chapter

2

Instruction Set Architecture

Chapter Objectives

In this chapter you will learn about:

- Machine instructions and program execution
- Addressing methods for accessing register and memory operands
- Assembly language for representing machine instructions, data, and programs
- Stacks and subroutines

This chapter considers the way programs are executed in a computer from the machine instruction set viewpoint. Chapter 1 introduced the general concept that both program instructions and data operands are stored in the memory. In this chapter, we discuss how instructions are composed and study the ways in which sequences of instructions are brought from the memory into the processor and executed to perform a given task. The addressing methods that are commonly used for accessing operands in memory locations and processor registers are also presented.

The emphasis here is on basic concepts. We use a generic style to describe machine instructions and operand addressing methods that are typical of those found in commercial processors. A sufficient number of instructions and addressing methods are introduced to enable us to present complete, realistic programs for simple tasks. These generic programs are specified at the assembly-language level, where machine instructions and operand addressing information are represented by symbolic names. A complete instruction set, including operand addressing methods, is often referred to as the *instruction set architecture* (ISA) of a processor. For the discussion of basic concepts in this chapter, it is not necessary to define a complete instruction set, and we will not attempt to do so. Instead, we will present enough examples to illustrate the capabilities of a typical instruction set.

The concepts introduced in this chapter and in Chapter 3, which deals with input/output techniques, are essential for understanding the functionality of computers. Our choice of the generic style of presentation makes the material easy to read and understand. Also, this style allows a general discussion that is not constrained by the characteristics of a particular processor.

Since it is interesting and important to see how the concepts discussed are implemented in a real computer, we supplement our presentation in Chapters 2 and 3 with four examples of popular commercial processors. These processors are presented in Appendices B to E. Appendix B deals with the Nios II processor from Altera Corporation. Appendix C presents the ColdFire processor from Freescale Semiconductor, Inc. Appendix D discusses the ARM processor from ARM Ltd. Appendix E presents the basic architecture of processors made by Intel Corporation. The generic programs in Chapters 2 and 3 are presented in terms of the specific instruction sets in each of the appendices.

The reader can choose only one processor and study the material in the corresponding appendix to get an appreciation for commercial ISA design. However, knowledge of the material in these appendices is not essential for understanding the material in the main body of the book.

The vast majority of programs are written in high-level languages such as C, C++, or Java. To execute a high-level language program on a processor, the program must be translated into the machine language for that processor, which is done by a compiler program. Assembly language is a readable symbolic representation of machine language. In this book we make extensive use of assembly language, because this is the best way to describe how computers work.

We will begin the discussion in this chapter by considering how instructions and data are stored in the memory and how they are accessed for processing.

2.1 MEMORY LOCATIONS AND ADDRESSES

We will first consider how the memory of a computer is organized. The memory consists of many millions of storage *cells*, each of which can store a *bit* of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For

this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a *word* of information, and n is called the *word length*. The memory of a computer can be schematically represented as a collection of words, as shown in Figure 2.1.

Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits, as shown in Figure 2.2. A unit of 8 bits is called a *byte*. Machine instructions may require one or more words for their representation. We will discuss how machine instructions are encoded into memory words in a later section, after we have described instructions at the assembly-language level.

Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or *addresses* for each location. It is customary to use numbers from 0 to $2^k - 1$, for some suitable value of k, as the addresses of successive locations in the memory. Thus, the memory can have up to 2^k addressable locations. The 2^k addresses constitute the *address space* of the computer. For example, a 24-bit address generates an address space of 2^{24} (16,777,216) locations. This number is usually written as 16M (16 mega), where 1M is the number 2^{20} (1,048,576). A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations, where 1G is 2^{30}. Other notational conventions

