
    a += 1;
}
``` |

2.18.1 [5] <2.7> For the table above, draw a control-flow graph of the C code.

2.18.2 [5] <2.7> For the table above, translate the C code to ARM assembly code. Use a minimum number of instructions. Assume that the value a, b, i, j are in registers r0, r1, r3, r3, respectively. Also, assume that register r1 holds the base address of the array D.

2.18.3 [5] <2.7> How many ARM instructions does it take to implement the C code? If the variables a and b are initialized to 10 and 1 and all elements of D are initially 0, what is the total number of ARM instructions that is executed to complete the loop?

For these problems, the table holds ARM assembly code fragments. You will be asked to evaluate each of the code fragments, familiarizing you with the different ARM branch instructions.

a.	
| | ```
        MOV r1, #100
LOOP: LDR r3, [r2, #0]
        ADD r4, r4, r3
        ADD r2, r2, #4
        SUB r1, r1, #1
        CMP r1, #0
        BNE LOOP
``` |
| b. | |
| | ```
        ADD r1, r2, #400
LOOP: LDR r3, [r2, #0]
        ADD r4, r4, r3
        LDR r3, [r2, #0]
        ADD r4, r4, r3
        ADD r2, r2, #8
        CMP r1, r2
        BNE LOOP
``` |

2.18.4 [5] <2.7> What is the total number of ARM instructions executed?

2.18.5 [5] <2.7> Translate the loops above into C. Assume that the C-level integer i is held in register r1, r4 holds the C-level integer called result, and r2 holds the base address of the integer MemArray.

2.18.6 [5] <2.7> Rewrite the loop in ARM assembly to reduce the number of ARM instructions executed.

Exercise 2.19

For the following problems, the table holds C code functions. Assume that the first function listed in the table is called first. You will be asked to translate these C code routines into ARM Asembly.

a.	
| | ```
int compare(int a, int b) {
    if (sub(a, b) >= 0)
        return 1;
    else
        return 0;
}
int sub (int a, int b) {
    return a-b;
}
``` |
| b. | |
| | ```
int fib_iter(int a, int b, int n){
    if(n == 0)
        return b;
    else
        return fib_iter(a+b, a, n-1);
}
``` |

2.19.1 [15] <2.8> Implement the C code in the table in ARM assembly. What is the total number of ARM instructions needed to execute the function?

2.19.2 [5] <2.8> Functions can often be implemented by compilers "in-line". An in-line function is when the body of the function is copied into the program space, allowing the overhead of the function call to be eliminated. Implement an "in-line" version of the C code in the table in ARM assembly. What is the reduction in the total number of ARM assembly instructions needed to complete the function? Assume that the C variable n is initialized to 5.

2.19.3 [5] <2.8> For each function call, show the contents of the stack after the function call is made. Assume the stack pointer is originally at addresss 0x7ffffffc, and follow the register conventions as specified in Figure 2.11.

The following three problems in this exercise refer to a function f that calls another function func. The code for C function func is already compiled in another module using the ARM calling convention from Figure 2.14. The function declaration for func is "int func(int a, int b);". The code for function f is as follows:

a.	```int f(int a, int b, int c){```
	``` return func(func(a,b),c);```
	```}```
b.	```int f(int a, int b, int c){```
	``` return func(a,b)+func(b,c);```
	```}```

2.19.4 [10] <2.8> Translate function f into ARM assembler, also using the ARM calling convention from Figure 2.14. If you need to use registers r4 through r11, use the lower-numbered registers first.

2.19.5 [5] <2.8> Can we use the tail-call optimization in this function? If no, explain why not. If yes, what is the difference in the number of executed instructions in f with and without the optimization?

2.19.6 [5] <2.8> Right before your function f from Problem 2.19.4 returns, what do we know about contents of registers and sp? Keep in mind that we know what the entire function f looks like, but for function func we only know its declaration.

Exercise 2.20

This exercise deals with recursive procedure calls. For the following problems, the table has an assembly code fragment that computes the factorial of a number. However, the entries in the table have errors, and you will be asked to fix these errors.

a.	FACT:	SUB SP,SP,#8
		STR LR,[SP,#4]
		STR R0,[SP,#8]
		CMP R0,#1
		BGE L1
		MOV R1,#1
		ADD SP,SP,#8
		MOV PC,LR
	L1:	SUB R0,R0,#1
		BL FACT
		LDR R0,[SP,#4]
		LDR LR,[SP,0]
		ADD SP,SP,#4
		MUL R1,R0,R1
		MOV PC,LR
b.	FACT:	SUB SP,SP,#8
		STR LR,[SP,#4]
		STR R0,[SP,#8]
		CMP R0,#1
		BGE L1
		MOV R1,#1
		ADD SP,SP,#8
		MOV PC,LR
	L1:	SUB R0,R0,#1
		BL FACT
		LDR R0,[SP,#4]
		LDR LR,[SP,0]
		ADD SP,SP,#4
		MUL R1,R0,R1
		MOV PC,LR

2.20.1 [5] <2.8> The ARM assembly program above computes the factorial of a given input. The integer input is passed through register $r0$, and the result is returned in register $r1$. In the assembly code, there are a few errors. Correct the ARM errors.

2.20.2 [10] <2.8> For the recursive factorial ARM program above, assume that the input is 4. Rewrite the factorial program to operate in a nonrecursive manner.

What is the total number of instructions used to execute your solution from 2.20.2 versus the recursive version of the factorial program?

2.20.3 [5] <2.8> Show the contents of the stack after each function call, assuming that the input is 4.

For the following problems, the table has an assembly code fragment that computes a Fibonacci number. However, the entries in the table have errors, and you will be asked to fix these errors.

a.
```
FIB:    SUB     sp,sp,#12
        STR     lr,[sp,#0]
        STR     r2,[sp,#4]
        STR     r1,[sp,#8]
        CMP     r1,#1
        BGE     L1
        MOV     r0,r1
        B       EXIT

