(160)

System Software

An Introduction to Systems Programming

Third Edition

Leland L. Beck
San Diego State University

(1109)

University
15710

Addison Wesley Longman

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Menlo Park, California • New York • Harlow, England
Don Mills, Ontario • Sydney • Mexico City • Madrid • Amsterdam

Chapter 1

Background

This chapter contains a variety of information that serves as background for the material presented later. Section 1.1 gives a brief introduction to system software and an overview of the structure of this book. Section 1.2 begins a discussion of the relationships between system software and machine architecture, which continues throughout the text. Section 1.3 describes the Simplified Instructional Computer (SIC) that is used to present fundamental software concepts. Sections 1.4 and 1.5 provide an introduction to the architecture of several computers that are used as examples throughout the text. Further information on most of the machine architecture topics discussed can be found in Tabak (1995) and Patterson and Hennessy (1996).

Most of the material in this chapter is presented at a summary level, with many details omitted. The level of detail given here is sufficient background for the remainder of the text. You should not attempt to memorize the material in this chapter, or be overly concerned with minor points. Instead, it is recommended that you read through this material, and then use it for reference as needed in later chapters. References are provided throughout the chapter for readers who want further information.

1.1 INTRODUCTION

This text is an introduction to the design and implementation of system software. *System software* consists of a variety of programs that support the operation of a computer. This software makes it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

When you took your first programming course, you were already using many different types of system software. You probably wrote programs in a high-level language like C++ or Pascal, using a *text editor* to create and modify the program. You translated these programs into machine language using a *compiler*. The resulting machine language program was loaded into memory and prepared for execution by a *loader* or *linker*. You may have used a *debugger* to help detect errors in the program.

In later courses, you probably wrote programs in assembler language. You may have used macro instructions in these programs to read and write data, or to perform other higher-level functions. You used an *assembler*, which probably included a *macro processor*, to translate these programs into machine language. The translated programs were prepared for execution by the loader or linker, and may have been tested using the debugger.

You controlled all of these processes by interacting with the *operating system* of the computer. If you were using a system like UNIX or DOS, you probably typed commands at a keyboard. If you were using a system like MacOS or Windows, you probably specified commands with menus and a point-and-click interface. In either case, the operating system took care of all the machine-level details for you. Your computer may have been connected to a network, or may have been shared by other users. It may have had many different kinds of storage devices, and several ways of performing input and output. However, you did not need to be concerned with these issues. You could concentrate on what you wanted to do, without worrying about how it was accomplished.

As you read this book, you will learn about several important types of system software. You will come to understand the processes that were going on "behind the scenes" as you used the computer in previous courses. By understanding the system software, you will gain a deeper understanding of how computers actually work.

The major topics covered in this book are assemblers, loaders and linkers, macro processors, compilers, and operating systems; each of Chapters 2 through 6 is devoted to one of these subjects. We also consider implementations of these types of software on several real machines. One central theme of the book is the relationship between system software and machine architecture: the design of an assembler, operating system, etc., is influenced by the architecture of the machine on which it is to run. Some of these influences are discussed in the next section; many other examples appear throughout the text.

Chapter 7 contains a survey of some other important types of system software: database management systems, text editors, and interactive debugging systems. Chapter 8 contains an introduction to software engineering concepts and techniques, focusing on the use of such methods in writing system software. This chapter can be read at any time after the introduction to assemblers in Section 2.1.

The depth of treatment in this text varies considerably from one topic to another. The chapters on assemblers, loaders and linkers, and macro processors contain enough implementation details to prepare the reader to write these types of software for a real computer. Compilers and operating systems, on the other hand, are very large topics; each has, by itself, been the subject of many complete books and courses. It is obviously impossible to provide a full coverage of these subjects in a single chapter of any reasonable size. Instead, we provide an introduction to the most important concepts and issues related to compilers and operating systems, stressing the relationships between software design and machine architecture. Other subtopics are discussed as space permits, with references provided for readers who wish to explore these areas further. Our goal is to provide a good overview of these subjects that can also serve as background for students who will later take more advanced software courses. This same approach is also applied to the other topics surveyed in Chapter 7.

1.2 SYSTEM SOFTWARE AND MACHINE ARCHITECTURE

One characteristic in which most system software differs from application software is machine dependency. An application program is primarily concerned with the solution of some problem, using the computer as a tool. The focus is on the application, not on the computing system. System programs, on the other hand, are intended to support the operation and use of the computer itself, rather than any particular application. For this reason, they are usually related to the architecture of the machine on which they are to run. For example, assemblers translate mnemonic instructions into machine code; the instruction formats, addressing modes, etc., are of direct concern in assembler design. Similarly, compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system. Many other examples of such machine dependencies may be found throughout this book.

On the other hand, there are some aspects of system software that do not directly depend upon the type of computing system being supported. For example, the general design and logic of an assembler is basically the same on most computers. Some of the code optimization techniques used by compilers are independent of the target machine (although there are also machine-dependent optimizations). Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used. We will also see many examples of such machine-independent features in the chapters that follow.

Because most system software is machine-dependent, we must include real machines and real pieces of software in our study. However, most real computers have certain characteristics that are unusual or even unique. It can be

difficult to distinguish between those features of the software that are truly fundamental and those that depend solely on the idiosyncrasies of a particular machine. To avoid this problem, we present the fundamental functions of each piece of software through discussion of a Simplified Instructional Computer (SIC). SIC is a hypothetical computer that has been carefully designed to include the hardware features most often found on real machines, while avoiding unusual or irrelevant complexities. In this way, the central concepts of a piece of system software can be clearly separated from the implementation details associated with a particular machine. This approach provides the reader with a starting point from which to begin the design of system software for a new or unfamiliar computer.

Each major chapter in this text first introduces the basic functions of the type of system software being discussed. We then consider machine-dependent and machine-independent extensions to these functions, and examples of implementations on actual machines. Specifically, the major chapters are divided into the following sections:

1. Features that are fundamental, and that should be found in any example of this type of software.
2. Features whose presence and character are closely related to the machine architecture.
3. Other features that are commonly found in implementations of this type of software, and that are relatively machine-independent.
4. Major design options for structuring a particular piece of software—for example, single-pass versus multi-pass processing.
5. Examples of implementations on actual machines, stressing unusual software features and those that are related to machine characteristics.

This chapter contains brief descriptions of SIC and of the real machines that are used as examples. You are encouraged to read these descriptions now, and refer to them as necessary when studying the examples in each chapter.

1.3 THE SIMPLIFIED INSTRUCTIONAL COMPUTER (SIC)

In this section we describe the architecture of our Simplified Instructional Computer (SIC). This machine has been designed to illustrate the most commonly encountered hardware features and concepts, while avoiding most of the idiosyncrasies that are often found in real machines.

Like many other products, SIC comes in two versions: the standard model and an XE version (XE stands for "extra equipment," or perhaps "extra expensive"). The two versions have been designed to be *upward compatible*—that is, an object program for the standard SIC machine will also execute properly on a SIC/XE system. (Such upward compatibility is often found on real computers that are closely related to one another.) Section 1.3.1 summarizes the standard features of SIC. Section 1.3.2 describes the additional features that are included in SIC/XE. Section 1.3.3 presents simple examples of SIC and SIC/XE programming. These examples are intended to help you become more familiar with the SIC and SIC/XE instruction sets and assembler language. Practice exercises in SIC and SIC/XE programming can be found at the end of this chapter.

1.3.1 SIC Machine Architecture

Memory

Memory consists of 8-bit bytes; any 3 consecutive bytes form a *word* (24 bits). All addresses on SIC are byte addresses; words are addressed by the location of their lowest numbered byte. There are a total of 32,768 (2^{15}) bytes in the computer memory.

Registers

There are five registers, all of which have special uses. Each register is 24 bits in length. The following table indicates the numbers, mnemonics, and uses of these registers. (The numbering scheme has been chosen for compatibility with the XE version of SIC.)

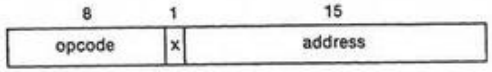
Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

Data Formats

Integers are stored as 24-bit binary numbers; 2's complement representation is used for negative values. Characters are stored using their 8-bit ASCII codes (see Appendix B). There is no floating-point hardware on the standard version of SIC.

Instruction Formats

All machine instructions on the standard version of SIC have the following 24-bit format:



The flag bit *x* is used to indicate indexed-addressing mode.

Addressing Modes

There are two addressing modes available, indicated by the setting of the *x* bit in the instruction. The following table describes how the *target address* is calculated from the address given in the instruction. Parentheses are used to indicate the contents of a register or a memory location. For example, (X) represents the contents of register X.

Mode	Indication	Target address calculation
Direct	x = 0	TA = address
Indexed	x = 1	TA = address + (X)

Instruction Set

SIC provides a basic set of instructions that are sufficient for most simple tasks. These include instructions that load and store registers (LDA, LDX, STA, STX, etc.), as well as integer arithmetic operations (ADD, SUB, MUL, DIV). All arithmetic operations involve register A and a word in memory, with the result being left in the register. There is an instruction (COMP) that compares the value in register A with a word in memory; this instruction sets a *condition code* CC to indicate the result (<, =, or >). Conditional jump instructions (JLT, JEQ, JGT) can test the setting of CC, and jump accordingly. Two instructions are provided for subroutine linkage. JSUB jumps to the subroutine, placing the return address in register L; RSUB returns by jumping to the address contained in register L.

Appendix A gives a complete list of all SIC (and SIC/XE) instructions, with their operation codes and a specification of the function performed by each.

Input and Output

On the standard version of SIC, input and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A. Each device is assigned a unique 8-bit code. There are three I/O instructions, each of which specifies the device code as an operand.

The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. The condition code is set to indicate the result of this test. (A setting of < means the device is ready to send or receive, and = means the device is not ready.) A program needing to transfer data must wait until the device is ready, then execute a Read Data (RD) or Write Data (WD). This sequence must be repeated for each byte of data to be read or written. The program shown in Fig. 2.1 (Chapter 2) illustrates this technique for performing I/O.

1.3.2 SIC/XE Machine Architecture

Memory

The memory structure for SIC/XE is the same as that previously described for SIC. However, the maximum memory available on a SIC/XE system is 1 megabyte (2^{20} bytes). This increase leads to a change in instruction formats and addressing modes.

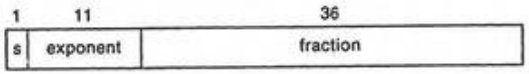
Registers

The following additional registers are provided by SIC/XE:

Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register—no special use
T	5	General working register—no special use
F	6	Floating-point accumulator (48 bits)

Data Formats

SIC/XE provides the same data formats as the standard version. In addition, there is a 48-bit floating-point data type with the following format:



The fraction is interpreted as a value between 0 and 1; that is, the assumed binary point is immediately before the high-order bit. For normalized floating-point numbers, the high-order bit of the fraction must be 1. The exponent is interpreted as an unsigned binary number between 0 and 2047. If the exponent has value e and the fraction has value f, the absolute value of the number represented is

$$f * 2^{(e-1024)}.$$

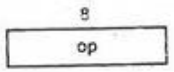
The sign of the floating-point number is indicated by the value of s (0 = positive, 1 = negative). A value of zero is represented by setting all bits (including sign, exponent, and fraction) to 0.

Instruction Formats

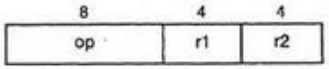
The larger memory available on SIC/XE means that an address will (in general) no longer fit into a 15-bit field; thus the instruction format used on the standard version of SIC is no longer suitable. There are two possible options—either use some form of relative addressing, or extend the address field to 20 bits. Both of these options are included in SIC/XE (Formats 3 and 4 in the following description). In addition, SIC/XE provides some instructions that do not reference memory at all. Formats 1 and 2 in the following description are used for such instructions.

The new set of instruction formats is as follows. The settings of the flag bits in Formats 3 and 4 are discussed under Addressing Modes. Bit e is used to distinguish between Formats 3 and 4 (e = 0 means Format 3, e = 1 means Format 4). Appendix A indicates the format to be used with each machine instruction.

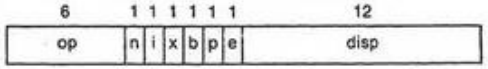
Format 1 (1 byte):



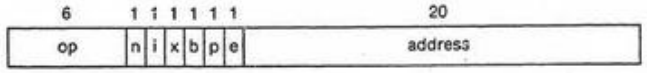
Format 2 (2 bytes):



Format 3 (3 bytes):



Format 4 (4 bytes):



Addressing Modes

Two new relative addressing modes are available for use with instructions assembled using Format 3. These are described in the following table:

Mode	Indication	Target address calculation	
Base relative	b = 1, p = 0	TA = (B) + disp	(0 ≤ disp ≤ 4095)
Program-counter relative	b = 0, p = 1	TA = (PC) + disp	(−2048 ≤ disp ≤ 2047)

For *base relative* addressing, the displacement field *disp* in a Format 3 instruction is interpreted as a 12-bit unsigned integer. For *program-counter relative* addressing, this field is interpreted as a 12-bit signed integer, with negative values represented in 2's complement notation.

If bits b and p are both set to 0, the *disp* field from the Format 3 instruction is taken to be the target address. For a Format 4 instruction, bits b and p are normally set to 0, and the target address is taken from the address field of the instruction. We will call this *direct* addressing, to distinguish it from the relative addressing modes described above.

Any of these addressing modes can also be combined with *indexed* addressing—if bit x is set to 1, the term (X) is added in the target address calculation. Notice that the standard version of the SIC machine uses only direct addressing (with or without indexing).

Bits i and n in Formats 3 and 4 are used to specify how the target address is used. If bit $i = 1$ and $n = 0$, the target address itself is used as the operand value; no memory reference is performed. This is called *immediate* addressing. If bit $i = 0$ and $n = 1$, the word at the location given by the target address is fetched; the *value* contained in this word is then taken as the *address* of the operand value. This is called *indirect* addressing. If bits i and n are both 0 or both 1, the target address is taken as the location of the operand; we will refer to this as *simple* addressing. Indexing cannot be used with immediate or indirect addressing modes.

Many authors use the term *effective address* to denote what we have called the target address for an instruction. However, there is disagreement concerning the meaning of effective address when referring to an instruction that uses indirect addressing. To avoid confusion, we use the term *target address* throughout this book.