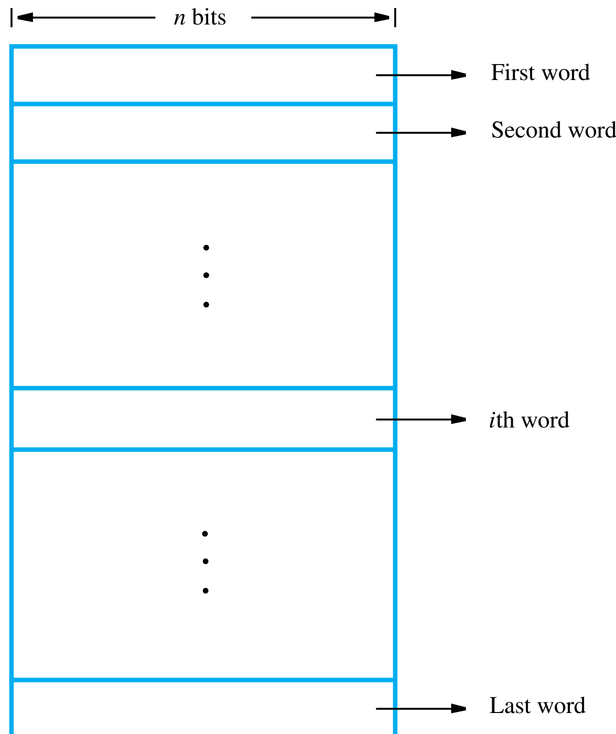

Figure 2.1 Memory words.

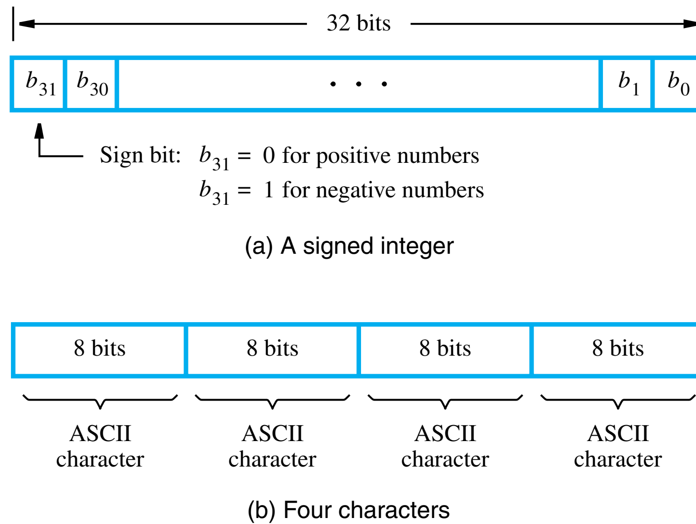

Figure 2.2 Examples of encoded information in a 32-bit word.

that are commonly used are K (kilo) for the number 2^{10} (1,024), and T (tera) for the number 2^{40}.

2.1.1 Byte Addressability

We now have three basic information quantities to deal with: bit, byte, and word. A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits. It is impractical to assign distinct addresses to individual bit locations in the memory. The most practical assignment is to have successive addresses refer to successive byte locations in the memory. This is the assignment used in most modern computers. The term *byte-addressable memory* is used for this assignment. Byte locations have addresses 0, 1, 2, ... Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, . . . , with each word consisting of four bytes.

2.1.2 Big-Endian and Little-Endian Assignments

There are two ways that byte addresses can be assigned across words, as shown in Figure 2.3. The name *big-endian* is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name *little-endian* is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word. The words “more significant” and “less significant” are used in relation to the weights (powers of 2) assigned to bits when the word represents a number. Both little-endian and big-endian assignments are used in commercial machines. In both cases, byte addresses 0, 4, 8, . . . , are taken as the addresses of successive words in the memory

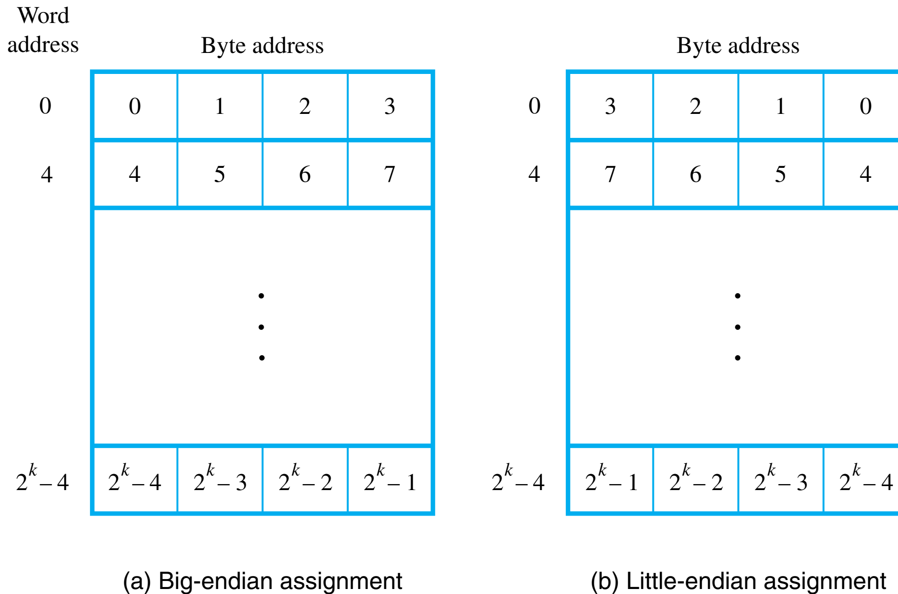

Figure 2.3 Byte and word addressing.