L1:     SUB     r1,r1,#1
        BL      FIB
        MOV     r2,r0
        SUB     r1,r1,#1
        BL      FIB
        ADD     r0,r0,r2

EXIT:   LDR     lr,[sp,#0]
        LDR     r1,[sp,#8]
        LDR     r2,[sp,#4]
        ADD     sp,sp,#12
        MOV     pc,lr
```

b.
```
FIB:    SUB     sp,sp,#12
        STR     lr,[sp,#0]
        STR     r2,[sp,#4]
        STR     r1,[sp,#8]
        CMP     r1,#1
        BGE     L1
        MOV     r0,r1
        B       EXIT

L1:     SUB     r1,r1,#1
        BL      FIB
        MOV     r2,r0
        SUB     r1,r1,#1
        BL      FIB
        ADD     r0,r0,r2

EXIT:   LDR     lr,[sp,#0]
        LDR     r1,[sp,#8]
        LDR     r2,[sp,#4]
        ADD     sp,sp,#12
        MOV     pc,lr
```

2.20.4 [5] <2.8> The ARM assembly program above computes the Fibonacci of a given input. The integer input is passed through register r1, and the result is returned in register r0. In the assembly code, there are a few errors. Correct the ARM errors.

2.20.5 [10] <2.8> For the recursive Fibonacci ARM program above, assume that the input is 4. Rewrite the Fibonacci program to operate in a nonrecursive manner. What is the total number of instructions used to execute your solution from 2.20.2 versus the recursive version of the factorial program?

2.20.6 [5] <2.8> Show the contents of the stack after each function call, assuming that the input is 4.

Exercise 2.21

Assume that the stack and the static data segments are empty and that the stack and global pointers start at address 0x7fff fffc and 0x1000 8000, respectively. Assume the calling conventions as specified in Figure 2.11 and that function inputs are passed using registers r0 and returned in register r1.

| **a.** | ```
main()
{
    leaf_function(1);
}
int leaf_function (int f)
{
    int result;
    result = f + 1;
    if (f > 5)
        return result;
    leaf_function(result);
}
``` |
| **b.** | ```
int my_global = 100;
main()
{
    int x = 10;
    int y = 20;
    int z;
    z = my_function(x, my_global)
}
int my_function(int x, int y)
{
    return x - y;
}
``` |

2.21.1 [5] <2.8> Show the contents of the stack and the static data segments after each function call.

2.21.2 [5] <2.8> Write ARM code for the code in the table above.

2.21.3 [5] <2.8> If the leaf function could use **temporary** registers, write the ARM code for the code in the table above.

The following three problems in this exercise refer to this function, written in ARM assembler following the calling conventions from Figure 2.14:

| **a.** | ```
f:    SUB r4,r0,r3
      ADD r4,r2,r4  LSL #1
      SUB r4,r4,r1
      MOV pc,lr
``` |

```
b.   f: ADD sp,sp,#8
        STR  lr,[sp,#4]
        STR  r3,[sp,#0]
        MOV r3,r2
        BL  g
        ADD  r0,r0,r3
        LDR  lr,[sp,#4]
        LDR  r3,[sp,#0]
        SUB sp,sp,#8
        MOV pc,lr
```

2.21.4 [10] <2.8> This code contains a mistake that violates the ARM calling convention. What is this mistake and how should it be fixed?

2.21.5 [10] <2.8> What is the C equivalent of this code? Assume that the function's arguments are named a, b, c, etc. in the C version of the function.

2.21.6 [10] <2.8> At the point where this function is called register r0, r1, r2, and r3 have values 1, 100, 1000, and 30, respectively. What is the value returned by this function? If another function g is called from f, assume that the value returned from g is always 500.