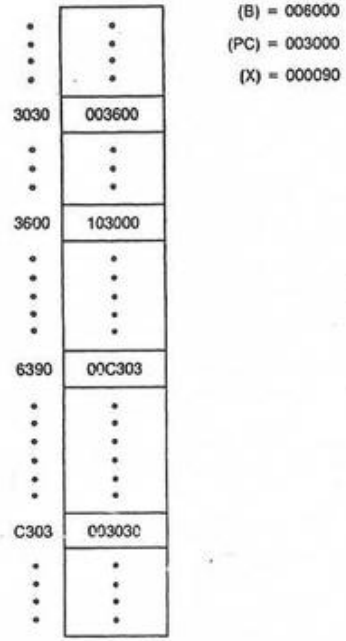
SIC/XE instructions that specify neither immediate nor indirect addressing are assembled with bits n and i both set to 1. Assemblers for the standard version of SIC will, however, set the bits in both of these positions to 0. (This is because the 8-bit binary codes for all of the SIC instructions end in 00.) All SIC/XE machines have a special hardware feature designed to provide the upward compatibility mentioned earlier. If bits n and i are both 0, then bits b, p, and e are considered to be part of the address field of the instruction (rather than flags indicating addressing modes). This makes Instruction Format 3 identical to the format used on the standard version of SIC, providing the desired compatibility.

Figure 1.1 gives examples of the different addressing modes available on SIC/XE. Figure 1.1(a) shows the contents of registers B, PC, and X, and of selected memory locations. (All values are given in hexadecimal.) Figure 1.1(b) gives the machine code for a series of LDA instructions. The target address generated by each instruction, and the value that is loaded into register A, are also shown. You should carefully examine these examples, being sure you understand the different addressing modes illustrated.

For ease of reference, all of the SIC/XE instruction formats and addressing modes are summarized in Appendix A.

Instruction Set

SIC/XE provides all of the instructions that are available on the standard version. In addition, there are instructions to load and store the new registers (LDB, STB, etc.) and to perform floating-point arithmetic operations (ADDF,



(a)

| Machine instruction | | |

SUBF, MULF, DIVF). There are also instructions that take their operands from registers. Besides the RMO (register move) instruction, these include register-to-register arithmetic operations (ADDR, SUBR, MULR, DIVR). A special *supervisor call* instruction (SVC) is provided. Executing this instruction generates an interrupt that can be used for communication with the operating system. (Supervisor calls and interrupts are discussed in Chapter 6.)

There are also several other new instructions. Appendix A gives a complete list of all SIC/XE instructions, with their operation codes and a specification of the function performed by each.

Input and Output

The I/O instructions we discussed for SIC are also available on SIC/XE. In addition, there are I/O channels that can be used to perform input and output while the CPU is executing other instructions. This allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test, and halt the operation of I/O channels. (These concepts are discussed in detail in Chapter 6.)

1.3.3 SIC Programming Examples

This section presents simple examples of SIC and SIC/XE assembler language programming. These examples are intended to help you become more familiar with the SIC and SIC/XE instruction sets and assembler language. It is assumed that the reader is already familiar with the assembler language of at least one machine and with the basic ideas involved in assembly-level programming.

The primary subject of this book is systems programming, not assembler language programming. The following chapters contain discussions of various types of system software, and in some cases SIC programs are used to illustrate the points being made. This section contains material that may help you to understand these examples more easily. However, it does not contain any new material on system software or systems programming. Thus, this section can be skipped without any loss of continuity.

Figure 1.2 contains examples of data movement operations for SIC and SIC/XE. There are no memory-to-memory move instructions; thus, all data movement must be done using registers. Figure 1.2(a) shows two examples of data movement. In the first, a 3-byte word is moved by loading it into register A and then storing the register at the desired destination. Exactly the same thing could be accomplished using register X (and the instructions LDX, STX) or register L (LDL, STL). In the second example, a single byte of data is moved using the instructions LDCH (Load Character) and STCH (Store Character).

These instructions operate by loading or storing the rightmost 8-bit byte of register A; the other bits in register A are not affected.

Figure 1.2(a) also shows four different ways of defining storage for data items in the SIC assembler language. (These assembler directives are discussed in more detail in Section 2.1.) The statement WORD reserves one word of storage, which is initialized to a value defined in the operand field of the statement. Thus the WORD statement in Fig. 1.2(a) defines a data word labeled FIVE whose value is initialized to 5. The statement RESW reserves one or more words of storage for use by the program. For example, the RESW statement in Fig. 1.2(a) defines one word of storage labeled ALPHA, which will be used to hold a value generated by the program.

The statements BYTE and RESB perform similar storage-definition functions for data items that are characters instead of words. Thus in Fig. 1.2(a) CHARZ is a 1-byte data item whose value is initialized to the character "Z", and C1 is a 1-byte variable with no initial value.

```
        LDA     FIVE        LOAD CONSTANT 5 INTO REGISTER A
        STA     ALPHA       STORE IN ALPHA
        LDCH    CHARZ       LOAD CHARACTER 'Z' INTO REGISTER A
        STCH    C1          STORE IN CHARACTER VARIABLE C1
         .
         .
         .
ALPHA   RESW    1           ONE-WORD VARIABLE
FIVE    WORD    5           ONE-WORD CONSTANT
CHARZ   BYTE    C'Z'        ONE-BYTE CONSTANT
C1      RESB    1           ONE-BYTE VARIABLE
```

(a)

```
        LDA     #5          LOAD VALUE 5 INTO REGISTER A
        STA     ALPHA       STORE IN ALPHA
        LDA     #90         LOAD ASCII CODE FOR 'Z' INTO REG A
        STCH    C1          STORE IN CHARACTER VARIABLE C1
         .
         .
         .
ALPHA   RESW    1           ONE-WORD VARIABLE
C1      RESB    1           ONE-BYTE VARIABLE
```

(b)

Figure 1.2 Sample data movement operations for (a) SIC and (b) SIC/XE.

The instructions shown in Fig. 1.2(a) would also work on SIC/XE; however, they would not take advantage of the more advanced hardware features available. Figure 1.2(b) shows the same two data-movement operations as they might be written for SIC/XE. In this example, the value 5 is loaded into register A using immediate addressing. The operand field for this instruction contains the flag # (which specifies immediate addressing) and the data value to be loaded. Similarly, the character "Z" is placed into register A by using immediate addressing to load the value 90, which is the decimal value of the ASCII code that is used internally to represent the character "Z".

Figure 1.3(a) shows examples of arithmetic instructions for SIC. All arithmetic operations are performed using register A, with the result being left in register A. Thus this sequence of instructions stores the value (ALPHA + INCR – 1) in BETA and the value (GAMMA + INCR – 1) in DELTA.

Figure 1.3(b) illustrates how the same calculations could be performed on SIC/XE. The value of INCR is loaded into register S initially, and the register-to-register instruction ADDR is used to add this value to register A when it is needed. This avoids having to fetch INCR from memory each time it is used in a calculation, which may make the program more efficient. Immediate addressing is used for the constant 1 in the subtraction operations.

Looping and indexing operations are illustrated in Fig. 1.4. Figure 1.4(a) shows a loop that copies one 11-byte character string to another. The index register (register X) is initialized to zero before the loop begins. Thus, during the first execution of the loop, the target address for the LDCH instruction will be the address of the first byte of STR1. Similarly, the STCH instruction will store the character being copied into the first byte of STR2. The next instruction, TIX, performs two functions. First it adds 1 to the value in register X, and then it compares the new value of register X to the value of the operand (in this case, the constant value 11). The condition code is set to indicate the result of this comparison. The JLT instruction jumps if the condition code is set to "less than." Thus, the JLT causes a jump back to the beginning of the loop if the new value in register X is less than 11.

During the second execution of the loop, register X will contain the value 1. Thus, the target address for the LDCH instruction will be the second byte of STR1, and the target address for the STCH instruction will be the second byte of STR2. The TIX instruction will again add 1 to the value in register X, and the loop will continue in this way until all 11 bytes have been copied from STR1 to STR2. Notice that after the TIX instruction is executed, the value in register X is equal to the number of bytes that have already been copied.

Figure 1.4(b) shows the same loop as it might be written for SIC/XE. The main difference is that the instruction TIXR is used in place of TIX. TIXR works exactly like TIX, except that the value used for comparison is taken from another register (in this case, register T), not from memory. This makes

the loop more efficient, because the value does not have to be fetched from memory each time the loop is executed. Immediate addressing is used to initialize register T to the value 11 and to initialize register X to 0.

```
        LDA     ALPHA       LOAD ALPHA INTO REGISTER A
        ADD     INCR        ADD THE VALUE OF INCR
        SUB     ONE         SUBTRACT 1
        STA     BETA        STORE IN BETA
        LDA     GAMMA       LOAD GAMMA INTO REGISTER A
        ADD     INCR        ADD THE VALUE OF INCR
        SUB     ONE         SUBTRACT 1
        STA     DELTA       STORE IN DELTA
        .
        .
        .
ONE     WORD    1           ONE-WORD CONSTANT
.                           ONE-WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

(a)

```
        LDS     INCR        LOAD VALUE OF INCR INTO REGISTER S
        LDA     ALPHA       LOAD ALPHA INTO REGISTER A
        ADDR    S,A         ADD THE VALUE OF INCR
        SUB     #1          SUBTRACT 1
        STA     BETA        STORE IN BETA
        LDA     GAMMA       LOAD GAMMA INTO REGISTER A
        ADDR    S,A         ADD THE VALUE OF INCR
        SUB     #1          SUBTRACT 1
        STA     DELTA       STORE IN DELTA
        .
        .
        .
.                           ONE WORD VARIABLES
ALPHA   RESW    1
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

(b)

Figure 1.3 Sample arithmetic operations for (a) SIC and (b) SIC/XE.

```
        LDX     ZERO        INITIALIZE INDEX REGISTER TO 0
MOVECH  LDCH    STR1,X      LOAD CHARACTER FROM STR1 INTO REG A
        STCH    STR2,X      STORE CHARACTER INTO STR2
        TIX     ELEVEN      ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT     MOVECH      LOOP IF INDEX IS LESS THAN 11
        .
        .
        .
STR1    BYTE    C'TEST STRING'   11-BYTE STRING CONSTANT
STR2    RESB    11               11-BYTE VARIABLE
.                                ONE-WORD CONSTANTS
ZERO    WORD    0
ELEVEN  WORD    11
```

(a)

```
        LDT     #11         INITIALIZE REGISTER T TO 11
        LDX     #0          INITIALIZE INDEX REGISTER TO 0
MOVECH  LDCH    STR1,X      LOAD CHARACTER FROM STR1 INTO REG A
        STCH    STR2,X      STORE CHARACTER INTO STR2
        TIXR    T           ADD 1 TO INDEX, COMPARE RESULT TO 11
        JLT     MOVECH      LOOP IF INDEX IS LESS THAN 11
        .
        .
        .
STR1    BYTE    C'TEST STRING'   11-BYTE STRING CONSTANT
STR2    RESB    11               11-BYTE VARIABLE
```

(b)

Figure 1.4 Sample looping and indexing operations for (a) SIC and (b) SIC/XE.

Figure 1.5 contains another example of looping and indexing operations. The variables ALPHA, BETA, and GAMMA are arrays of 100 words each. In this case, the task of the loop is to add together the corresponding elements of ALPHA and BETA, storing the results in the elements of GAMMA. The general principles of looping and indexing are the same as previously discussed. However, the value in the index register must be incremented by 3 for each iteration of this loop, because each iteration processes a 3-byte (i.e., one-word) element of the arrays. The TIX instruction always adds 1 to register X, so it is not suitable for this program fragment. Instead, we use arithmetic and comparison instructions to handle the index value.


```
        LDA     ZERO        INITIALIZE INDEX VALUE TO 0
        STA     INDEX
ADDLP   LDX     INDEX       LOAD INDEX VALUE INTO REGISTER X
        LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        LDA     INDEX       ADD 3 TO INDEX VALUE
        ADD     THREE
        STA     INDEX
        COMP    K300        COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX IS LESS THAN 300
        .
        .
        .
INDEX   RESW    1           ONE-WORD VARIABLE FOR INDEX VALUE
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
.                           ONE-WORD CONSTANTS
ZERO    WORD    0
K300    WORD    300
THREE   WORD    3
```

(a)

```
        LDS     #3          INITIALIZE REGISTER S TO 3
        LDT     #300        INITIALIZE REGISTER T TO 300
        LDX     #0          INITIALIZE INDEX REGISTER TO 0
ADDLP   LDA     ALPHA,X     LOAD WORD FROM ALPHA INTO REGISTER A
        ADD     BETA,X      ADD WORD FROM BETA
        STA     GAMMA,X     STORE THE RESULT IN A WORD IN GAMMA
        ADDR    S,X         ADD 3 TO INDEX VALUE
        COMPR   X,T         COMPARE NEW INDEX VALUE TO 300
        JLT     ADDLP       LOOP IF INDEX VALUE IS LESS THAN 300
        .
        .
        .
.                           ARRAY VARIABLES--100 WORDS EACH
ALPHA   RESW    100
BETA    RESW    100
GAMMA   RESW    100
```

(b)

Figure 1.5 Sample indexing and looping operations for (a) SIC and (b) SIC/XE.

In Fig. 1.5(a), we define a variable INDEX that holds the value to be used for indexing for each iteration of the loop. Thus, INDEX should be 0 for the first iteration, 3 for the second, and so on. INDEX is initialized to 0 before the start of the loop. The first instruction in the body of the loop loads the current value of INDEX into register X so that it can be used for target address calculation. The next three instructions in the loop load a word from ALPHA, add the corresponding word from BETA, and store the result in the corresponding word of GAMMA. The value of INDEX is then loaded into register A, incremented by 3, and stored back into INDEX. After being stored, the new value of INDEX is still present in register A. This value is then compared to 300 (the length of the arrays in bytes) to determine whether or not to terminate the loop. If the value of INDEX is less than 300, then all bytes of the arrays have not yet been processed. In that case, the JLT instruction causes a jump back to the beginning of the loop, where the new value of INDEX is loaded into register X.

This particular loop is cumbersome on SIC, because register A must be used for adding the array elements together and also for incrementing the index value. The loop can be written much more efficiently for SIC/XE, as shown in Fig. 1.5(b). In this example, the index value is kept permanently in register X. The amount by which to increment the index value (3) is kept in register S, and the register-to-register ADDR instruction is used to add this increment to register X. Similarly, the value 300 is kept in register T, and the instruction COMPR is used to compare registers X and T in order to decide when to terminate the loop.

Figure 1.6 shows a simple example of input and output on SIC; the same instructions would also work on SIC/XE. (The more advanced input and output facilities available on SIC/XE, such as I/O channels and interrupts, are discussed in Chapter 6.) This program fragment reads 1 byte of data from device F1 and copies it to device 05. The actual input of data is performed using the RD (Read Data) instruction. The operand for the RD is a byte in memory that contains the hexadecimal code for the input device (in this case, F1). Executing the RD instruction transfers 1 byte of data from this device into the rightmost byte of register A. If the input device is character-oriented (for example, a keyboard), the value placed in register A is the ASCII code for the character that was read.