of a computer with a 32-bit word length. These are the addresses used when accessing the memory to store or retrieve a word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The most common convention, and the one we will use in this book, is shown in Figure 2.2*a*. It is the most natural ordering for the encoding of numerical data. The same ordering is also used for labeling bits within a byte, that is, $b_7, b_6, \ldots, b_0$, from left to right.

2.1.3 Word Alignment

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, . . . , as shown in Figure 2.3. We say that the word locations have *aligned* addresses if they begin at a byte address that is a multiple of the number of bytes in a word. For practical reasons associated with manipulating binary-coded addresses, the number of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0, 2, 4, . . . , and for a word length of 64 (2^3 bytes), aligned words begin at byte addresses 0, 8, 16, ...

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have *unaligned* addresses. But, the most common case is to use aligned addresses, which makes accessing of memory operands more efficient, as we will see in Chapter 8.

2.1.4 Accessing Numbers and Characters

A number usually occupies one word, and can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

For programming convenience it is useful to have different ways of specifying addresses in program instructions. We will deal with this issue in Section 2.4.

2.2 Memory Operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, *Read* and *Write*.

The Read operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Read operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The Write operation transfers an item of information from the processor to a specific memory location, overwriting the former contents of that location. To initiate a Write operation, the processor sends the address of the desired location to the memory, together with the data to be written into that location. The memory then uses the address and data to perform the write.

The details of the hardware implementation of these operations are treated in Chapters 5 and 6. In this chapter, we consider all operations from the viewpoint of the ISA, so we concentrate on the logical handling of instructions and operands.

2.3 Instructions and Instruction Sequencing

The tasks carried out by a computer program consist of a sequence of small steps, such as adding two numbers, testing for a particular condition, reading a character from the keyboard, or sending a character to be displayed on a display screen. A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

We begin by discussing instructions for the first two types of operations. To facilitate the discussion, we first need some notation.

2.3.1 REGISTER TRANSFER NOTATION

We need to describe the transfer of information from one location in a computer to another. Possible locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify such locations symbolically with convenient names. For example, names that represent the addresses of memory locations may be LOC, PLACE, A, or VAR2. Predefined names for the processor registers may be R0 or R5. Registers in the I/O subsystem may be identified by names such as DATAIN or OUTSTATUS. To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name. Thus, the expression

$$R2 \leftarrow [LOC]$$

means that the contents of memory location LOC are transferred into processor register R2.

As another example, consider the operation that adds the contents of registers R2 and R3, and places their sum into register R4. This action is indicated as

$$R4 \leftarrow [R2] + [R3]$$

This type of notation is known as *Register Transfer Notation* (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be placed, overwriting the old contents of that location.

In computer jargon, the words "transfer" and "move" are commonly used to mean "copy." Transferring data from a *source* location A to a *destination* location B means that the contents of location A are read and then written into location B. In this operation, only the contents of the destination will change. The contents of the source will stay the same.

2.3.2 ASSEMBLY-LANGUAGE NOTATION

We need another type of notation to represent machine instructions and programs. For this, we use *assembly language*. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R2, is specified by the statement

```
Load    R2, LOC
```

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example of adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

```
Add    R4, R2, R3
```

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*, which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation. To avoid the need for details of a particular assembly language at this early stage, we will continue the presentation in this chapter by using English words rather than processor-specific mnemonics.

2.3.3 RISC and CISC Instruction Sets

One of the most important characteristics that distinguish different computers is the nature of their instructions. There are two fundamentally different approaches in the design of instruction sets for modern computers. One popular approach is based on the premise that higher performance can be achieved if each instruction occupies exactly one word in memory, and all operands needed to execute a given arithmetic or logic operation specified by an instruction are already in processor registers. This approach is conducive to an implementation of the processing unit in which the various operations needed to process a sequence of instructions are performed in “pipelined” fashion to overlap activity and reduce total execution time of a program, as we will discuss in Chapter 6. The restriction that each instruction must fit into a single word reduces the complexity and the number of different types of instructions that may be included in the instruction set of a computer. Such computers are called *Reduced Instruction Set Computers* (RISC).