Exercise 2.22

This exercise explores ASCII and Unicode conversion. The following table shows strings of characters.

a.	A byte
b.	computer

2.22.1 [5] <2.9> Translate the strings into decimal ASCII byte values.

2.22.2 [5] <2.9> Translate the strings into 16-bit Unicode (using hex notation and the Basic Latin character set).

The following table shows hexadecimal ASCII character values.

a.	61 64 64
b.	73 68 69 66 74

2.22.3 [5] <2.5, 2.9> Translate the hexadecimal ASCII values to text.

Exercise 2.23

In this exercise, you will be asked to write a ARM assembly program that converts strings into the number format as specified in the table.

a.	positive integer decimal strings
b.	2's complement hexadecimal integers

2.23.1 [10] <2.9> Write a program in ARM assembly language to convert an ASCII number string with the conditions listed in the table above, to an integer. Your program should expect register r0 to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register r1. If a non-digit character appears anywhere in the string, your program should stop with the value –1 in register r1. For example, if register r0 points to a sequence of three bytes 50_{ten}, 52_{ten}, 0_{ten} (the null-terminated string "24"), then when the program stops, register r1 should contain the value 24_{ten}.

Exercise 2.24

Assume that the register r1 contains the address 0x1000 0000 and the register r2 contains the address 0x1000 0010.

a.	`LDRB r0, [r1,#0]` `STRH r0, [r2,#0]`
b.	`LDRB r0, [r1,#0]` `STRB r0, [r2,#0]`

2.24.1 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	12	34	56	78

What value is stored at the address pointed to by register r2? Assume that the memory location pointed to r2 is initialized to 0xFFFF FFFF.

2.24.2 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	80	80	80	80

What value is stored at the address pointed to by register r2? Assume that the memory location pointed to r2 is initialized to 0x0000 0000.

2.24.3 [5] <2.9> Assume that the data (in hexadecimal) at address 0x1000 0000 is:

1000 0000	11	00	00	FF

What value is stored at the address pointed to by register r2? Assume that the memory location pointed to r2 is initialized to 0x5555 5555.

Exercise 2.25

In this exercise, you will explore 32-bit constants in ARM. For the following problems, you will be using the binary data in the table below.

a.	1010 1101 0001 0000 0000 0000 0000 0010$_{two}$
b.	1111 1111 1111 1111 1111 1111 1111 1111$_{two}$

2.25.1 [10] <2.10> Write the ARM code that creates the 32-bit constants listed above and stores that value to register r1

2.25.2 [5] <2.6, 2.10> If the current value of the PC is 0x00000000, write the instruction(s), to get to the PC address shown in the table above.

2.25.3 [10] <2.10> If the immediate field of an ARM instruction was only 8 bits wide, would it be possible to create 32-bit constants ? If so, write the ARM code to do that.

2.25.4 [5] <2.10> How would you create the one's complement of these numbers?

2.25.5 [5] <2.10> Write the ARM machine code for the instructions used in 2.25.1.

Exercise 2.26

For this exercise, you will explore the addressing modes in ARM. Consider the four addressing modes given in the table below.

a.	Register offset
b.	Scaled register offset
c	Pre-indexed
d	Post-indexed

2.26.1 [5] <2.10> Give an example ARM instruction for each of the above ARM addressing modes.

2.26.2 [5] <2.10> For the instructions in 2.26.1, what is the instruction format type used for the given instruction?

2.26.3 [5] <2.10> List benefits and drawbacks of each ARM addressing mode. Write ARM code that shows these benefits and drawbacks.

For the following problems, you will use the instructions and information given below to determine the effect of the different addressing modes.

```
LDR r0,[r1,#4]
LDR r2,[r0,r3, LSL#2]
LDR r3,[r1,#4]!
STR r0,[r2],#4
```
```
r0=0x0000000 r1=0x00009000  r2=0x00009004 r3=0x00000002
mem[0x00000000]= 0x01010101
mem[0x00000004]= 0x04040404
mem[0x00000008]= 0x05050505
mem[0x00009000]= 0x02020202
mem[0x00009004]= 0x00000000
mem[0x00009008]= 0x06060606
```

2.26.4 [10] <2.10> What will be the values in the registers r0, r1, r2 and r3 when the above instructions are executed?

2.26.5 [10] <2.10> What will be the values in the memory locations given above?

Exercise 2.27

For this exercise, you will explore the addressing modes in ARM. Consider the four addressing modes given in the table below.

a.	Immediate
b.	Scaled register
c	Scaled register Pre-indexed
d	PC-relative

2.27.1 [5] <2.10> Give an example ARM instruction for each of the above ARM addressing modes.

2.27.2 [5] <2.10> For the instructions in 2.27.1, what is the instruction format type used for the given instruction?