Before the RD can be executed, however, the input device must be ready to transmit the data. For example, if the input device is a keyboard, the operator must have typed a character. The program checks for this by using the TD (Test Device) instruction. When the TD is executed, the status of the addressed device is tested and the condition code is set to indicate the result of this test. If the device is ready to transmit data, the condition code is set to "less than"; if the device is not ready, the condition code is set to "equal." As Fig. 1.6


```
INLOOP   TD      INDEV        TEST INPUT DEVICE
         JEQ     INLOOP       LOOP UNTIL DEVICE IS READY
         RD      INDEV        READ ONE BYTE INTO REGISTER A
         STCH    DATA         STORE BYTE THAT WAS READ
         .
         .
         .
OUTLP    TD      OUTDEV       TEST OUTPUT DEVICE
         JEQ     OUTLP        LOOP UNTIL DEVICE IS READY
         LDCH    DATA         LOAD DATA BYTE INTO REGISTER A
         WD      OUTDEV       WRITE ONE BYTE TO OUTPUT DEVICE
         .
         .
         .
INDEV    BYTE    X'F1'        INPUT DEVICE NUMBER
OUTDEV   BYTE    X'05'        OUTPUT DEVICE NUMBER
DATA     RESB    1            ONE-BYTE VARIABLE
```

Figure 1.6 Sample input and output operations for SIC.

illustrates, the program must execute the TD instruction and then check the condition code by using a conditional jump. If the condition code is "equal" (device not ready), the program jumps back to the TD instruction. This two-instruction loop will continue until the device becomes ready; then the RD will be executed.

Output is performed in the same way. First the program uses TD to check whether the output device is ready to receive a byte of data. Then the byte to be written is loaded into the rightmost byte of register A, and the WD (Write Data) instruction is used to transmit it to the device.

Figure 1.7 shows how these instructions can be used to read a 100-byte record from an input device into memory. The read operation in this example is placed in a subroutine. This subroutine is called from the main program by using the JSUB (Jump to Subroutine) instruction. At the end of the subroutine there is an RSUB (Return from Subroutine) instruction, which returns control to the instruction that follows the JSUB.

The READ subroutine itself consists of a loop. Each execution of this loop reads 1 byte of data from the input device, using the same techniques illustrated in Fig. 1.6. The bytes of data that are read are stored in a 100-byte buffer area labeled RECORD. The indexing and looping techniques that are used in storing characters in this buffer are essentially the same as those illustrated in Fig. 1.4(a).

Figure 1.7(b) shows the same READ subroutine as it might be written for SIC/XE. The main differences from Fig. 1.7(a) are the use of immediate addressing and the TIXR instruction, as was illustrated in Fig. 1.4(a).

```
        JSUB    READ        CALL READ SUBROUTINE
        .
        .
        .
                            SUBROUTINE TO READ 100-BYTE RECORD
.
READ    LDX     ZERO        INITIALIZE INDEX REGISTER TO 0
RLOOP   TD      INDEV       TEST INPUT DEVICE
        JEQ     RLOOP       LOOP IF DEVICE IS BUSY
        RD      INDEV       READ ONE BYTE INTO REGISTER A
        STCH    RECORD,X    STORE DATA BYTE INTO RECORD
        TIX     K100        ADD 1 TO INDEX AND COMPARE TO 100
        JLT     RLOOP       LOOP IF INDEX IS LESS THAN 100
        RSUB                EXIT FROM SUBROUTINE
        .
        .
        .
INDEV   BYTE    X'F1'       INPUT DEVICE NUMBER
RECORD  RESB    100         100-BYTE BUFFER FOR INPUT RECORD
.                           ONE-WORD CONSTANTS
ZERO    WORD    0
K100    WORD    100
```

(a)

```
        JSUB    READ        CALL READ SUBROUTINE
        .
        .
        .
                            SUBROUTINE TO READ 100-BYTE RECORD
.
READ    LDX     #0          INITIALIZE INDEX REGISTER TO 0
        LDT     #100        INITIALIZE REGISTER T TO 100
RLOOP   TD      INDEV       TEST INPUT DEVICE
        JEQ     RLOOP       LOOP IF DEVICE IS BUSY
        RD      INDEV       READ ONE BYTE INTO REGISTER A
        STCH    RECORD,X    STORE DATA BYTE INTO RECORD
        TIXR    T           ADD 1 TO INDEX AND COMPARE TO 100
        JLT     RLOOP       LOOP IF INDEX IS LESS THAN 100
        RSUB                EXIT FROM SUBROUTINE
        .
        .
        .
INDEV   BYTE    X'F1'       INPUT DEVICE NUMBER
RECORD  RESB    100         100-BYTE BUFFER FOR INPUT RECORD
```

(b)

Figure 1.7 Sample subroutine call and record input operations for (a) SIC and (b) SIC/XE.

1.4 TRADITIONAL (CISC) MACHINES

This section introduces the architectures of two of the machines that will be used as examples later in the text. Section 1.4.1 describes the VAX architecture, and Section 1.4.2 describes the architecture of the Intel x86 family of processors.

The machines described in this section are classified as Complex Instruction Set Computers (CISC). CISC machines generally have a relatively large and complicated instruction set, several different instruction formats and lengths, and many different addressing modes. Thus the implementation of such an architecture in hardware tends to be complex.

You may want to compare the examples in this section with the Reduced Instruction Set Computer (RISC) examples in Section 1.5. Further discussion of CISC versus RISC designs can be found in Tabak (1995).

1.4.1 VAX Architecture

The VAX family of computers was introduced by Digital Equipment Corporation (DEC) in 1978. The VAX architecture was designed for compatibility with the earlier PDP-11 machines. A compatibility mode was provided at the hardware level so that many PDP-11 programs could run unchanged on the VAX. It was even possible for PDP-11 programs and VAX programs to share the same machine in a multi-user environment.

This section summarizes some of the main characteristics of the VAX architecture. For further information, see Baase (1992).

Memory

The VAX memory consists of 8-bit bytes. All addresses used are byte addresses. Two consecutive bytes form a *word;* four bytes form a *longword;* eight bytes form a *quadword;* sixteen bytes form an *octaword.* Some operations are more efficient when operands are aligned in a particular way—for example, a longword operand that begins at a byte address that is a multiple of 4.

All VAX programs operate in a *virtual address space* of 2^{32} bytes. This virtual memory allows programs to operate as though they had access to an extremely large memory, regardless of the amount of memory actually present on the system. Routines in the operating system take care of the details of memory management. We discuss virtual memory in connection with our study of operating systems in Chapter 6. One half of the VAX virtual address space is called *system space,* which contains the operating system, and is shared by all programs. The other half of the address space is called *process space,* and

is defined separately for each program. A part of the process space contains stacks that are available to the program. Special registers and machine instructions aid in the use of these stacks.

Registers

There are 16 general-purpose registers on the VAX, denoted by R0 through R15. Some of these registers, however, have special names and uses. All general registers are 32 bits in length. Register R15 is the *program counter*, also called PC. It is updated during instruction execution to point to the next instruction byte to be fetched. R14 is the *stack pointer* SP, which points to the current top of the stack in the program's process space. Although it is possible to use other registers for this purpose, hardware instructions that implicitly use the stack always use SP. R13 is the *frame pointer* FP. VAX procedure call conventions build a data structure called a stack frame, and place its address in FP. R12 is the *argument pointer* AP. The procedure call convention uses AP to pass a list of arguments associated with the call.

Registers R6 through R11 have no special functions, and are available for general use by the program. Registers R0 through R5 are likewise available for general use; however, these registers are also used by some machine instructions.

In addition to the general registers, there is a *processor status longword* (PSL), which contains state variables and flags associated with a process. The PSL includes, among many other items of information, a condition code and a flag that specifies whether PDP-11 compatibility mode is being used by a process. There are also a number of control registers that are used to support various operating system functions.

Data Formats

Integers are stored as binary numbers in a byte, word, longword, quadword, or octaword; 2's complement representation is used for negative values. Characters are stored using their 8-bit ASCII codes.

There are four different floating-point data formats on the VAX, ranging in length from 4 to 16 bytes. Two of these are compatible with those found on the PDP-11, and are standard on all VAX processors. The other two are available as options, and provide for an extended range of values by allowing more bits in the exponent field. In each case, the principles are the same as those we discussed for SIC/XE: a floating-point value is represented as a fraction that is to be multiplied by a specified power of 2.

VAX processors provide a *packed decimal* data format. In this format, each byte represents two decimal digits, with each digit encoded using 4 bits of the byte. The sign is encoded in the last 4 bits. There is also a *numeric* format that

is used to represent numeric values with one digit per byte. In this format, the sign may appear either in the last byte, or as a separate byte preceding the first digit. These two variations are called *trailing numeric* and *leading separate numeric*.

VAX also supports queues and variable-length bit strings. Data structures such as these can, of course, be implemented on any machine; however, VAX provides direct hardware support for them. There are single machine instructions that insert and remove entries in queues, and perform a variety of operations on bit strings. The existence of such powerful machine instructions and complex primitive data types is one of the more unusual features of the VAX architecture.

Instruction Formats

VAX machine instructions use a variable-length instruction format. Each instruction consists of an operation code (1 or 2 bytes) followed by up to six *operand specifiers*, depending on the type of instruction. Each operand specifier designates one of the VAX addressing modes and gives any additional information necessary to locate the operand. (See the description of addressing modes in the following section for further information.)

Addressing Modes

VAX provides a large number of addressing modes. With few exceptions, any of these addressing modes may be used with any instruction. The operand itself may be in a register (*register* mode), or its address may be specified by a register (*register deferred* mode). If the operand address is in a register, the register contents may be automatically incremented or decremented by the operand length (*autoincrement* and *autodecrement* modes). There are several base relative addressing modes, with displacement fields of different lengths; when used with register PC, these become program-counter relative modes. All of these addressing modes may also include an index register, and many of them are available in a form that specifies indirect addressing (called *deferred* modes on VAX). In addition, there are immediate operands and several special-purpose addressing modes. For further details, see Baase (1992).

Instruction Set

One of the goals of the VAX designers was to produce an instruction set that is symmetric with respect to data type. Many instruction mnemonics are formed by combining the following elements:

1. a prefix that specifies the type of operation,
2. a suffix that specifies the data type of the operands,
3. a modifier (on some instructions) that gives the number of operands involved.

For example, the instruction ADDW2 is an add operation with two operands, each a word in length. Likewise, MULL3 is a multiply operation with three longword operands, and CVTWL specifies a conversion from word to longword. (In the latter case, a two-operand instruction is assumed.) For a typical instruction, operands may be located in registers, in memory, or in the instruction itself (immediate addressing). The same machine instruction code is used, regardless of operand locations.

VAX provides all of the usual types of instructions for computation, data movement and conversion, comparison, branching, etc. In addition, there are a number of operations that are much more complex than the machine instructions found on most computers. These operations are, for the most part, hardware realizations of frequently occurring sequences of code. They are implemented as single instructions for efficiency and speed. For example, VAX provides instructions to load and store multiple registers, and to manipulate queues and variable-length bit fields. There are also powerful instructions for calling and returning from procedures. A single instruction saves a designated set of registers, passes a list of arguments to the procedure, maintains the stack, frame, and argument pointers, and sets a mask to enable error traps for arithmetic operations. For further information on all of the VAX instructions, see Baase (1992).

Input and Output

Input and output on the VAX are accomplished by I/O device controllers. Each controller has a set of control/status and data registers, which are assigned locations in the physical address space. The portion of the address space into which the device controller registers are mapped is called *I/O space*.

No special instructions are required to access registers in I/O space. An I/O device driver issues commands to the device controller by storing values into the appropriate registers, exactly as if they were physical memory locations. Likewise, software routines may read these registers to obtain status information. The association of an address in I/O space with a physical register in a device controller is handled by the memory management routines.

1.4.2 Pentium Pro Architecture

The Pentium Pro microprocessor, introduced near the end of 1995, is the latest in the Intel x86 family. Other recent microprocessors in this family are the 80486 and Pentium. Processors of the x86 family are presently used in a majority of personal computers, and there is a vast amount of software for these processors. It is expected that additional generations of the x86 family will be developed in the future.

The various x86 processors differ in implementation details and operating speed. However, they share the same basic architecture. Each succeeding generation has been designed to be compatible with the earlier versions. This section contains an overview of the x86 architecture, which will serve as background for the examples to be discussed later in the book. Further information about the x86 family can be found in Intel (1995), Anderson and Shanley (1995), and Tabak (1995).

Memory

Memory in the x86 architecture can be described in at least two different ways. At the physical level, memory consists of 8-bit bytes. All addresses used are byte addresses. Two consecutive bytes form a *word;* four bytes form a *doubleword* (also called a dword). Some operations are more efficient when operands are aligned in a particular way—for example, a doubleword operand that begins at a byte address that is a multiple of 4.

However, programmers usually view the x86 memory as a collection of *segments*. From this point of view, an address consists of two parts—a segment number and an offset that points to a byte within the segment. Segments can be of different sizes, and are often used for different purposes. For example, some segments may contain executable instructions, and other segments may be used to store data. Some data segments may be treated as stacks that can be used to save register contents, pass parameters to subroutines, and for other purposes.

It is not necessary for all of the segments used by a program to be in physical memory. In some cases, a segment can also be divided into *pages*. Some of the pages of a segment may be in physical memory, while others may be stored on disk. When an x86 instruction is executed, the hardware and the operating system make sure that the needed byte of the segment is loaded into physical memory. The segment/offset address specified by the programmer is automatically translated into a physical byte address by the x86 Memory

Management Unit (MMU). Chapter 6 contains a brief discussion of methods that can be used in this kind of address translation.

Registers

There are eight general-purpose registers, which are named EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. Each general-purpose register is 32 bits long (i.e., one doubleword). Registers EAX, EBX, ECX, and EDX are generally used for data manipulation; it is possible to access individual words or bytes from these registers. The other four registers can also be used for data, but are more commonly used to hold addresses. The general-purpose register set is identical for all members of the x86 family beginning with the 80386. This set is also compatible with the more limited register sets found in earlier members of the family.

There are also several different types of special-purpose registers in the x86 architecture. EIP is a 32-bit register that contains a pointer to the next instruction to be executed. FLAGS is a 32-bit register that contains many different bit flags. Some of these flags indicate the status of the processor; others are used to record the results of comparisons and arithmetic operations. There are also six 16-bit *segment registers* that are used to locate segments in memory. Segment register CS contains the address of the currently executing code segment, and SS contains the address of the current stack segment. The other segment registers (DS, ES, FS, and GS) are used to indicate the addresses of data segments.

Floating-point computations are performed using a special *floating-point unit* (FPU). This unit contains eight 80-bit data registers and several other control and status registers.

All of the registers discussed so far are available to application programs. There are also a number of registers that are used only by system programs such as the operating system. Some of these registers are used by the MMU to translate segment addresses into physical addresses. Others are used to control the operation of the processor, or to support debugging operations.

Data Formats

The x86 architecture provides for the storage of integers, floating-point values, characters, and strings. Integers are normally stored as 8-, 16-, or 32-bit binary numbers. Both signed and unsigned integers (also called ordinals) are supported; 2's complement is used for negative values. The FPU can also handle 64-bit signed integers. In memory, the least significant part of a numeric value is stored at the lowest-numbered address. (This is commonly called *little-endian* byte ordering, because the "little end" of the value comes first in memory.)