An alternative to the RISC approach is to make use of more complex instructions which may span more than one word of memory, and which may specify more complicated operations. This approach was prevalent prior to the introduction of the RISC approach in the 1970s. Although the use of complex instructions was not originally identified by any particular label, computers based on this idea have been subsequently called *Complex Instruction Set Computers* (CISC).

We will start our presentation by concentrating on RISC-style instruction sets because they are simpler and therefore easier to understand. Later we will deal with CISC-style instruction sets and explain the key differences between the two approaches.

2.3.4 Introduction to RISC Instruction Sets

Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A *load/store architecture* is used, in which
    - Memory operands are accessed only using Load and Store instructions.
    - All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

the loop terminates. Until then, the conditional branch instruction transfers control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from registers R3, R4, and R5, into memory locations SUM1, SUM2, and SUM3, respectively.

It should be emphasized that the contents of the index register, R2, are not changed when it is used in the Index addressing mode to access the scores. The contents of R2 are changed only by the last Add instruction in the loop, to move from one student record to the next.

In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears. In the example just given, the ID locations of successive student records are the reference points, and the test scores are the operands accessed by the Index addressing mode.

We have introduced the most basic form of indexed addressing that uses a register R*i* and a constant offset X. Several variations of this basic form provide for efficient access to memory operands in practical programming situations (although they may not be included in some processors). For example, a second register R*j* may be used to contain the offset X, in which case we can write the Index mode as

(R*i*,R*j*)

The effective address is the sum of the contents of registers R*i* and R*j*. The second register is usually called the *base* register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Yet another version of the Index mode uses two registers plus a constant, which can be denoted as

X(R*i*,R*j*)

In this case, the effective address is the sum of the constant X and the contents of registers R*i* and R*j*. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (R*i*,R*j*) part of the addressing mode.

Finally, we should note that in the basic Index mode

X(R*i*)

if the contents of the register are equal to zero, then the effective address is just equal to the sign-extended value of X. This has the same effect as the Absolute mode. If register R0 always contains the value zero, then the Absolute mode is implemented simply as

X(R0)

2.5 Assembly Language

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the patterns. So far, we have used normal words, such as Load, Store, Add, and

Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called *mnemonics*, such as LD, ST, ADD, and BR. A shorthand notation is also useful when identifying registers, such as R3 for register 3. Finally, symbols such as LOC may be defined as needed to represent particular memory locations. A complete set of such symbolic names and rules for their use constitutes a programming language, generally referred to as an *assembly language*. The set of rules for using the mnemonics and for specification of complete instructions and programs is called the *syntax* of the language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*. The assembler program is one of a collection of utility programs that are a part of the system software of a computer. The assembler, like any other program, is stored as a sequence of machine instructions in the memory of the computer. A user program is usually entered into the computer through a keyboard and stored either in the memory or on a magnetic disk. At this point, the user program is simply a set of lines of alphanumeric characters. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine-language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a *source program*, and the assembled machine-language program is called an *object program*. We will discuss how the assembler program works in Section 2.5.2 and in Chapter 4. First, we present a few aspects of assembly language itself.

The assembly language for a given computer may or may not be case sensitive, that is, it may or may not distinguish between capital and lower-case letters. In this section, we use capital letters to denote all names and labels in our examples to improve the readability of the text. For example, we write a Store instruction as

```
ST     R2, SUM
```

The mnemonic ST represents the binary pattern, or *operation (OP) code*, for the operation performed by the instruction. The assembler translates this mnemonic into the binary OP code that the computer recognizes.

The OP-code mnemonic is followed by at least one blank space or tab character. Then the information that specifies the operands is given. In the Store instruction above, the source operand is in register R2. This information is followed by the specification of the destination operand, separated from the source operand by a comma. The destination operand is in the memory location that has its binary address represented by the name SUM.

Since there are several possible addressing modes for specifying operand locations, an assembly-language instruction must indicate which mode is being used. For example, a numerical value or a name used by itself, such as SUM in the preceding instruction, may be used to denote the Absolute mode. The number sign usually denotes an immediate operand. Thus, the instruction

```
ADD     R2, R3, #5
```

adds the number 5 to the contents of register R3 and puts the result into register R2. The number sign is not the only way to denote the Immediate addressing mode. In some assembly languages, the Immediate addressing mode is indicated in the OP-code mnemonic.