2.27.3 [5] <2.10> List benefits and drawbacks of each ARM addressing mode. Write ARM code that shows these benefits and drawbacks.

Exercise 2.28

The following table contains ARM assembly code for a lock.

```
try:    MOV R3,#1
        SWP R2,R3,[R1,#0]
        CMP R2,#1
        BEQ try
        LDR R4,[R2,#0]
        ADD R3,R4,#1
        STR R3,[R2,#0]
        SWP R2,R3,[R1,#0]
```

2.28.1 [5] <2.11> For each test and fail of the "swp", how many instructions need to be executed?

2.28.2 [5] <2.11> For the swp-based lock-unlock code above, explain why this code may fail.

2.28.3 [15] <2.11> Re-write the code above so that the code may operate correct. Be sure to avoid any race conditions.

Each entry in the following table has code and also shows the contents of various registers. The notation, "(r1)" shows the contents of a memory location pointed to by register r1. The assembly code in each table is executed in the cycle shown on parallel processors with a shared memory space.

a.

Processor 1	Processor 2	Cycle	Processor 1		MEM	Processor 2	
			r2	r3	(r1)	r2	r3
		0	1	2	99	30	40
SWP R2,R3, [R1,#0]		1					
	SWP R2,R3,[R1,#0]	2					

b.

Processor 1	Processor 2	Cycle	Processor 1		MEM	Processor 2	
			r2	r3		(r1)	r2	r3	
		0	2	3		1	10	20	
	try: MOV R3,#1	1							
try: MOV R3,#1	SWP R2,R3,[R1,#0]	2							
SWP R2,R3,[R1,#0]		3							
CMP R2,#1		4							
BEQ try	CMP R2,#1	5							
	BEQ try	6							

2.28.4 [5] <2.11> Fill out the table with the value of the registers for each given cycle.

Exercise 2.29

The first three problems in this exercise refer to a critical section of the form

```
lock(lk);
operation
unlock(lk);
```

where the "operation" updates the shared variable `shvar` using the local (nonshared) variable `x` as follows:

	Operation
a.	shvar=shvar+x;
b.	shvar=min(shvar,x);

2.29.1 [10] <2.11> Write the ARM assembler code for this critical section, assuming that the address of the `lk` variable is in `r1`, the address of the `shvar` variable is in `r4`, and the value of variable `x` is in `r5`. Your critical section should not contain any function calls, i.e., you should include the ARM instructions for `lock()`, `unlock()`, `max()`, and `min()` operations. Use swp instructions to implement the `lock()` operation, and the `unlock()` operation is simply an ordinary store instruction.

2.29.2 [10] <2.11> Repeat problem 2.29.1, but this time use swp to perform an atomic update of the `shvar` variable directly, without using `lock()` and `unlock()`. Note that in this problem there is no variable `lk`.

2.29.3 [10] <2.11> Compare the best-case performance of your code from 2.29.1 and 2.29.2, assuming that each instruction takes one cycle to execute. Note: best-case means that swp always succeeds, the lock is always free when we want to `lock()`, and if there is a branch we take the path that completes the operation with fewer executed instructions.

2.29.4 [10] <2.11> Using your code from 2.29.2 as an example, explain what happens when two processors begin to execute this critical section at the same time, assuming that each processor executes exactly one instruction per cycle.

2.29.5 [10] <2.11> Explain why in your code from 2.29.2 register `r4` contains the address of variable `shvar` and not the value of that variable, and why register `r5` contains the value of variable `x` and not its address.

2.29.6 [10] <2.11> If we want to atomically perform the same operation on two shared variables (e.g., shvar1 and shvar2) in the same critical section, we can do this easily using the approach from 2.29.1 (simply put both updates between the lock operation and the corresponding unlock operation). Explain why we cannot do this using the approach from 2.29.2., i.e., why we cannot use swp to access both shared variables in a way that guarantees that both updates are executed together as a single atomic operation.

Exercise 2.30

Assembler pseudoinstructions are not a part of the ARM instruction set, but often appear in ARM programs. The table below contains some ARM pseudoinstructions that, when assembled, are translated to other ARM assembly instructions.

a.	LDR r0,#constant

2.30.1 [5] <2.12> For each pseudo instruction in the table above, give atleast two different sequence of actual ARM instructions to accomplish the same thing.

2.30.2 [5] <2.12> If the constant value is FFF0, which of these would you choose?