Integers can also be stored in *binary coded decimal* (BCD). In the unpacked BCD format, each byte represents one decimal digit. The value of this digit is encoded (in binary) in the low-order 4 bits of the byte; the high-order bits are normally zero. In the packed BCD format, each byte represents two decimal digits, with each digit encoded using 4 bits of the byte.

There are three different floating-point data formats. The single-precision format is 32 bits long. It stores 24 significant bits of the floating-point value, and allows for a 7-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.) The double-precision format is 64 bits long. It stores 53 significant bits, and allows for a 10-bit exponent. The extended-precision format is 80 bits long. It stores 64 significant bits, and allows for a 15-bit exponent.

Characters are stored one per byte, using their 8-bit ASCII codes. Strings may consist of bits, bytes, words, or doublewords; special instructions are provided to handle each type of string.

Instruction Formats

All of the x86 machine instructions use variations of the same basic format. This format begins with optional prefixes containing flags that modify the operation of the instruction. For example, some prefixes specify a repetition count for an instruction. Others specify a segment register that is to be used for addressing an operand (overriding the normal default assumptions made by the hardware). Following the prefixes (if any) is an opcode (1 or 2 bytes); some operations have different opcodes, each specifying a different variant of the operation. Following the opcode are a number of bytes that specify the operands and addressing modes to be used. (See the description of addressing modes in the next section for further information.)

The opcode is the only element that is always present in every instruction. Other elements may or may not be present, and may be of different lengths, depending on the operation and the operands involved. Thus, there are a large number of different potential instruction formats, varying in length from 1 byte to 10 bytes or more.

Addressing Modes

The x86 architecture provides a large number of addressing modes. An operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register* mode).

Operands stored in memory are often specified using variations of the general target address calculation

TA = (base register) + (index register) * (scale factor) + displacement

Any general-purpose register may be used as a base register; any general-purpose register except ESP can be used as an index register. The scale factor may have the value 1, 2, 4, or 8, and the displacement may be an 8-, 16-, or 32-bit value. The base and index register numbers, scale, and displacement are encoded as parts of the operand specifiers in the instruction. Various combinations of these items may be omitted, resulting in eight different addressing modes. The address of an operand in memory may also be specified as an absolute location (*direct* mode), or as a location relative to the EIP register (*relative* mode).

Instruction Set

The x86 architecture has a large and complex instruction set, containing more than 400 different machine instructions. An instruction may have zero, one, two, or three operands. There are register-to-register instructions, register-to-memory instructions, and a few memory-to-memory instructions. In some cases, operands may also be specified in the instruction as immediate values.

Most data movement and integer arithmetic instructions can use operands that are 1, 2, or 4 bytes long. String manipulation instructions, which use repetition prefixes, can deal directly with variable-length strings of bytes, words, or doublewords. There are many instructions that perform logical and bit manipulations, and support control of the processor and memory-management systems.

The x86 architecture also includes special-purpose instructions to perform operations frequently required in high-level programming languages—for example, entering and leaving procedures and checking subscript values against the bounds of an array.

Input and Output

Input is performed by instructions that transfer one byte, word, or doubleword at a time from an I/O port into register EAX. Output instructions transfer one byte, word, or doubleword from EAX to an I/O port. Repetition prefixes allow these instructions to transfer an entire string in a single operation.

1.5 RISC MACHINES

This section introduces the architectures of three RISC machines that will be used as examples later in the text. Section 1.5.1 describes the architecture of the SPARC family of processors. Section 1.5.2 describes the PowerPC family of microprocessors for personal computers. Section 1.5.3 describes the architecture of the Cray T3E supercomputing system.

All of these machines are examples of RISC (Reduced Instruction Set Computers), in contrast to traditional CISC (Complex Instruction Set Computer) implementations such as Pentium and VAX. The RISC concept, developed in the early 1980s, was intended to simplify the design of processors. This simplified design can result in faster and less expensive processor development, greater reliability, and faster instruction execution times.

In general, a RISC system is characterized by a standard, fixed instruction length (usually equal to one machine word), and single-cycle execution of most instructions. Memory access is usually done by load and store instructions only. All instructions except for load and store are register-to-register operations. There are typically a relatively large number of general-purpose registers. The number of machine instructions, instruction formats, and addressing modes is relatively small.

The discussions in the following sections will illustrate some of these RISC characteristics. Further information about the RISC approach, including its advantages and disadvantages, can be found in Tabak (1995).

1.5.1 UltraSPARC Architecture

The UltraSPARC processor, announced by Sun Microsystems in 1995, is the latest member of the SPARC family. Other members of this family include a variety of SPARC and SuperSPARC processors. The original SPARC architecture was developed in the mid-1980s, and has been implemented by a number of manufacturers. The name SPARC stands for scalable processor architecture. This architecture is intended to be suitable for a wide range of implementations, from microcomputers to supercomputers.

Although SPARC, SuperSPARC, and UltraSPARC architectures differ slightly, they are upward compatible and share the same basic structure. This section contains an overview of the UltraSPARC architecture, which will serve as background for the examples to be discussed later in the book. Further information about the SPARC family can be found in Tabak (1995) and Sun Microsystems (1995a).

Memory

Memory consists of 8-bit bytes; all addresses used are byte addresses. Two consecutive bytes form a *halfword;* four bytes form a *word;* eight bytes form a *doubleword*. Halfwords are stored in memory beginning at byte addresses that are multiples of 2. Similarly, words begin at addresses that are multiples of 4, and doublewords at addresses that are multiples of 8.

UltraSPARC programs can be written using a virtual address space of 2^{64} bytes. This address space is divided into *pages;* multiple page sizes are supported. Some of the pages used by a program may be in physical memory, while others may be stored on disk. When an instruction is executed, the hardware and the operating system make sure that the needed page is loaded into physical memory. The virtual address specified by the instruction is automatically translated into a physical address by the UltraSPARC Memory Management Unit (MMU). Chapter 6 contains a brief discussion of methods that can be used in this kind of address translation.

Registers

The SPARC architecture includes a large *register file* that usually contains more than 100 general-purpose registers. (The exact number varies from one implementation to another.) However, any procedure can access only 32 registers, designated r0 through r31. The first eight of these registers (r0 through r7) are global—that is, they can be accessed by all procedures on the system. (Register r0 always contains the value zero.)

The other 24 registers available to a procedure can be visualized as a *window* through which part of the register file can be seen. These windows overlap, so some registers in the register file are shared between procedures. For example, registers r8 through r15 of a calling procedure are physically the same registers as r24 through r31 of the called procedure. This facilitates the passing of parameters.

The SPARC hardware manages the windows into the register file. If a set of concurrently running procedures needs more windows than are physically available, a "window overflow" interrupt occurs. The operating system must then save the contents of some registers in the file (and restore them later) to provide the additional windows that are needed.

In the original SPARC architecture, the general-purpose registers were 32 bits long. Later implementations (including UltraSPARC) expanded these registers to 64 bits. Some SPARC implementations provide several physically different sets of global registers, for use by application procedures and by various hardware and operating system functions.

Floating-point computations are performed using a special *floating-point unit* (FPU). On UltraSPARC, this unit contains a file of 64 double-precision floating-point registers, and several other control and status registers.

Besides these register files, there are a program counter PC (which contains the address of the next instruction to be executed), condition code registers, and a number of other control registers.

Data Formats

The UltraSPARC architecture provides for the storage of integers, floating-point values, and characters. Integers are stored as 8-, 16-, 32-, or 64-bit binary numbers. Both signed and unsigned integers are supported; 2's complement is used for negative values. In the original SPARC architecture, the most significant part of a numeric value is stored at the lowest-numbered address. (This is commonly called *big-endian* byte ordering, because the "big end" of the value comes first in memory.) UltraSPARC supports both big-endian and little-endian byte orderings.

There are three different floating-point data formats. The single-precision format is 32 bits long. It stores 23 significant bits of the floating-point value, and allows for an 8-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.) The double-precision format is 64 bits long. It stores 52 significant bits, and allows for a 11-bit exponent. The quad-precision format stores 63 significant bits, and allows for a 15-bit exponent.

Characters are stored one per byte, using their 8-bit ASCII codes.

Instruction Formats

There are three basic instruction formats in the SPARC architecture. All of these formats are 32 bits long; the first 2 bits of the instruction word identify which format is being used. Format 1 is used for the Call instruction. Format 2 is used for branch instructions (and one special instruction that enters a value into a register). The remaining instructions use Format 3, which provides for register loads and stores, and three-operand arithmetic operations.

The fixed instruction length in the SPARC architecture is typical of RISC systems, and is intended to speed the process of instruction fetching and decoding. Compare this approach with the complex variable-length instructions found on CISC systems such as VAX and x86.

Addressing Modes

As in most architectures, an operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register direct* mode). Operands in memory are addressed using one of the following three modes:

Mode	Target address calculation
PC-relative	TA = (PC) + displacement {30 bits, signed}
Register indirect with displacement	TA = (register) + displacement {13 bits, signed}
Register indirect indexed	TA = (register-1) + (register-2)

PC-relative mode is used only for branch instructions.

The relatively few addressing modes of SPARC allow for more efficient implementations than the 10 or more modes found on CISC systems such as x86.

Instruction Set

The basic SPARC architecture has fewer than 100 machine instructions, reflecting its RISC philosophy. (Compare this with the 300 to 400 instructions often found in CISC systems.) The only instructions that access memory are loads and stores. All other instructions are register-to-register operations.

Instruction execution on a SPARC system is *pipelined*—while one instruction is being executed, the next one is being fetched from memory and decoded. In most cases, this technique speeds instruction execution. However, an ordinary branch instruction might cause the process to "stall." The instruction following the branch (which had already been fetched and decoded) would have to be discarded without being executed.

To make the pipeline work more efficiently, SPARC branch instructions (including subroutine calls) are *delayed branches*. This means that the instruction immediately following the branch instruction is actually executed *before* the branch is taken. For example, in the instruction sequence

```
SUB     %L0, 11, %L1
BA      NEXT
MOV     %L1, %O3
```

the MOV instruction is executed before the branch BA. This MOV instruction is said to be in the *delay slot* of the branch. The programmer must take this characteristic into account when writing an assembler language program. Further discussions and examples of the use of delayed branches can be found in Section 2.5.2.

The UltraSPARC architecture also includes special-purpose instructions to provide support for operating systems and optimizing compilers. For example, high-bandwidth block load and store operations can be used to speed

common operating system functions. Communication in a multi-processor system is facilitated by special "atomic" instructions that can execute without allowing other memory accesses to intervene. Conditional move instructions may allow a compiler to eliminate many branch instructions in order to optimize program execution.

Input and Output

In the SPARC architecture, communication with I/O devices is accomplished through memory. A range of memory locations is logically replaced by device registers. Each I/O device has a unique address, or set of addresses, assigned to it. When a load or store instruction refers to this device register area of memory, the corresponding device is activated. Thus input and output can be performed with the regular instruction set of the computer, and no special I/O instructions are needed.

1.5.2 PowerPC Architecture

IBM first introduced the POWER architecture early in 1990 with the RS/6000. (POWER is an acronym for Performance Optimization With Enhanced RISC.) It was soon realized that this architecture could form the basis for a new family of powerful and low-cost microprocessors. In October 1991, IBM, Apple, and Motorola formed an alliance to develop and market such microprocessors, which were named PowerPC. The first products using PowerPC chips were delivered near the end of 1993. Recent implementations of the PowerPC architecture include the PowerPC 601, 603, and 604; others are expected in the near future.

As its name implies, PowerPC is a RISC architecture. As we shall see, it has much in common with other RISC systems such as SPARC. There are also a few differences in philosophy, which we will note in the course of the discussion. This section contains an overview of the PowerPC architecture, which will serve as background for the examples to be discussed later in the book. Further information about PowerPC can be found in IBM (1994a) and Tabak (1995).

Memory

Memory consists of 8-bit bytes; all addresses used are byte addresses. Two consecutive bytes form a *halfword;* four bytes form a *word;* eight bytes form a *doubleword;* sixteen bytes form a *quadword.* Many instructions may execute

more efficiently if operands are aligned at a starting address that is a multiple of their length.

PowerPC programs can be written using a virtual address space of 2^{64} bytes. This address space is divided into fixed-length *segments*, which are 256 megabytes long. Each segment is divided into *pages*, which are 4096 bytes long. Some of the pages used by a program may be in physical memory, while others may be stored on disk. When an instruction is executed, the hardware and the operating system make sure that the needed page is loaded into physical memory. The virtual address specified by the instruction is automatically translated into a physical address. Chapter 6 contains a brief discussion of methods that can be used in this kind of address translation.

Registers

There are 32 general-purpose registers, designated GPR0 through GPR31. In the full PowerPC architecture, each register is 64 bits long. PowerPC can also be implemented in a 32-bit subset, which uses 32-bit registers. The general-purpose registers can be used to store and manipulate integer data and addresses.

Floating-point computations are performed using a special *floating-point unit* (FPU). This unit contains thirty-two 64-bit floating-point registers, and a status and control register.

A 32-bit condition register reflects the result of certain operations, and can be used as a mechanism for testing and branching. This register is divided into eight 4-bit subfields, named CR0 through CR7. These subfields can be set and tested individually by PowerPC instructions.

The PowerPC architecture includes a Link Register (LR) and a Count Register (CR), which are used by some branch instructions. There is also a Machine Status Register (MSR) and variety of other control and status registers, some of which are implementation dependent.

Data Formats

The PowerPC architecture provides for the storage of integers, floating-point values, and characters. Integers are stored as 8-, 16-, 32-, or 64-bit binary numbers. Both signed and unsigned integers are supported; 2's complement is used for negative values. By default, the most significant part of a numeric value is stored at the lowest-numbered address (big-endian byte ordering). It is possible to select little-endian byte ordering by setting a bit in a control register.

There are two different floating-point data formats. The single-precision format is 32 bits long. It stores 23 significant bits of the floating-point value, and allows for an 8-bit exponent (power of 2). (The remaining bit is used to store the sign of the floating-point value.) The double-precision format is 64 bits long. It stores 52 significant bits, and allows for a 11-bit exponent.

Characters are stored one per byte, using their 8-bit ASCII codes.

Instruction Formats

There are seven basic instruction formats in the PowerPC architecture, some of which have subforms. All of these formats are 32 bits long. Instructions must be aligned beginning at a word boundary (i.e., a byte address that is a multiple of 4). The first 6 bits of the instruction word always specify the opcode; some instruction formats also have an additional "extended opcode" field.

The fixed instruction length in the PowerPC architecture is typical of RISC systems. The variety and complexity of instruction formats is greater than that found on most RISC systems (such as SPARC). However, the fixed length makes instruction decoding faster and simpler than on CISC systems like VAX and x86.