For example, the previous Add instruction may be written as

```
ADDI    R2, R3, 5
```

The suffix I in the mnemonic ADDI states that the second source operand is given in the Immediate addressing mode.

Indirect addressing is usually specified by putting parentheses around the name or symbol denoting the pointer to the operand. For example, if register R2 contains the address of a number in the memory, then this number can be loaded into register R3 using the instruction

```
LD    R3, (R2)
```

2.5.1 Assembler Directives

In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an *equate* statement such as

```
TWENTY    EQU    20
```

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name TWENTY should be replaced by the value 20 wherever it appears in the program. Such statements, called *assembler directives* (or *commands*), are used by the assembler while it translates a source program into an object program.

To illustrate the use of assembly language further, let us reconsider the program in Figure 2.8. In order to run this program on a computer, it is necessary to write its source code in the required assembly language, specifying all of the information needed to generate the corresponding object program. Suppose that each instruction and each data item occupies one word of memory. Also assume that the memory is byte-addressable and that the word length is 32 bits. Suppose also that the object program is to be loaded in the main memory as shown in Figure 2.12. The figure shows the memory addresses where the machine instructions and the required data items are to be found after the program is loaded for execution. If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

To provide this information, the source program may be written as shown in Figure 2.13. The program begins with the assembler directive, ORIGIN, which tells the assembler program where in the memory to place the instructions that follow. It specifies that the instructions

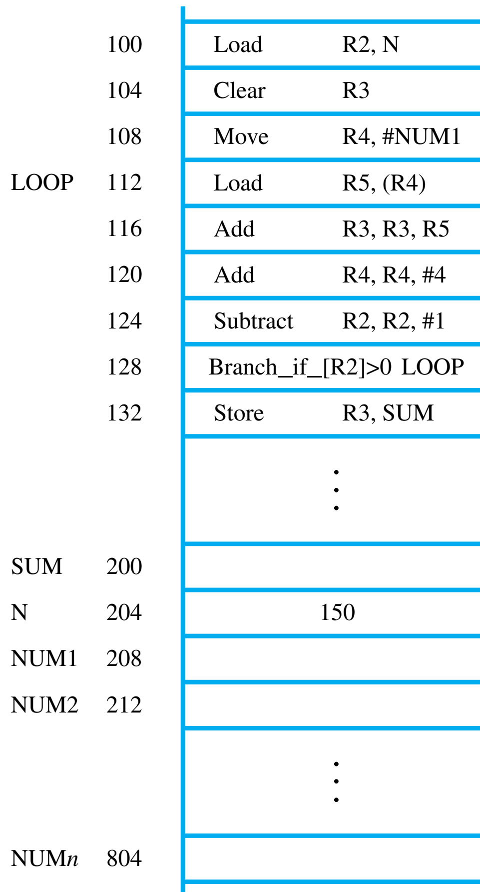

Figure 2.12 Memory arrangement for the program in Figure 2.8.

of the object program are to be loaded in the memory starting at address 100. It is followed by the source program instructions written with the appropriate mnemonics and syntax. Note that we use the statement

BGT R2, R0, LOOP

to represent an instruction that performs the operation

Branch_if_[R2]>0 LOOP

The second ORIGIN directive tells the assembler program where in the memory to place the data block that follows. In this case, the location specified has the address 200. This is intended to be the location in which the final sum will be stored. A 4-byte space for the sum is reserved by means of the assembler directive RESERVE. The next word, at address 204, has to contain the value 150 which is the number of entries in the list.

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Figure 2.13 Assembly language representation for the program in Figure 2.12.

The DATAWORD directive is used to inform the assembler of this requirement. The next RESERVE directive declares that a memory block of 600 bytes is to be reserved for data. This directive does not cause any data to be loaded in these locations. Data may be loaded in the memory using an input procedure, as we will explain in Chapter 3. The last statement in the source program is the assembler directive END, which tells the assembler that this is the end of the source program text.

We previously described how the EQU directive can be used to associate a specific value, which may be an address, with a particular name. A different way of associating addresses with names or labels is illustrated in Figure 2.13. Any statement that results in instructions or data being placed in a memory location may be given a memory address label. The assembler automatically assigns the address of that location to the label. For example, in the data block that follows the second ORIGIN directive, we used the labels SUM, N, and NUM1. Because the first RESERVE statement after the ORIGIN directive is given the label SUM, the name SUM is assigned the value 200. Whenever SUM is encountered in the program, it will be replaced with this value. Using SUM as a label in

this manner is equivalent to using the assembler directive

SUM EQU 200

Similarly, the labels N and NUM1 are assigned the values 204 and 208, respectively, because they represent the addresses of the two word locations immediately following the word location with address 200.