Exercise 2.31

The table below contains the link-level details of two different procedures. In this exercise, you will be taking the place of the linker.

a.		Procedure A				Procedure B		
	Text Segment	Address	Instruction		Text Segment	Address	Instruction	
		0	LDR r0, [r3, #0]			0	STR r1, [r3, #0]	
		4	BL 0			4	BL 0	
		…	…			…	…	
	Data Segment	0	(X)		Data Segment	0	(Y)	
		…	…			…	…	
	Relocation Info	Address	Instruction Type	Dependency	Relocation Info	Address	Instruction Type	Dependency
		0	LDR	X		0	STR	Y
		4	BL	B		4	BL	A
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		—	X			—	Y	
		—	B			—	A	

b.	Procedure A				Procedure B			
	Text Segment	Address	Instruction		Text Segment	Address	Instruction	
		0	LDR r0, [r3,#0]			0	STR r0, [r3,#0]	
		4	ORR r1, r0, #0			4	B 0	
		8	BL 0			…	…	
		…	…			0x180	MOV pc, lr	
						…	…	
	Data Segment	0	(X)		Data Segment	0	(Y)	
		…	…			…	…	
	Relocation Info	Address	Instruction Type	Dependency	Relocation Info	Address	Instruction Type	Dependency
		0	LDR	X		0	STR	Y
		4	ORR	X		4	B	F00
		8	BL	B				
	Symbol Table	Address	Symbol		Symbol Table	Address	Symbol	
		—	X			—	Y	
		—	B			0x180	F00	

2.31.1 [5] <2.12> Link the object files above to form the executable file header. Assume that Procedure A has a text size of 0x140, data size of 0x40 and Procedure B has a text size of 0x300 and data size of 0x50. Also assume the memory allocation strategy as shown in Figure 2.13.

2.31.2 [5] <2.12> What limitations, if any, are there on the size of an executable?

Exercise 2.32

The first three problems in this exercise assume that function swap, instead of the code in Figure 2.22, is defined in C as follows:

```
a.   void swap(int v[], int k, int j){
        int temp;
        temp=v[k];
        v[k]=v[j];
        v[j]=temp;
     }
```

```
b.   void swap(int *p){
        int temp;
        temp=*p;
        *p=*(p+1);
        *(p+1)=*p;
     }
```

2.32.1 [10] <2.13> Translate this function into ARM assembler code.

2.32.2 [5] <2.13> What needs to change in the `sort` function?

2.32.3 [5] <2.13> If we were sorting 8-bit bytes, not 32-bit words, how would your ARM code for `swap` in 2.32.1 change?

Exercise 2.33

The problems in this exercise refer to the following function, given as array code:

a.	```int find(int a[], int n, int x){ int i; for(i=0;i!=n;i++) if(a[i]==x) return i; return -1; }```
b.	```int count(int a[], int n, int x){ int res=0; int i; for(i=0;i!=n;i++) if(a[i]==x) res=res+1; return res; }```

2.33.1 [10] <2.14> Translate this function into ARM assembly.

2.33.2 [10] <2.14> Convert this function into pointer-based code (in C).

2.33.3 [10] <2.14> Translate your pointer-based C code from 2.33.2 into ARM assembly.

2.33.4 [5] <2.14> Compare the worst-case number of executed instructions per nonlast loop iteration in your array-based code from 2.33.1 and your pointer-based code from 2.33.3. Note: the worst-case occurs when branch conditions are such that the longest path through the code is taken, i.e., if there is an if statement, the result of the condition check is such that the path with more instructions is taken. However, if the result of the condition check would cause the loop to exit, then we assume that the path that keeps us in the loop is taken.

2.33.5 [5] <2.14> Compare the number of registers needed for your array-based code from 2.33.1 and for your pointer-based code from 2.33.3.

Exercise 2.34

The table below contains ARM assembly code. In the following problems, you will translate ARM assembly code to MIPS.

| a. | ```
        MOV    r0, ;10            ;init loop counter to 10
LOOP:   ADD    r0, r1             ;add r1 to r0
        SUBS   r0, 1              ;decrement counter
        BNE    LOOP               ;if Z=0 repeat loop
``` |
|---|---|
| b. | ```
        ROR    r1, r2, #4         ;r1 = r23:0 concatenated with r231:4
``` |

2.34.1 [5] <2.16> For the table above, translate this ARM assembly code to MIPS assembly code. Assume that ARM registers r0, r1, and r2 hold the same values as MIPS registers $s0, $s1, and $s2, respectively. Use MIPS temporary registers ($t0, etc.) where necessary.

2.34.2 [5] <2.16> For the MIPS assembly instructions in 2.34.1, indicate the instruction types.

The table below contains MIPS assembly code. In the following problems, you will translate MIPS assembly code to ARM.

| a. | ```
slt $t0, $s0, $s1
blt $t0, $0, FARAWAY
``` |
|---|---|
| b. | ```
add $s0, $s1, $s2
``` |

2.34.3 [5] <2.16> For the table above, find the ARM assembly code that corresponds to the sequence of MIPS assembly code.

2.34.4 [5] <2.16> Show the bit fields that represent the ARM assembly code.