Addressing Modes

As in most architectures, an operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register direct* mode). The only instructions that address memory are load and store operations, and branch instructions.

Load and store operations use one of the following three addressing modes:

Mode	Target address calculation
Register indirect	TA = (register)
Register indirect with index	TA = (register-1) + (register-2)
Register indirect with immediate index	TA = (register) + displacement (16 bits, signed)

The register numbers and displacement are encoded as part of the instruction.

Branch instructions use one of the following three addressing modes:

Mode	Target address calculation
Absolute	TA = actual address
Relative	TA = current instruction address + displacement {25 bits, signed}
Link Register	TA = (LR)
Count Register	TA = (CR)

The absolute address or displacement is encoded as part of the instruction.

Instruction Set

The PowerPC architecture has approximately 200 machine instructions. Some instructions are more complex than those found in most RISC systems. For example, load and store instructions may automatically update the index register to contain the just-computed target address. There are floating-point "multiply and add" instructions that take three input operands and perform a multiplication and an addition in one instruction. Such instructions reflect the PowerPC approach of using more powerful instructions, so fewer instructions are required to perform a task. This is in contrast to the more usual RISC approach, which keeps instructions simple so they can be executed as fast as possible.

In spite of this difference in philosophy, PowerPC is generally considered to be a true RISC architecture. Further discussions of these issues can be found in Smith and Weiss (1994).

Instruction execution on a PowerPC system is pipelined, as we discussed for SPARC. However, the pipelining is more sophisticated than on the original SPARC systems, with branch prediction used to speed execution. As a result, the delayed branch technique we described for SPARC is not used on PowerPC (and most other modern architectures). Further discussion of pipelining and branch prediction can be found in Tabak (1995).

Input and Output

The PowerPC architecture provides two different methods for performing I/O operations. In one approach, segments in the virtual address space are mapped onto an external address space (typically an I/O bus). Segments that are mapped in this way are called *direct-store* segments. This method is similar to the approach used in the SPARC architecture.

A reference to an address that is not in a direct-store segment represents a normal virtual memory access. In this situation, I/O is performed using the regular virtual memory management hardware and software.

1.5.3 Cray T3E Architecture

The T3E series of supercomputers was announced by Cray Research, Inc., near the end of 1995. The T3E is a massively parallel processing (MPP) system, designed for use on technical applications in scientific computing. The earlier Cray T3D system had a similar (but not identical) architecture.

A T3E system contains a large number of processing elements (PE), arranged in a three-dimensional network as illustrated in Fig. 1.8. This network provides a path for transferring data between processors. It also implements control functions that are used to synchronize the operation of the PEs used by a program. The interconnect network is circular in each dimension. Thus PEs at "opposite" ends of the three-dimensional array are adjacent with respect to the network. This is illustrated by the dashed lines in Fig. 1.8; for simplicity, most of these "circular" connections have been omitted from the drawing.

Each PE consists of a DEC Alpha EV5 RISC microprocessor (currently model 21164), local memory, and performance-accelerating control logic developed by Cray. A T3E system may contain from 16 to 2048 processing elements.

This section contains an overview of the architecture of the T3E and the DEC Alpha microprocessor. Sections 3.5.3 and 5.5.3 discuss some of the ways programs can take advantage of the multiprocessor architecture of this machine. Further information about the T3E can be found in Cray Research (1995c). Further information about the DEC Alpha architecture can be found in Sites (1992) and Tabak (1995).

Memory

Each processing element in the T3E has its own local memory with a capacity of from 64 megabytes to 2 gigabytes. The local memory within each PE is part

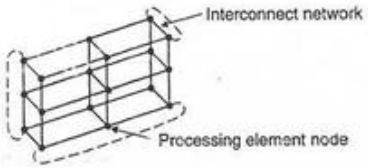


Figure 1.8 Overall T3E architecture.

of a physically distributed, logically shared memory system. System memory is physically distributed because each PE contains local memory. System memory is logically shared because the microprocessor in one PE can access the memory of another PE without involving the microprocessor in that PE.

The memory within each processing element consists of 8-bit bytes; all addresses used are byte addresses. Two consecutive bytes form a *word;* four bytes form a *longword;* eight bytes form a *quadword.* Many Alpha instructions may execute more efficiently if operands are aligned at a starting address that is a multiple of their length. The Alpha architecture supports 64-bit virtual addresses.

Registers

The Alpha architecture includes 32 general-purpose registers, designated R0 through R31; R31 always contains the value zero. Each general-purpose register is 64 bits long. These general-purpose registers can be used to store and manipulate integer data and addresses.

There are also 32 floating-point registers, designated F0 through F31; F31 always contains the value zero. Each floating-point register is 64 bits long.

In addition to the general-purpose and floating-point registers, there is a 64-bit program counter PC and several other status and control registers.

Data Formats

The Alpha architecture provides for the storage of integers, floating-point values, and characters. Integers are stored as longwords or quadwords; 2's complement is used for negative values. When interpreted as an integer, the bits of a longword or quadword have steadily increasing significance beginning with bit 0 (which is stored in the lowest-addressed byte).

There are two different types of floating-point data formats in the Alpha architecture. One group of three formats is included for compatibility with the VAX architecture. The other group consists of four IEEE standard formats, which are compatible with those used on most modern systems.

Characters may be stored one per byte, using their 8-bit ASCII codes. However, there are no byte load or store operations in the Alpha architecture; only longwords and quadwords can be transferred between a register and memory. As a consequence, characters that are to be manipulated separately are usually stored one per longword.

Instruction Formats

There are five basic instruction formats in the Alpha architecture, some of which have subforms. All of these formats are 32 bits long. (As we have noted before, this fixed length is typical of RISC systems.) The first 6 bits of the instruction word always specify the opcode; some instruction formats also have an additional "function" field.

Addressing Modes

As in most architectures, an operand value may be specified as part of the instruction itself (*immediate* mode), or it may be in a register (*register direct* mode). As in most RISC systems, the only instructions that address memory are load and store operations, and branch instructions.

Operands in memory are addressed using one of the following two modes:

Mode	Target address calculation
PC-relative	TA = (PC) + displacement (23 bits, signed)
Register indirect with displacement	TA = (register) + displacement (16 bits, signed)

Register indirect with displacement mode is used for load and store operations and for subroutine jumps. PC-relative mode is used for conditional and unconditional branches.

Instruction Set

The Alpha architecture has approximately 130 machine instructions, reflecting its RISC orientation. The instruction set is designed so that an implementation of the architecture can be as fast as possible. For example, there are no byte or word load and store instructions. This means that the memory access interface does not need to include shift-and-mask operations. Further discussion of this approach can be found in Smith and Weiss (1994).

Input and Output

The T3E system performs I/O through multiple ports into one or more I/O channels, which can be configured in a number of ways. These channels are

integrated into the network that interconnects the processing nodes. A system may be configured with up to one I/O channel for every eight PEs. All channels are accessible and controllable from all PEs.

Further information about this "scalable" I/O architecture can be found in Cray Research (1995c).

EXERCISES

Section 1.3

1. Write a sequence of instructions for SIC to set ALPHA equal to the product of BETA and GAMMA. Assume that ALPHA, BETA, and GAMMA are defined as in Fig. 1.3(a).
2. Write a sequence of instructions for SIC/XE to set ALPHA equal to 4 * BETA – 9. Assume that ALPHA and BETA are defined as in Fig. 1.3(b). Use immediate addressing for the constants.
3. Write a sequence of instructions for SIC to set ALPHA equal to the integer portion of BETA ÷ GAMMA. Assume that ALPHA and BETA are defined as in Fig. 1.3(a).
4. Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the integer portion of the quotient and DELTA to the remainder. Use register-to-register instructions to make the calculation as efficient as possible.
5. Write a sequence of instructions for SIC/XE to divide BETA by GAMMA, setting ALPHA to the value of the quotient, rounded to the nearest integer. Use register-to-register instructions to make the calculation as efficient as possible.
6. Write a sequence of instructions for SIC to clear a 20-byte string to all blanks.
7. Write a sequence of instructions for SIC/XE to clear a 20-byte string to all blanks. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.
8. Suppose that ALPHA is an array of 100 words, as defined in Fig. 1.5(a). Write a sequence of instructions for SIC to set all 100 elements of the array to 0.
9. Suppose that ALPHA is an array of 100 words, as defined in Fig. 1.5(b). Write a sequence of instructions for SIC/XE to set all 100 elements of the array to 0. Use immediate addressing and register-to-register instructions to make the process as efficient as possible.

10. Suppose that RECORD contains a 100-byte record, as in Fig. 1.7(a). Write a subroutine for SIC that will write this record onto device 05.
11. Suppose that RECORD contains a 100-byte record, as in Fig. 1.7(b). Write a subroutine for SIC/XE that will write this record onto device 05. Use immediate addressing and register-to-register instructions to make the subroutine as efficient as possible.
12. Write a subroutine for SIC that will read a record into a buffer, as in Fig. 1.7(a). The record may be any length from 1 to 100 bytes. The end of the record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH.
13. Write a subroutine for SIC/XE that will read a record into a buffer, as in Fig. 1.7(b). The record may be any length from 1 to 100 bytes. The end of the record is marked with a "null" character (ASCII code 00). The subroutine should place the length of the record read into a variable named LENGTH. Use immediate addressing and register-to-register instructions to make the subroutine as efficient as possible.

Chapter 2

Assemblers

In this chapter we discuss the design and implementation of assemblers. There are certain fundamental functions that any assembler must perform, such as translating mnemonic operation codes to their machine language equivalents and assigning machine addresses to symbolic labels used by the programmer. If we consider only these fundamental functions, most assemblers are very much alike.

Beyond this most basic level, however, the features and design of an assembler depend heavily upon the source language it translates and the machine language it produces. One aspect of this dependence is, of course, the existence of different machine instruction formats and codes to accomplish (for example) an ADD operation. As we shall see, there are also many subtler ways that assemblers depend upon machine architecture. On the other hand, there are some features of an assembler language (and the corresponding assembler) that have no direct relation to machine architecture—they are, in a sense, arbitrary decisions made by the designers of the language.

We begin by considering the design of a basic assembler for the standard version of our Simplified Instructional Computer (SIC). Section 2.1 introduces the most fundamental operations performed by a typical assembler, and describes common ways of accomplishing these functions. The algorithms and data structures that we describe are shared by almost all assemblers. Thus this level of presentation gives us a starting point from which to approach the study of more advanced assembler features. We can also use this basic structure as a framework from which to begin the design of an assembler for a completely new or unfamiliar machine.

In Section 2.2, we examine some typical extensions to the basic assembler structure that might be dictated by hardware considerations. We do this by discussing an assembler for the SIC/XE machine. Although this SIC/XE assembler certainly does not include all possible hardware-dependent features, it does contain some of the ones most commonly found in real machines. The principles and techniques should be easily applicable to other computers.

Section 2.3 presents a discussion of some of the most commonly encountered machine-independent assembler language features and their implementation. Once again, our purpose is not to cover all possible options, but rather

to introduce concepts and techniques that can be used in new and unfamiliar situations.

Section 2.4 examines some important alternative design schemes for an assembler. These are features of an assembler that are not reflected in the assembler language. For example, some assemblers process a source program in one pass instead of two; other assemblers may make more than two passes. We are concerned with the implementation of such assemblers, and also with the environments in which each might be useful.

Finally, in Section 2.5 we briefly consider some examples of actual assemblers for real machines. We do not attempt to discuss all aspects of these assemblers in detail. Instead, we focus on the most interesting features that are introduced by hardware or software design decisions.

2.1 BASIC ASSEMBLER FUNCTIONS

Figure 2.1 shows an assembler language program for the basic version of SIC. We use variations of this program throughout this chapter to show different assembler features. The line numbers are for reference only and are not part of the program. These numbers also help to relate corresponding parts of different versions of the program. The mnemonic instructions used are those introduced in Section 1.3.1 and Appendix A. Indexed addressing is indicated by adding the modifier ",X" following the operand (see line 160). Lines beginning with "." contain comments only.

In addition to the mnemonic machine instructions, we have used the following *assembler directives:*

START	Specify name and starting address for the program.
END	Indicate the end of the source program and (optionally) specify the first executable instruction in the program.
BYTE	Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
WORD	Generate one-word integer constant.
RESB	Reserve the indicated number of bytes for a data area.
RESW	Reserve the indicated number of words for a data area.

The program contains a main routine that reads records from an input device (identified with device code F1) and copies them to an output device (code 05). This main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from the buffer to the out-

Line Source statement

```
 5   COPY     START   1000        COPY FILE FROM INPUT TO OUTPUT
10   FIRST    STL     RETADR      SAVE RETURN ADDRESS
15   CLOOP    JSUB    RDREC       READ INPUT RECORD
20            LDA     LENGTH      TEST FOR EOF (LENGTH = 0)
25            COMP    ZERO
30            JEQ     ENDFIL      EXIT IF EOF FOUND
35            JSUB    WRREC       WRITE OUTPUT RECORD
40            J       CLOOP       LOOP
45   ENDFIL   LDA     EOF         INSERT END OF FILE MARKER
50            STA     BUFFER
55            LDA     THREE       SET LENGTH = 3
60            STA     LENGTH
65            JSUB    WRREC       WRITE EOF
70            LDL     RETADR      GET RETURN ADDRESS
75            RSUB                RETURN TO CALLER
80   EOF      BYTE    C'EOF'
85   THREE    WORD    3
90   ZERO     WORD    0
95   RETADR   RESW    1
100  LENGTH   RESW    1           LENGTH OF RECORD
105  BUFFER   RESB    4096        4096-BYTE BUFFER AREA
110  .
115  .        SUBROUTINE TO READ RECORD INTO BUFFER
120  .
125  RDREC    LDX     ZERO        CLEAR LOOP COUNTER
130           LDA     ZERO        CLEAR A TO ZERO
135  RLOOP    TD      INPUT       TEST INPUT DEVICE
140           JEQ     RLOOP       LOOP UNTIL READY
145           RD      INPUT       READ CHARACTER INTO REGISTER A
150           COMP    ZERO        TEST FOR END OF RECORD (X'00')
155           JEQ     EXIT        EXIT LOOP IF EOR
160           STCH    BUFFER,X    STORE CHARACTER IN BUFFER
165           TIX     MAXLEN      LOOP UNLESS MAX LENGTH
170           JLT     RLOOP         HAS BEEN REACHED
175  EXIT     STX     LENGTH      SAVE RECORD LENGTH
180           RSUB                RETURN TO CALLER
185  INPUT    BYTE    X'F1'       CODE FOR INPUT DEVICE
190  MAXLEN   WORD    4096
195  .
200  .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205  .
210  WRREC    LDX     ZERO        CLEAR LOOP COUNTER
215  WLOOP    TD      OUTPUT      TEST OUTPUT DEVICE
220           JEQ     WLOOP       LOOP UNTIL READY
225           LDCH    BUFFER,X    GET CHARACTER FROM BUFFER
230           WD      OUTPUT      WRITE CHARACTER
235           TIX     LENGTH      LOOP UNTIL ALL CHARACTERS
240           JLT     WLOOP         HAVE BEEN WRITTEN
245           RSUB                RETURN TO CALLER
250  OUTPUT   BYTE    X'05'       CODE FOR OUTPUT DEVICE
255           END     FIRST
```

Figure 2.1 Example of a SIC assembler language program.

put device. Each subroutine must transfer the record one character at a time because the only I/O instructions available are RD and WD. The buffer is necessary because the I/O rates for the two devices, such as a disk and a slow printing terminal, may be very different. (In Chapter 6, we see how to use channel programs and operating system calls on a SIC/XE system to accomplish the same functions.) The end of each record is marked with a null character (hexadecimal 00). If a record is longer than the length of the buffer (4096 bytes), only the first 4096 bytes are copied. (For simplicity, the program does not deal with error recovery when a record containing 4096 bytes or more is read.) The end of the file to be copied is indicated by a zero-length record. When the end of file is detected, the program writes EOF on the output device and terminates by executing an RSUB instruction. We assume that this program was called by the operating system using a JSUB instruction; thus, the RSUB will return control to the operating system.