Most assembly languages require statements in a source program to be written in the form

Label: Operation Operand(s) Comment

These four *fields* are separated by an appropriate delimiter, perhaps one or more blank or tab characters. The Label is an optional name associated with the memory address where the machine-language instruction produced from the statement will be loaded. Labels may also be associated with addresses of data items. In Figure 2.13 there are four labels: LOOP, SUM, N, and NUM1.

The Operation field contains an assembler directive or the OP-code mnemonic of the desired instruction. The Operand field contains addressing information for accessing the operands. The Comment field is ignored by the assembler program. It is used for documentation purposes to make the program easier to understand.

We have introduced only the very basic characteristics of assembly languages. These languages differ in detail and complexity from one computer to another.

2.5.2 Assembly and Execution of Programs

A source program written in an assembly language must be assembled into a machine-language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the addresses given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands, and it reserves memory space as requested by RESERVE commands.

A key part of the assembly process is determining the values that replace the names. In some cases, where the value of a name is specified by an EQU directive, this is a straightforward task. In other cases, where a name is defined in the Label field of a given instruction, the value represented by the name is determined by the location of this instruction in the assembled object program. Hence, the assembler must keep track of addresses as it generates the machine code for successive instructions. For example, the names LOOP and SUM in the program of Figure 2.13 will be assigned the values 112 and 200, respectively.

In some cases, the assembler does not directly replace a name representing an address with the actual value of this address. For example, in a branch instruction, the name that specifies the location to which a branch is to be made (the branch target) is not replaced by the actual address. A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter

to the target instruction. The assembler computes this *branch offset*, which can be positive or negative, and puts it into the machine instruction. We will show how branch instructions may be implemented in Section 2.13.

The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk. The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a *loader* must already be in the memory. Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory. The loader must know the length of the program and the address in the memory where it will be stored. The assembler usually places this information in a header preceding the object code. Having loaded the object code, the loader starts execution of the object program by branching to the first instruction to be executed, which may be identified by an address label such as START. The assembler places that address in the header of the object code for the loader to use at execution time.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can only detect and report syntax errors. To help the user find other programming errors, the system software usually includes a *debugger* program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

In this section, we introduced some important issues in assembly and execution of programs. Chapter 4 provides a more detailed discussion of these issues.

2.5.3 Number Notation

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly-language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used as an immediate operand, it can be given as a decimal number, as in the instruction

```
ADDI    R2, R3, 93
```

or as a binary number identified by an assembler-specific prefix symbol such as a percent sign, as in

```
ADDI    R2, R3, %01011101
```

Binary numbers can be written more compactly as *hexadecimal*, or *hex*, numbers, in which four bits are represented by a single hex digit. The first ten patterns 0000, 0001, . . . , 1001, referred to as *binary-coded decimal* (BCD), are represented by the digits 0, 1, . . . , 9. The remaining six 4-bit patterns, 1010, 1011, . . . , 1111, are represented by the letters A, B, . . . , F. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by the prefix 0x (as in the C language) or

by a dollar sign prefix. Thus, we would write

```
ADDI    R2, R3, 0x5D
```

2.6 STACKS

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A *stack* is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a *pushdown* stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, *last-in–first-out* (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms *push* and *pop* are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

In modern computers, a stack is implemented by using a portion of the main memory for this purpose. One processor register, called the *stack pointer* (SP), is used to point to a particular stack structure called the *processor stack*, whose use will be explained shortly.

Data can be stored in a stack with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 2.14 shows an example of a stack of word data items. The stack contains numerical values, with 43 at the bottom and −28 at the top. The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time. If we assume a byte-addressable memory with a 32-bit word length, the push operation can be implemented as

```
Subtract    SP, SP, #4
Store       Rj, (SP)
```

where the Subtract instruction subtracts 4 from the contents of SP and places the result in SP. Assuming that the new item to be pushed on the stack is in processor register R*j*, the Store instruction will place this value on the stack. These two instructions copy the word from R*j* onto the top of the stack, decrementing the stack pointer by 4 before the store (push) operation. The pop operation can be implemented as

```
Load    Rj, (SP)
Add     SP, SP, #4
```

These two instructions load (pop) the top value from the stack into register R*j* and then increment the stack pointer by 4 so that it points to the new top element. Figure 2.15 shows the effect of each of these operations on the stack in Figure 2.14.