Exercise 2.35

The ARM processor has a few different addressing modes that are not supported in MIPS. The following problems explore how these addressing modes can be realized on MIPS.

| a. | ```
LDR    r0, [r1]             ; r0 = memory[r1]
``` |
|---|---|
| b. | ```
LDMIA  r0, {r1, r2, r4}     ; r1 = memory[r0], r2 = memory[r0+4]
                            ; r4 = memory[r0+8]
``` |

2.35.1 [5] <2.16> Identify the type of addressing mode of the ARM assembly instructions in the table above.

2.35.2 [5] <2.16> For the ARM assembly instructions above, write a sequence of ARM assembly instructions to accomplish the same data transfer. In the following problems, you will compare code written using the ARM and ARM instruction sets.

The following table shows code written in the ARM instruction set.

| **a.** | ```
        LDR   r0, =Table1        ;load base address of table
        LDR   r1, #100           ;initialize loop counter
        EOR   r2, r2, r2         ;clear r2
ADDLP:  LDR   r4, [r0]           ;get first addition operand
        ADD   r2, r2, r4         ;add to r2
        ADD   r0, r0, #4         ;increment to next table element
        SUBS  r1, r1, #1         ;decrement loop counter
        BNE   ADDLP              ;if loop counter != 0, go to ADDLP
``` | |
| **b.** | ```
        ROR   r1, r2, #4         ;r1 = r23:0 concatenated with r231:4
``` | |

2.35.3 [10] <2.16> For the ARM assembly code above, write an equivalent MIPS assembly code routine.

2.35.4 [5] <2.16> What is the total number of ARM assembly instructions required to execute the code? What is the total number of MIPS assembly instructions required to execute the code?

2.35.5 [5] <2.16> Assuming that the average CPI of the MIPS assembly routine is the same as the average CPI of the ARM assembly routine, and the MIPS processor has an operation frequency that is 1.5 times of the ARM processor, how much faster is the ARM processor than the MIPS processor?

Exercise 2.36

The ARM processor has an interesting way of supporting immediate constants. This exercise investigates those differences. The following table contains ARM instructions.

a.	`ADD, r3, r2, r1, LSL #3   ;r3 = r2 + (r1 << 3)`
b.	`ADD, r3, r2, r1, ROR #3   ;r3 = r2 + (r1, rotated_right 3 bits)`

2.36.1 [5] <2.16> Write the equivalent MIPS code for the ARM assembly code above.

2.36.2 [5] <2.16> If the register R1 had the constant value of 8, re-write your MIPS code to minimize the number of MIPS assembly instructions needed.

2.36.3 [5] <2.16> If the register R1 had the constant value of 0x06000000, rewrite your MIPS code to minimize the number of MIPS assembly instructions needed.

The following table contains MIPS instructions.

a.	`addi r3, r2, 0x1`
b.	`addi r3, r2, 0x8000`

2.36.4 [5] <2.16> For the MIPS assembly code above, write the equivalent ARM assembly code.

Exercise 2.37

This exercise explores the differences between the ARM and x86 instruction sets. The following table contains x86 assembly code.

a.	`        mov edx, [esi+4*ebx]`

b.	`START:  mov  ax, 00101100b`
	`        mov  cx, 00000011b`
	`        mov  bx, 11110000b`
	`        and  ax, bx`
	`        or   ax, cx`

2.37.1 [10] <2.17> Write pseudo code for the given routine.

2.37.2 [10] <2.17> What is the equivalent ARM for the given routine?

The following table contains x86 assembly instructions.

a.	`mov edx, [esi+4*ebx]`
b.	`add  eax, 0x12345678`

2.37.3 [5] <2.17> For each assembly instruction, show the size of each of the bit fields that represent the instruction. Treat the label `MY_FUNCTION` as a 32-bit constant.

2.37.4 [10] <2.17> Write equivalent ARM assembly statements.

Exercise 2.38

The x86 instruction set includes the REP prefix that causes the instruction to be repeated a given number of times or until a condition is satisfied. The first three problems in this exercise refer to the following x86 instruction:

	Instruction	Interpretation
a.	REP MOVSB	Repeat until ECX is zero: Mem8[EDI]=Mem8[ESI], EDI=EDI+1, ESI=ESI+1, ECX=ECX-1
b.	REP MOVSD	Repeat until ECX is zero: Mem32[EDI]=Mem32[ESI], EDI=EDI+4, ESI=ESI+4, ECX=ECX-1

2.38.1 [5] <2.17> What would be a typical use for this instruction?

2.38.2 [5] <2.17> Write ARM code that performs the same operation, assuming that r0 corresponds to ECX, r1 to EDI, r2 to ESI, and r3 to EAX.