2.1.1 A Simple SIC Assembler

Figure 2.2 shows the same program as in Fig. 2.1, with the generated object code for each statement. The column headed Loc gives the machine address (in hexadecimal) for each part of the assembled program. We have assumed that the program starts at address 1000. (In an actual assembler listing, of course, the comments would be retained; they have been eliminated here to save space.)

The translation of source program to object code requires us to accomplish the following functions (not necessarily in the order given):

1. Convert mnemonic operation codes to their machine language equivalents—e.g., translate STL to 14 (line 10).
2. Convert symbolic operands to their equivalent machine addresses—e.g., translate RETADR to 1033 (line 10).
3. Build the machine instructions in the proper format.
4. Convert the data constants specified in the source program into their internal machine representations—e.g., translate EOF to 454F46 (line 80).
5. Write the object program and the assembly listing.

All of these functions except number 2 can easily be accomplished by sequential processing of the source program, one line at a time. The translation of addresses, however, presents a problem. Consider the statement

```
10    1000    FIRST    STL    RETADR    141033
```


Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	2039	RDREC	LDX	ZERO	041030
130	203C		LDA	ZERO	001030
135	203F	RLOOP	TD	INPUT	E0205D
140	2042		JEQ	RLOOP	30203F
145	2045		RD	INPUT	D8205D
150	2048		COMP	ZERO	281030
155	204B		JEQ	EXIT	302057
160	204E		STCH	BUFFER,X	549039
165	2051		TIX	MAXLEN	2C205E
170	2054		JLT	RLOOP	38203F
175	2057	EXIT	STX	LENGTH	101036
180	205A		RSUB		4C0000
185	205D	INPUT	BYTE	X'F1'	F1
190	205E	MAXLEN	WORD	4096	001000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	2061	WRREC	LDX	ZERO	041030
215	2064	WLOOP	TD	OUTPUT	E02079
220	2067		JEQ	WLOOP	302064
225	206A		LDCH	BUFFER,X	509039
230	206D		WD	OUTPUT	DC2079
235	2070		TIX	LENGTH	2C1036
240	2073		JLT	WLOOP	382064
245	2076		RSUB		4C0000
250	2079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 2.2 Program from Fig. 2.1 with object code.

This instruction contains a *forward reference*—that is, a reference to a label (RETADR) that is defined later in the program. If we attempt to translate the program line by line, we will be unable to process this statement because we do not know the address that will be assigned to RETADR. Because of this, most assemblers make two passes over the source program. The first pass does little more than scan the source program for label definitions and assign addresses (such as those in the Loc column in Fig. 2.2). The second pass performs most of the actual translation previously described.

In addition to translating the instructions of the source program, the assembler must process statements called *assembler directives* (or *pseudo-instructions*). These statements are not translated into machine instructions (although they may have an effect on the object program). Instead, they provide instructions to the assembler itself. Examples of assembler directives are statements like BYTE and WORD, which direct the assembler to generate constants as part of the object program, and RESB and RESW, which instruct the assembler to reserve memory locations without generating data values. The other assembler directives in our sample program are START, which specifies the starting memory address for the object program, and END, which marks the end of the program.

Finally, the assembler must write the generated object code onto some output device. This *object program* will later be loaded into memory for execution. The simple object program format we use contains three types of records: Header, Text, and End. The Header record contains the program name, starting address, and length. Text records contain the translated (i.e., machine code) instructions and data of the program, together with an indication of the addresses where these are to be loaded. The End record marks the end of the object program and specifies the address in the program where execution is to begin. (This is taken from the operand of the program's END statement. If no operand is specified, the address of the first executable instruction is used.)

The formats we use for these records are as follows. The details of the formats (column numbers, etc.) are arbitrary; however, the information contained in these records must be present (in some form) in the object program.

Header record:

Col. 1	H
Col. 2–7	Program name
Col. 8–13	Starting address of object program (hexadecimal)
Col. 14–19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T
Col. 2–7	Starting address for object code in this record(hexadecimal)
Col. 8–9	Length of object code in this record in bytes (hexadecimal)
Col. 10–69	Object code, represented in hexadecimal (2 columns per byte of object code)

End record:

Col. 1	E
Col. 2–7	Address of first executable instruction in object program (hexadecimal)

To avoid confusion, we have used the term *column* rather than *byte* to refer to positions within object program records. This is not meant to imply the use of any particular medium for the object program.

Figure 2.3 shows the object program corresponding to Fig. 2.2, using this format. In this figure, and in the other object programs we display, the symbol ^ is used to separate fields visually. Of course, such symbols are not present in the actual object program. Note that there is no object code corresponding to addresses 1033–2038. This storage is simply reserved by the loader for use by the program during execution. (Chapter 3 contains a detailed discussion of the operation of the loader.)

We can now give a general description of the functions of the two passes of our simple assembler.

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

Figure 2.3 Object program corresponding to Fig. 2.2.

Pass 1 (define symbols):

1. Assign addresses to all statements in the program.
2. Save the values (addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives. (This includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW, etc.)

Pass 2 (assemble instructions and generate object program):

1. Assemble instructions (translating operation codes and looking up addresses).
2. Generate data values defined by BYTE, WORD, etc.
3. Perform processing of assembler directives not done during Pass 1.
4. Write the object program and the assembly listing.

In the next section we discuss these functions in more detail, describe the internal tables required by the assembler, and give an overall description of the logic flow of each pass.

2.1.2 Assembler Algorithm and Data Structures

Our simple assembler uses two major internal data structures: the Operation Code Table (OPTAB) and the Symbol Table (SYMTAB). OPTAB is used to look up mnemonic operation codes and translate them to their machine language equivalents. SYMTAB is used to store values (addresses) assigned to labels.

We also need a Location Counter LOCCTR. This is a variable that is used to help in the assignment of addresses. LOCCTR is initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR. Thus whenever we reach a label in the source program, the current value of LOCCTR gives the address to be associated with that label.

The Operation Code Table must contain (at least) the mnemonic operation code and its machine language equivalent. In more complex assemblers, this table also contains information about instruction format and length. During Pass 1, OPTAB is used to look up and validate operation codes in the source program. In Pass 2, it is used to translate the operation codes to machine language. Actually, in our simple SIC assembler, both of these processes could be done together in either Pass 1 or Pass 2. However, for a machine (such as SIC/XE) that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

Likewise, we must have the information from OPTAB in Pass 2 to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction. We have chosen to retain this structure in the current discussion because it is typical of most real assemblers.

OPTAB is usually organized as a hash table, with mnemonic operation code as the key. (The information in OPTAB is, of course, predefined when the assembler itself is written, rather than being loaded into the table at execution time.) The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. In most cases, OPTAB is a static table—that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored. Most of the time, however, a general-purpose hashing method is used. Further information about the design and construction of hash tables may be found in any good data structures text, such as Lewis and Denenberg (1991) or Knuth (1973).

The symbol table (SYMTAB) includes the name and value (address) for each label in the source program, together with flags to indicate error conditions (e.g., a symbol defined in two different places). This table may also contain other information about the data area or instruction labeled—for example, its type or length. During Pass 1 of the assembler, labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses (from LOCCTR). During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely (if ever) deleted from this table, efficiency of deletion is not an important consideration. Because SYMTAB is used heavily throughout the assembly, care should be taken in the selection of a hashing function. Programmers often select many labels that have similar characteristics—for example, labels that start or end with the same characters (like LOOP1, LOOP2, LOOPA) or are of the same length (like A, X, Y, Z). It is important that the hashing function used perform well with such non-random keys. Division of the entire key by a prime table length often gives good results.

It is possible for both passes of the assembler to read the original source program as input. However, there is certain information (such as location counter values and error flags for statements) that can or should be communicated between the two passes. For this reason, Pass 1 usually writes an *intermediate file* that contains each source statement together with its assigned address, error indicators, etc. This file is used as the input to Pass 2. This working copy of the source program can also be used to retain the results of certain

operations that may be performed during Pass 1 (such as scanning the operand field for symbols and addressing flags), so these need not be performed again during Pass 2. Similarly, pointers into OPTAB and SYMTAB may be retained for each operation code and symbol used. This avoids the need to repeat many of the table-searching operations.

Figures 2.4(a) and (b) show the logic flow of the two passes of our assembler. Although described for the simple assembler we are discussing, this is also the underlying logic for more complex two-pass assemblers that we consider later. We assume for simplicity that the source lines are written in a fixed format with fields LABEL, OPCODE, and OPERAND. If one of these fields contains a character string that represents a number, we denote its numeric value with the prefix # (for example, #[OPERAND]).

At this stage, it is very important for you to understand thoroughly the algorithms in Fig. 2.4. You are strongly urged to follow through the logic in these algorithms, applying them by hand to the program in Fig. 2.1 to produce the object program of Fig. 2.3.

Much of the detail of the assembler logic has, of course, been left out to emphasize the overall structure and main concepts. You should think about these details for yourself, and you should also attempt to identify those functions of the assembler that should be implemented as separate procedures or modules. (For example, the operations "search symbol table" and "read input line" might be good candidates for such implementation.) This kind of thoughtful analysis should be done before you make any attempt to actually implement an assembler or any other large piece of software.

Chapter 8 contains an introduction to software engineering tools and techniques, and illustrates the use of such techniques in designing and implementing a simple assembler. You may want to read this material now to gain further insight into how an assembler might be constructed.

2.2 MACHINE-DEPENDENT ASSEMBLER FEATURES

In this section, we consider the design and implementation of an assembler for the more complex XE version of SIC. In doing so, we examine the effect of the extended hardware on the structure and functions of the assembler. Many real machines have certain architectural features that are similar to those we consider here. Thus our discussion applies in large part to these machines as well as to SIC/XE.

Figure 2.5 shows the example program from Fig. 2.1 as it might be rewritten to take advantage of the SIC/XE instruction set. In our assembler language, indirect addressing is indicated by adding the prefix @ to the operand

Pass 1:

```
begin
  read first input line
  if OPCODE = 'START' then
     begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line
     end {if START}
  else
     initialize LOCCTR to 0
  while OPCODE ≠ 'END' do
     begin
        if this is not a comment line then
           begin
              if there is a symbol in the LABEL field then
                 begin
                    search SYMTAB for LABEL
                    if found then
                       set error flag (duplicate symbol)
                    else
                       insert (LABEL,LOCCTR) into SYMTAB
                 end {if symbol}
              search OPTAB for OPCODE
              if found then
                 add 3 {instruction length} to LOCCTR
              else if OPCODE = 'WORD' then
                 add 3 to LOCCTR
              else if OPCODE = 'RESW' then
                 add 3 * #[OPERAND] to LOCCTR
              else if OPCODE = 'RESB' then
                 add #[OPERAND] to LOCCTR
              else if OPCODE  = 'BYTE' then
                 begin
                    find length of constant in bytes
                    add length to LOCCTR
                 end {if BYTE}
              else
                 set error flag (invalid operation code)
           end {if not a comment}
        write line to intermediate file
        read next input line
     end {while not END}
  write last line to intermediate file
  save (LOCCTR - starting address) as program length
end {Pass 1}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

جامعة الإسكندرية
مكتبة

Pass 2:

```
begin
  read first input line (from intermediate file)
  if OPCODE = 'START' then
     begin
        write listing line
        read next input line
     end (if START)
  write Header record to object program
  initialize first Text record
  while OPCODE ≠ 'END' do
     begin
        if this is not a comment line then
           begin
              search OPTAB for OPCODE
              if found then
                 begin
                    if there is a symbol in OPERAND field then
                       begin
                          search SYMTAB for OPERAND
                          if found then
                             store symbol value as operand address
                          else
                             begin
                                store 0 as operand address
                                set error flag (undefined symbol)
                             end
                       end (if symbol)
                    else
                       store 0 as operand address
                    assemble the object code instruction
                 end (if opcode found)
              else if OPCODE = 'BYTE' or 'WORD' then
                 convert constant to object code
              if object code will not fit into the current Text record then
                 begin
                    write Text record to object program
                    initialize new Text record
                 end
              add object code to Text record
           end (if not comment)
        write listing line
        read next input line
     end (while not END)
  write last Text record to object program
  write End record to object program
  write last listing line
end (Pass 2)
```

Figure 2.4(b) Algorithm for Pass 2 of assembler.

Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
12		LDB	#LENGTH	ESTABLISH BASE REGISTER
13		BASE	LENGTH	
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		
120	.			
125	RDREC	CLEAR	X	CLEAR LOOP COUNTER
130		CLEAR	A	CLEAR A TO ZERO
132		CLEAR	S	CLEAR S TO ZERO
133		+LDT	#4096	
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		JEQ	RLOOP	LOOP UNTIL READY
145		RD	INPUT	READ CHARACTER INTO REGISTER A
150		COMPR	A,S	TEST FOR END OF RECORD (X'00')
155		JEQ	EXIT	EXIT LOOP IF EOR
160		STCH	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIXR	T	LOOP UNLESS MAX LENGTH
170		JLT	RLOOP	HAS BEEN REACHED
175	EXIT	STX	LENGTH	SAVE RECORD LENGTH
180		RSUB		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
195	.			
200	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.			
210	WRREC	CLEAR	X	CLEAR LOOP COUNTER
212		LDT	LENGTH	
215	WLOOP	TD	OUTPUT	TEST OUTPUT DEVICE
220		JEQ	WLOOP	LOOP UNTIL READY
225		LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
230		WD	OUTPUT	WRITE CHARACTER
235		TIXR	T	LOOP UNTIL ALL CHARACTERS
240		JLT	WLOOP	HAVE BEEN WRITTEN
245		RSUB		RETURN TO CALLER
250	OUTPUT	BYTE	X'05'	CODE FOR OUTPUT DEVICE
255		END	FIRST	

Figure 2.5 Example of a SIC/XE program.

(see line 70). Immediate operands are denoted with the prefix # (lines 25, 55, 133). Instructions that refer to memory are normally assembled using either the program-counter relative or the base relative mode. The assembler directive BASE (line 13) is used in conjunction with base relative addressing. (See Section 2.2.1 for a discussion and examples.) If the displacements required for both program-counter relative and base relative addressing are too large to fit into a 3-byte instruction, then the 4-byte extended format (Format 4) must be used. The extended instruction format is specified with the prefix + added to the operation code in the source statement (see lines 15, 35, 65). It is the programmer's responsibility to specify this form of addressing when it is required.

The main differences between this version of the program and the version in Fig. 2.1 involve the use of register-to-register instructions (in place of register-to-memory instructions) wherever possible. For example, the statement on line 150 is changed from COMP ZERO to COMPR A,S. Similarly, line 165 is changed from TIX MAXLEN to TIXR T. In addition, immediate and indirect addressing have been used as much as possible (for example, lines 25, 55, and 70).

These changes take advantage of the more advanced SIC/XE architecture to improve the execution speed of the program. Register-to-register instructions are faster than the corresponding register-to-memory operations because they are shorter, and, more importantly, because they do not require another memory reference. (Fetching an operand from a register is much faster than retrieving it from main memory.) Likewise, when using immediate addressing, the operand is already present as part of the instruction and need not be fetched from anywhere. The use of indirect addressing often avoids the need for another instruction (as in the "return" operation on line 70). You may notice that some of the changes require the addition of other instructions to the program. For example, changing COMP to COMPR on line 150 forces us to add the CLEAR instruction on line 132. This still results in an improvement in execution speed. The CLEAR is executed only once for each record read, whereas the benefits of COMPR (as opposed to COMP) are realized for every byte of data transferred.

In Section 2.2.1, we examine the assembly of this SIC/XE program, focusing on the differences in the assembler that are required by the new addressing modes. (You may want to briefly review the instruction formats and target address calculations described in Section 1.3.2.) These changes are direct consequences of the extended hardware functions.

Section 2.2.2 discusses an indirect consequence of the change to SIC/XE. The larger main memory of SIC/XE means that we may have room to load and run several programs at the same time. This kind of sharing of the machine between programs is called *multiprogramming*. Such sharing often results in more productive use of the hardware. (We discuss this concept, and its implications for operating systems, in Chapter 6.) To take full advantage of this capability, however, we must be able to load programs into memory wherever there is room, rather than specifying a fixed address at assembly time. Section 2.2.2 introduces the idea of program *relocation* and discusses its implications for the assembler.

2.2.1 Instruction Formats and Addressing Modes

Figure 2.6 shows the object code generated for each statement in the program of Fig. 2.5. In this section we consider the translation of the source statements, paying particular attention to the handling of different instruction formats and different addressing modes. Note that the START statement now specifies a beginning program address of 0. As we discuss in the next section, this indicates a relocatable program. For the purposes of instruction assembly, however, the program will be translated exactly as if it were really to be loaded at machine address 0.

Translation of register-to-register instructions such as CLEAR (line 125) and COMPR (line 150) presents no new problems. The assembler must simply convert the mnemonic operation code to machine language (using OPTAB) and change each register mnemonic to its numeric equivalent. This translation is done during Pass 2, at the same point at which the other types of instructions are assembled. The conversion of register mnemonics to numbers can be done with a separate table; however, it is often convenient to use the symbol table for this purpose. To do this, SYMTAB would be preloaded with the register names (A, X, etc.) and their values (0, 1, etc.).

Most of the register-to-memory instructions are assembled using either program-counter relative or base relative addressing. The assembler must, in either case, calculate a displacement to be assembled as part of the object instruction. This is computed so that the correct target address results when the displacement is added to the contents of the program counter (PC) or the base register (B). Of course, the resulting displacement must be small enough to fit in the 12-bit field in the instruction. This means that the displacement must be between 0 and 4095 (for base relative mode) or between –2048 and +2047 (for program-counter relative mode).

If neither program-counter relative nor base relative addressing can be used (because the displacements are too large), then the 4-byte extended instruction format (Format 4) must be used. This 4-byte format contains a 20-bit address field, which is large enough to contain the full memory address. In this case, there is no displacement to be calculated. For example, in the instruction

```
15    0006    CLOOP    +JSUB    RDREC          4B101036
```

```
Line   Loc      Source statement                    Object code

  5    0000     COPY     START   0
 10    0000     FIRST    STL     RETADR             17202D
 12    0003              LDB     #LENGTH            69202D
 13                      BASE    LENGTH
 15    0006     CLOOP   +JSUB    RDREC              4B101036
 20    000A              LDA     LENGTH             032026
 25    000D              COMP    #0                 290000
 30    0010              JEQ     ENDFIL             332007
 35    0013             +JSUB    WRREC              4B10105D
 40    0017              J       CLOOP              3F2FEC
 45    001A     ENDFIL   LDA     EOF                032010
 50    001D              STA     BUFFER             0F2016
 55    0020              LDA     #3                 010003
 60    0023              STA     LENGTH             0F200D
 65    0026             +JSUB    WRREC              4B10105D
 70    002A              J       @RETADR            3E2003
 80    002D     EOF      BYTE    C'EOF'             454F46
 95    0030     RETADR   RESW    1
100    0033     LENGTH   RESW    1
105    0036     BUFFER   RESB    4096
110             .
115             .        SUBROUTINE TO READ RECORD INTO BUFFER
120             .
125    1036     RDREC    CLEAR   X                  B410
130    1038              CLEAR   A                  B400
132    103A              CLEAR   S                  B440
133    103C             +LDT     #4096              75101000
135    1040     RLOOP    TD      INPUT              E32019
140    1043              JEQ     RLOOP              332FFA
145    1046              RD      INPUT              DB2013
150    1049              COMPR   A,S                A004
155    104B              JEQ     EXIT               332008
160    104E              STCH    BUFFER,X           57C003
165    1051              TIXR    T                  B850
170    1053              JLT     RLOOP              3B2FEA
175    1056     EXIT     STX     LENGTH             134000
180    1059              RSUB                       4F0000
185    105C     INPUT    BYTE    X'F1'              F1
195             .
200             .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205             .
210    105D     WRREC    CLEAR   X                  B410
212    105F              LDT     LENGTH             774000
215    1062     WLOOP    TD      OUTPUT             E32011
220    1065              JEQ     WLOOP              332FFA
225    1068              LDCH    BUFFER,X           53C003
230    106B              WD      OUTPUT             DF2008
235    106E              TIXR    T                  B850
240    1070              JLT     WLOOP              3B2FEF
245    1073              RSUB                       4F0000
250    1076     OUTPUT   BYTE    X'05'              05
255                      END     FIRST
```

Figure 2.6 Program from Fig. 2.5 with object code.

the operand address is 1036. This full address is stored in the instruction, with bit *e* set to 1 to indicate extended instruction format.

Note that the programmer must specify the extended format by using the prefix + (as on line 15). If extended format is not specified, our assembler first attempts to translate the instruction using program-counter relative addressing. If this is not possible (because the required displacement is out of range), the assembler then attempts to use base relative addressing. If neither form of relative addressing is applicable and extended format is not specified, then the instruction cannot be properly assembled. In this case, the assembler must generate an error message.

We now examine the details of the displacement calculation for program-counter relative and base relative addressing modes. The computation that the assembler needs to perform is essentially the target address calculation in reverse. You may want to review this from Section 1.3.2.

The instruction

```
10    0000    FIRST    STL     RETADR          17202D
```

is a typical example of program-counter relative assembly. During execution of instructions on SIC (as in most computers), the program counter is advanced *after* each instruction is fetched and *before* it is executed. Thus during the execution of the STL instruction, PC will contain the address of the *next* instruction (that is, 0003). From the Loc column of the listing, we see that RETADR (line 95) is assigned the address 0030. (The assembler would, of course, get this address from SYMTAB.) The displacement we need in the instruction is 30 – 3 = 2D. At execution time, the target address calculation performed will be (PC) + disp, resulting in the correct address (0030). Note that bit *p* is set to 1 to indicate program-counter relative addressing, making the last 2 bytes of the instruction 202D. Also note that bits *n* and *i* are both set to 1, indicating neither indirect nor immediate addressing; this makes the first byte 17 instead of 14. (See Fig. 1.1 in Section 1.3.2 for a review of the location and setting of the addressing-mode bit flags.)

Another example of program-counter relative assembly is the instruction

```
40    0017    J       CLOOP           3F2FEC
```

Here the operand address is 0006. During instruction execution, the program counter will contain the address 001A. Thus the displacement required is 6 – 1A = –14. This is represented (using 2's complement for negative numbers) in a 12-bit field as FEC, which is the displacement assembled into the object code.

The displacement calculation process for base relative addressing is much the same as for program-counter relative addressing. The main difference is

that the assembler knows what the contents of the program counter will be at execution time. The base register, on the other hand, is under control of the programmer. Therefore, the programmer must tell the assembler what the base register will contain during execution of the program so that the assembler can compute displacements. This is done in our example with the assembler directive BASE. The statement BASE LENGTH (line 13) informs the assembler that the base register will contain the *address* of LENGTH. The preceding instruction (LDB #LENGTH) loads this value into the register during program execution. The assembler assumes for addressing purposes that register B contains this address until it encounters another BASE statement. Later in the program, it may be desirable to use register B for another purpose (for example, as temporary storage for a data value). In such a case, the programmer must use another assembler directive (perhaps NOBASE) to inform the assembler that the contents of the base register can no longer be relied upon for addressing.

It is important to understand that BASE and NOBASE are assembler directives, and produce no executable code. The programmer must provide instructions that load the proper value into the base register during execution. If this is not done properly, the target address calculation will not produce the correct operand address.

The instruction

```
160     104E            STCH    BUFFER,X        57C003
```

is a typical example of base relative assembly. According to the BASE statement, register B will contain 0033 (the address of LENGTH) during execution. The address of BUFFER is 0036. Thus the displacement in the instruction must be 36 – 33 = 3. Notice that bits *x* and *b* are set to 1 in the assembled instruction to indicate indexed and base relative addressing. Another example is the instruction STX LENGTH on line 175. Here the displacement calculated is 0.

Notice the difference between the assembly of the instructions on lines 20 and 175. On line 20, LDA LENGTH is assembled with program-counter relative addressing. On line 175, STX LENGTH uses base relative addressing, as noted previously. (If you calculate the program-counter relative displacement that would be required for the statement on line 175, you will see that it is too large to fit into the 12-bit displacement field.) The statement on line 20 could also have used base relative mode. In our assembler, however, we have arbitrarily chosen to attempt program-counter relative assembly first.

The assembly of an instruction that specifies immediate addressing is simpler because no memory reference is involved. All that is necessary is to convert the immediate operand to its internal representation and insert it into the instruction. The instruction


```
55      0020            LDA     #3              010003
```

is a typical example of this, with the operand stored in the instruction as 003, and bit *i* set to 1 to indicate immediate addressing. Another example can be found in the instruction

```
133     103C            +LDT    #4096           75101000
```

In this case the operand (4096) is too large to fit into the 12-bit displacement field, so the extended instruction format is called for. (If the operand were too large even for this 20-bit address field, immediate addressing could not be used.)

A different way of using immediate addressing is shown in the instruction

```
12      0003            LDB     #LENGTH         69202D
```

In this statement the immediate operand is the *symbol* LENGTH. Since the *value* of this symbol is the *address* assigned to it, this immediate instruction has the effect of loading register B with the address of LENGTH. Note here that we have combined program-counter relative addressing with immediate addressing. Although this may appear unusual, the interpretation is consistent with our previous uses of immediate operands. In general, the target address calculation is performed; then, if immediate mode is specified, the *target address* (not the *contents* stored at that address) becomes the operand. (In the LDA statement on line 55, for example, bits *x*, *b*, and *p* are all 0. Thus the target address is simply the displacement 003.)

The assembly of instructions that specify indirect addressing presents nothing really new. The displacement is computed in the usual way to produce the target address desired. Then bit *n* is set to indicate that the contents stored at this location represent the *address* of the operand, not the operand itself. Line 70 shows a statement that combines program-counter relative and indirect addressing in this way.

2.2.2 Program Relocation

As we mentioned before, it is often desirable to have more than one program at a time sharing the memory and other resources of the machine. If we knew in advance exactly which programs were to be executed concurrently in this way, we could assign addresses when the programs were assembled so that they would fit together without overlap or wasted space. Most of the time, however, it is not practical to plan program execution this closely. (We usually do not know exactly when jobs will be submitted, exactly how long they will

run, etc.) Because of this, it is desirable to be able to load a program into memory wherever there is room for it. In such a situation the actual starting address of the program is not known until load time.

The program we considered in Section 2.1 is an example of an absolute program (or absolute assembly). This program must be loaded at address 1000 (the address that was specified at assembly time) in order to execute properly. To see this, consider the instruction