2.38.3 [5] <2.17> If the x86 instruction takes one cycle to read memory, one cycle to write memory, and one cycle for each register update, and if ARM takes one cycle per instruction, what is the speed-up of using this x86 instruction instead of the equivalent ARM code when ECX is very large? Assume that the clock cycle time for x86 and ARM is the same.

The remaining three problems in this exercise refer to the following function, given in both C and x86 assembly. For each x86 instruction, we also show its length in the x86 variable-length instruction format and the interpretation (what the instruction does). Note that the x86 architecture has very few registers compared to ARM, and as a result the x86 calling convention is to push all arguments onto the stack. The return value of an x86 function is passed back to the caller in the EAX register.

	C code	x86 code

a.	`int f(int a, int b){`
	`  return a+b;`
	`}`
		`f: push %ebp          ; 1B, push %ebp to stack`
		`   mov %esp,%ebp       ; 2B, move %esp to %ebp`
		`   mov 0xc(%ebp),%eax  ; 3B, load 2nd arg to %eax`
		`   add 0x8(%ebp),%eax  ; 3B, add 1st arg to %eax`
		`   pop %ebp            ; 1B, restore %ebp`
		`   ret                 ; 1B, return`

b.	`void f(int *a, int *b){`
	`  *a=*a+*b;`
	`  *b=*a;`
	`}`
		`f: push %ebp          ; 1B, push %ebp to stack`
		`   mov %esp,%ebp       ; 2B, move %esp to %ebp`
		`   mov 8(%ebp),%eax    ; 3B, load 1st arg into %eax`
		`   mov 12(%ebp),%ecx   ; 3B, load 2nd arg into %ecx`
		`   mov (%eax),%edx     ; 2B, load *a into %edx`
		`   add (%ecx),%edx     ; 2B, add *b to %edx`
		`   mov %edx,(%eax)     ; 2B, store %edx to *a`
		`   mov %edx,(%ecx)     ; 2B, store %edx to *b`
		`   pop %ebp            ; 1B, restore %ebp`
		`   ret                 ; 1B, return`

2.38.4 [5] <2.17> Translate this function into ARM assembly. Compare the size (how many bytes of instruction memory are needed) for this x86 code and for your ARM code.

2.38.5 [5] <2.17> If the processor can execute two instructions per cycle, it must at least be able to read two consecutive instructions in each cycle. Explain how it would be done in ARM and how it would be done in x86.

2.38.6 [5] <2.17> If each ARM instruction takes one cycle, and if each x86 instruction takes one cycle plus a cycle for each memory read or write it has to perform, what is the speed-up of using x86 instead of ARM? Assume that the clock cycle time is the same in both x86 and ARM, and that the execution takes the shortest possible path through the function (i.e., every loop is exited immediately and every if statement takes the direction that leads toward the return from the function). Note that x86 ret instruction reads the return address from the stack.

Exercise 2.39

The CPI of the different instruction types is given in the following table.

	Arithmetic	Load/Store	Branch
a.	2	10	3
b.	1	10	4

2.39.1 [5] <2.18> Assume the following instruction breakdown given for executing a given program:

	Instructions (in millions)
Arithmetic	500
Load/Store	300
Branch	100

What is the execution time for the processor if the operation frequency is 5 GHz?

2.39.2 [5] <2.18> Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, and the cost of increasing the clock cycle time by only 10%. Is this a good design choice? Why?

2.39.3 [5] <2.18> Suppose that we find a way to double the performance of arithmetic instructions? What is the overall speed-up of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times!?

The following table shows the proportions of instruction execution for the different instruction types.

	Arithmetic	**Load/Store**	**Branch**
a.	60%	20%	20%
b.	80%	15%	5%

2.39.4 [5] <2.18> Given the instruction mix above and the assumption that an arithmetic instruction requires 2 cycles, a load/store instruction takes 6 cycles, and a branch instruction takes 3 cycles, find the average CPI.

2.39.5 [5] <2.18> For a 25% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

2.39.6 [5] <2.18> For a 50% improvement in performance, how many cycles, on average, may an arithmetic instruction take if load/store and branch instructions are not improved at all?

Exercise 2.40

The first three problems in this exercise refer to the following function, given in ARM assembly. Unfortunately, the programmer of this function has fallen prey to the pitfall of assuming that ARM is a word addressed machine, but in fact ARM is byte-addressed.

a.
```
; int f(int a[],int n,int x);
f: MOV R4,#0        ; ret=0
   MOV R5,#0        ; i=0
L: ADD R6,R5,R0     ; &(a[i])
   LDR R6,[R6,#0]   ; read a[i]
   CMP R6,R2        ; if( a[i]==x)
   BNE S
   ADD R4,R4,#1     ; ret++
S: ADD R5,R5,#1     ; i++
   CMP R5,R1        ; repeat if i!=n
   BNE L
   MOV R0,R4
   MOV PC,LR        ; return ret
```

b.
```
; void  f(int *a,int *b,int n);
f: MOV R4,R0        ; p=a
   MOV R5,R1        ; q=b
   ADD R6,R2,R0     ; &(a[n])
L: LDR R7,[R4,#0]   ; read *p
   LDR R8,[R5,#0]   ; read *q
   ADD R7,R7,R8     ; *p + *q
   STR R7,[R4,#0]   ; *p = *p + *q
   ADD R4,R4,#1     ; p=p+1
   ADD R5,R5,#1     ; q=q+1
   CMP R4,R6,#1     ; repeat if p!= &(a[n])
   BNE L
   MOV PC,LR        ; return
```

Note that in ARM assembly the ";" character denotes that the remainder of the line is a comment.

2.40.1 [5] <2.18> The ARM architecture requires word-sized accesses (LDR and STR) to be word-aligned, i.e. the lowermost 2 bits of the address must both be zero. Explain how this alignment requirement affects the execution of this function.

2.40.2 [5] <2.18> If "a" was a pointer to the beginning of an array of one-byte elements, and if we replaced LDR and STR with LDRB (load byte) and STRB (store byte), respectively, would this function be correct?

2.40.3 [5] <2.18> Change this code to make it correct for 32-bit integers.

The remaining three problems in this exercise refer to a program that allocates memory for an array, fills the array with some numbers, calls the sort function from Figure 2.25, and then prints out the array. The main function of the program is as follows (given as both C and ARM code):

main code in C	ARM version of the main code
``main(){
int *v;
int n=5;
v=my_alloc(5);
my_init(v,n);
sort(v,n);
.
.
.
``	```main:
MOV R9,#5
MOV R0,R9
BL my_alloc
MOV R10,R0
MOV R0,R10
MOV R1,R9
BL my_init
MOV R0,R10
MOV R1,R9
BL sort
``

The my_alloc function is defined as follows (given as both C and ARM code). Note that the programmer of this function has fallen prey to the pitfall of using a pointer to an automatic variable arr outside the function in which it is defined.

my_alloc in C	ARM code for my_alloc
``int *my_alloc(int n){
int arr[n];
return arr;
}
``	```my_alloc:
SUB SP,SP,4   ; push
STR R11,[SP,#0] ; r11 to stack
MOV R11,SP   ; save sp in r11
LSL R4,R0,R2   ; We need 4*n bytes
SUB SP,SP,R4   ; Make room for arr
MOV R0,SP   ; Return address of arr
MOV SP,R11   ; Restore sp from r11
LDR R11,[SP,#0] ; pop r11
ADD SP,SP,#4   ; from stack
MOV PC,LR
``

The `my_init` function is defined as follows (ARM code):

<table>
<tr><td>**a.**</td><td>

```
my_init:
    MOV R4,#0      ; i=0
    MOV R5,R0
L:  MOV R6,#0
    STR R6,[R5,#0] ; v[i]=0
    ADD R5,R5,#4
    ADD R4,R4,#1   ; i=i+1
    CMP R4,R1      ; untill i==n
    BNE L
    MOV PC,LR
```

</td></tr>
<tr><td>**b.**</td><td>

```
my_init:
    MOV R4,#0      ; i=0
    MOV R5,R0
L:  SUB R6,R1,R4
    STR R6,[R5,#0] ; a[i]=n-i
    ADD R5,R5,#4
    ADD R4,R4,#1   ; i=i+1
    CMP R4,R1      ; until i==n
    BNE L
    MOV PC,LR
```

</td></tr>
</table>

2.40.4 [5] <2.18> What are the contents (values of all five elements) of array v right before the "BL sort" instruction in the main code is executed?

2.40.5 [15] <2.18, 2.13> What are the contents of array v right before the sort function enters its outer loop for the first time? Assume that registers sp, r9, and r10 have values of 0x1000, 20, and 40, respectively, at the beginning of the main code.

2.40.6 [10] <2.18, 2.13> What are the contents of the 5-element array pointed by v right after "BL sort" returns to the main code?

Answers to Check Yourself

§2.2, page 80: ARM, C, Java
§2.3, page 86: 2) Very slow
§2.4, page 92: 3) -8_{ten}
§2.5, page 99: 4) sub r2, r0, r1
§2.6, page 104: Both. AND with a mask pattern of 1s will leaves 0s everywhere but the desired field. Shifting left by the right amount removes the bits from the left of the field. Shifting right by the appropriate amount puts the field into the rightmost bits of the word, with 0s in the rest of the word. Note that AND leaves the field where it was originally, and the shift pair moves the field into the rightmost part of the word.

§2.7, page 112: I. All are true. II. 1).
§2.8, page 122: Both are true.
§2.9, page 127: I. 2) II. 3)
§2.11, page 134: Both are true.
§2.12, page 143: 4) Machine independence.