```
55      101B            LDA     THREE           00102D
```

from Fig. 2.2. In the object program (Fig. 2.3), this statement is translated as 00102D, specifying that register A is to be loaded from memory address 102D. Suppose we attempt to load and execute the program at address 2000 instead of address 1000. If we do this, address 102D will not contain the value that we expect—in fact, it will probably be part of some other user's program.

Obviously we need to make some change in the address portion of this instruction so we can load and execute our program at address 2000. On the other hand, there are parts of the program (such as the constant 3 generated from line 85) that should remain the same regardless of where the program is loaded. Looking at the object code alone, it is in general not possible to tell which values represent addresses and which represent constant data items.

Since the assembler does not know the actual location where the program will be loaded, it cannot make the necessary changes in the addresses used by the program. However, the assembler can identify for the loader those parts of the object program that need modification. An object program that contains the information necessary to perform this kind of modification is called a *relocatable* program.

To look at this in more detail, consider the program from Figs. 2.5 and 2.6. In the preceding section, we assembled this program using a starting address of 0000. Figure 2.7(a) shows this program loaded beginning at address 0000. The JSUB instruction from line 15 is loaded at address 0006. The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. (These addresses are, of course, the same as those assigned by the assembler.)

Now suppose that we want to load this program beginning at address 5000, as shown in Fig. 2.7(b). The address of the instruction labeled RDREC is then 6036. Thus the JSUB instruction must be modified as shown to contain this new address. Likewise, if we loaded the program beginning at address 7420 (Fig. 2.7c), the JSUB instruction would need to be changed to 4B108456 to correspond to the new address of RDREC.

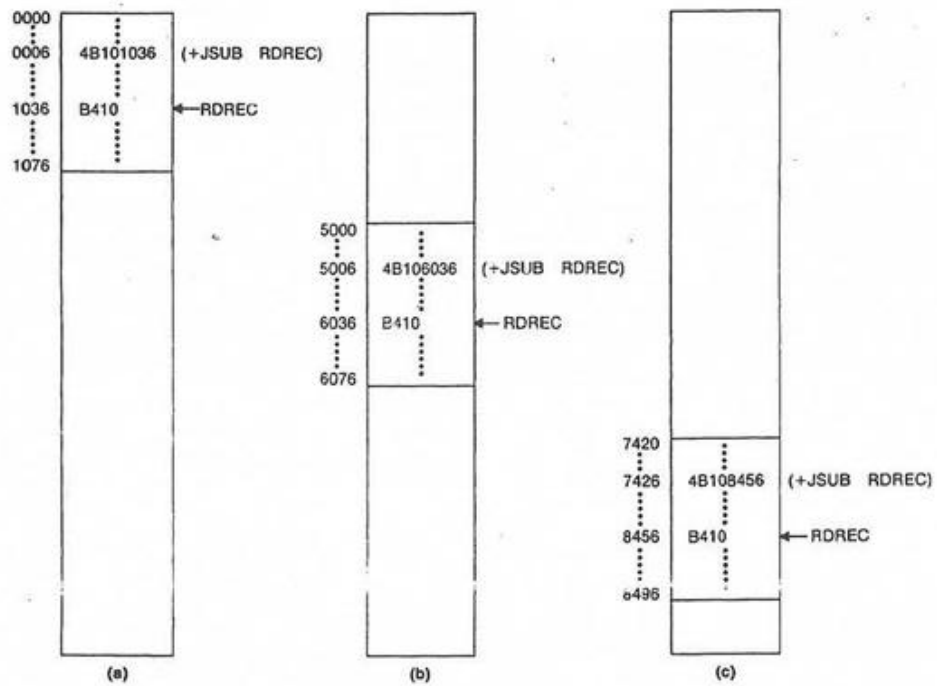


Figure 2.7 Examples of program relocation.

Note that no matter where the program is loaded, RDREC is always 1036 bytes past the starting address of the program. This means that we can solve the relocation problem in the following way:

1. When the assembler generates the object code for the JSUB instruction we are considering, it will insert the address of RDREC *relative to the start of the program.* (This is the reason we initialized the location counter to 0 for the assembly.)
2. The assembler will also produce a command for the loader, instructing it to *add* the beginning address of the program to the address field in the JSUB instruction at load time.

The command for the loader, of course, must also be a part of the object program. We can accomplish this with a Modification record having the following format:

Modification record:

Col. 1	M
Col. 2–7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
Col. 8–9	Length of the address field to be modified, in half-bytes (hexadecimal)

The length is stored in half-bytes (rather than bytes) because the address field to be modified may not occupy an integral number of bytes. (For example, the address field in the JSUB instruction we considered above occupies 20 bits, which is 5 half-bytes.) The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If this field occupies an odd number of half-bytes, it is assumed to begin in the middle of the first byte at the starting location. These conventions are, of course, closely related to the architecture of SIC/XE. For other types of machines, the half-byte approach might not be appropriate (see Exercise 2.2.9).

For the JSUB instruction we are using as an example, the Modification record would be

```
M00000705
```

This record specifies that the beginning address of the program is to be added to a field that begins at address 000007 (relative to the start of the program) and is 5 half-bytes in length. Thus in the assembled instruction 4B101036, the first 12 bits (4B1) will remain unchanged. The program load address will be added to the last 20 bits (01036) to produce the correct operand address. (You should check for yourself that this gives the results shown in Fig. 2.7.)

Exactly the same kind of relocation must be performed for the instructions on lines 35 and 65 in Fig. 2.6. The rest of the instructions in the program, however, need not be modified when the program is loaded. In some cases this is because the instruction operand is not a memory address at all (e.g., CLEAR S or LDA #3). In other cases no modification is needed because the operand is specified using program-counter relative or base relative addressing. For example, the instruction on line 10 (STL RETADR) is assembled using program-counter relative addressing with displacement 02D. No matter where the program is loaded in memory, the word labeled RETADR will always be 2D

bytes away from the STL instruction; thus no instruction modification is needed. When the STL is executed, the program counter will contain the (actual) address of the next instruction. The target address calculation process will then produce the correct (actual) operand address corresponding to RETADR.

Similarly the distance between LENGTH and BUFFER will always be 3 bytes. Thus the displacement in the base relative instruction on line 160 will be correct without modification. (The contents of the base register will, of course, depend upon where the program is loaded. However, this will be taken care of automatically when the program-counter relative instruction LDB #LENGTH is executed.)

By now it should be clear that the only parts of the program that require modification at load time are those that specify direct (as opposed to relative) addresses. For this SIC/XE program, the only such direct addresses are found in extended format (4-byte) instructions. This is an advantage of relative addressing—if we were to attempt to relocate the program from Fig. 2.1, we would find that almost every instruction required modification.

Figure 2.8 shows the complete object program corresponding to the source program of Fig. 2.5. Note that the Text records are exactly the same as those that would be produced by an absolute assembler (with program starting address of 0). However, the load addresses in the Text records are interpreted as relative, rather than absolute, locations. (The same is, of course, true of the addresses in the Modification and End records.) There is one Modification record for each address field that needs to be changed when the program is relocated (in this case, the three +JSUB instructions). You should verify these Modification records yourself and make sure you understand the contents of each. In Chapter 3 we consider in detail how the loader performs the required program modification. It is important that you understand the *concepts* involved now, however, because we build on these concepts in the next section.

```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05
M^000014^05
M^000027^05
E^000000
```

Figure 2.8 Object program corresponding to Fig. 2.6.

2.3 MACHINE-INDEPENDENT ASSEMBLER FEATURES

In this section, we discuss some common assembler features that are not closely related to machine architecture. Of course, more advanced machines tend to have more complex software; therefore the features we consider are more likely to be found on larger and more complex machines. However, the presence or absence of such capabilities is much more closely related to issues such as programmer convenience and software environment than it is to machine architecture.

In Section 2.3.1 we discuss the implementation of literals within an assembler, including the required data structures and processing logic. Section 2.3.2 discusses two assembler directives (EQU and ORG) whose main function is the definition of symbols. Section 2.3.3 briefly examines the use of expressions in assembler language statements, and discusses the different types of expressions and their evaluation and use.

In Sections 2.3.4 and 2.3.5 we introduce the important topics of program blocks and control sections. We discuss the reasons for providing such capabilities and illustrate some different uses with examples. We also introduce a set of assembler directives for supporting these features and discuss their implementation.

2.3.1 Literals

It is often convenient for the programmer to be able to write the value of a constant operand as a part of the instruction that uses it. This avoids having to define the constant elsewhere in the program and make up a label for it. Such an operand is called a *literal* because the value is stated "literally" in the instruction. The use of literals is illustrated by the program in Fig. 2.9. The object code generated for the statements of this program is shown in Fig. 2.10. (This program is a modification of the one in Fig. 2.5; other changes are discussed later in Section 2.3.)

In our assembler language notation, a literal is identified with the prefix =, which is followed by a specification of the literal value, using the same notation as in the BYTE statement. Thus the literal in the statement

```
45    001A    ENDFIL  LDA    =C'EOF'        032010
```

specifies a 3-byte operand whose value is the character string EOF. Likewise the statement

```
215   1062    WLOOP   TD     =X'05'         E32011
```


Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
13		LDB	#LENGTH	ESTABLISH BASE REGISTER
14		BASE	LENGTH	
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	=C'EOF'	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
93		LTORG		
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	
107	MAXLEN	EQU	BUFEND-BUFFER	MAXIMUM RECORD LENGTH
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		
120	.			
125	RDREC	CLEAR	X	CLEAR LOOP COUNTER
130		CLEAR	A	CLEAR A TO ZERO
132		CLEAR	S	CLEAR S TO ZERO
133		+LDT	#MAXLEN	
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		JEQ	RLOOP	LOOP UNTIL READY
145		RD	INPUT	READ CHARACTER INTO REGISTER A
150		COMPR	A,S	TEST FOR END OF RECORD (X'00')
155		JEQ	EXIT	EXIT LOOP IF EOR
160		STCH	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIXR	T	LOOP UNLESS MAX LENGTH
170		JLT	RLOOP	HAS BEEN REACHED
175	EXIT	STX	LENGTH	SAVE RECORD LENGTH
180		RSUB		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
195	.			
200	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.			
210	WRREC	CLEAR	X	CLEAR LOOP COUNTER
212		LDT	LENGTH	
215	WLOOP	TD	=X'05'	TEST OUTPUT DEVICE
220		JEQ	WLOOP	LOOP UNTIL READY
225		LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
230		WD	=X'05'	WRITE CHARACTER
235		TIXR	T	LOOP UNTIL ALL CHARACTERS
240		JLT	WLOOP	HAVE BEEN WRITTEN
245		RSUB		RETURN TO CALLER
255		END	FIRST	

Figure 2.9 Program demonstrating additional assembler features.

```
Line   Loc     Source statement                    Object code
  5    0000    COPY     START   0
 10    0000    FIRST    STL     RETADR             17202D
 13    0003             LDB     #LENGTH            69202D
 14                     BASE    LENGTH
 15    0006    CLOOP    +JSUB   RDREC              4B101036
 20    000A             LDA     LENGTH             032026
 25    000D             COMP    #0                 290000
 30    0010             JEQ     ENDFIL             332007
 35    0013             +JSUB   WRREC              4B10105D
 40    0017             J       CLOOP              3F2FEC
 45    001A    ENDFIL   LDA     =C'EOF'            032010
 50    001D             STA     BUFFER             0F2016
 55    0020             LDA     #3                 010003
 60    0023             STA     LENGTH             0F200D
 65    0026             +JSUB   WRREC              4B10105D
 70    002A             J       @RETADR            3E2003
 93                     LTORG
       002D    *        =C'EOF'                    454F46
 95    0030    RETADR   RESW    1
100    0033    LENGTH   RESW    1
105    0036    BUFFER   RESB    4096
106    1036    BUFEND   EQU     *
107    1000    MAXLEN   EQU     BUFEND-BUFFER
110            .
115            .        SUBROUTINE TO READ RECORD INTO BUFFER
120            .
125    1036    RDREC    CLEAR   X                  B410
130    1038             CLEAR   A                  B400
132    103A             CLEAR   S                  B440
133    103C             +LDT    #MAXLEN            75101000
135    1040    RLOOP    TD      INPUT              E32019
140    1043             JEQ     RLOOP              332FFA
145    1046             RD      INPUT              DB2013
150    1049             COMPR   A,S                A004
155    104B             JEQ     EXIT               332008
160    104E             STCH    BUFFER,X           57C003
165    1051             TIXR    T                  B850
170    1053             JLT     RLOOP              3B2FEA
175    1056    EXIT     STX     LENGTH             134000
180    1059             RSUB                       4F0000
185    105C    INPUT    BYTE    X'F1'              F1
195            .
200            .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205            .
210    105D    WRREC    CLEAR   X                  B410
212    105F             LDT     LENGTH             774000
215    1062    WLOOP    TD      =X'05'             E32011
220    1065             JEQ     WLOOP              332FFA
225    1068             LDCH    BUFFER,X           53C003
230    106B             WD      =X'05'             DF2008
235    106E             TIXR    T                  B850
240    1070             JLT     WLOOP              3B2FEF
245    1073             RSUB                       4F0000
255                     END     FIRST
       1076    *        =X'05'                     05
```

Figure 2.10 Program from Fig. 2.9 with object code.

specifies a 1-byte literal with the hexadecimal value 05. The notation used for literals varies from assembler to assembler; however, most assemblers use some symbol (as we have used =) to make literal identification easier.

It is important to understand the difference between a literal and an immediate operand. With immediate addressing, the operand value is assembled as part of the machine instruction. With a literal, the assembler generates the specified value as a constant at some other memory location. The *address* of this generated constant is used as the target address for the machine instruction. The effect of using a literal is exactly the same as if the programmer had defined the constant explicitly and used the label assigned to the constant as the instruction operand. (In fact, the generated object code for lines 45 and 215 in Fig. 2.10 is identical to the object code for the corresponding lines in Fig. 2.6.) You should compare the object instructions generated for lines 45 and 55 in Fig. 2.10 to make sure you understand how literals and immediate operands are handled.

All of the literal operands used in a program are gathered together into one or more *literal pools*. Normally literals are placed into a pool at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. Such a literal pool listing is shown in Fig. 2.10 immediately following the END statement. In this case, the pool consists of the single literal =X'05'.

In some cases, however, it is desirable to place literals into a pool at some other location in the object program. To allow this, we introduce the assembler directive LTORG (line 93 in Fig. 2.9). When the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the program). This literal pool is placed in the object program at the location where the LTORG directive was encountered (see Fig. 2.10). Of course, literals placed in a pool by LTORG will not be repeated in the pool at the end of the program.

If we had not used the LTORG statement on line 93, the literal =C'EOF' would be placed in the pool at the end of the program. This literal pool would begin at address 1073. This means that the literal operand would be placed too far away from the instruction referencing it to allow program-counter relative addressing. The problem, of course, is the large amount of storage reserved for BUFFER. By placing the literal pool before this buffer, we avoid having to use extended format instructions when referring to the literals. The need for an assembler directive such as LTORG usually arises when it is desirable to keep the literal operand close to the instruction that uses it.

Most assemblers recognize duplicate literals—that is, the same literal used in more than one place in the program—and store only one copy of the specified data value. For example, the literal =X'05' is used in our program on lines

215 and 230. However, only one data area with this value is generated. Both instructions refer to the same address in the literal pool for their operand.

The easiest way to recognize duplicate literals is by comparison of the character strings defining them (in this case, the string =X'05'). Sometimes a slight additional saving is possible if we look at the generated data value instead of the defining expression. For example, the literals =C'EOF' and =X'454F46' would specify identical operand values. The assembler might avoid storing both literals if it recognized this equivalence. However, the benefits realized in this way are usually not great enough to justify the additional complexity in the assembler.

If we use the character string defining a literal to recognize duplicates, we must be careful of literals whose value depends upon their location in the program. Suppose, for example, that we allow literals that refer to the current value of the location counter (often denoted by the symbol *). Such literals are sometimes useful for loading base registers. For example, the statements

```
BASE    *
LDB     =*
```

as the first lines of a program would load the beginning address of the program into register B. This value would then be available for base relative addressing.

Such a notation can, however, cause a problem with the detection of duplicate literals. If a literal =* appeared on line 13 of our example program, it would specify an operand with value 0003. If the same literal appeared on line 55, it would specify an operand with value 0020. In such a case, the literal operands have identical names; however, they have different values, and both must appear in the literal pool. The same problem arises if a literal refers to any other item whose value changes between one point in the program and another.

Now we are ready to describe how the assembler handles literal operands. The basic data structure needed is a *literal table* LITTAB. For each literal used, this table contains the literal name, the operand value and length, and the address assigned to the operand when it is placed in a literal pool. LITTAB is often organized as a hash table, using the literal name or value as the key.

As each literal operand is recognized during Pass 1, the assembler searches LITTAB for the specified literal name (or value). If the literal is already present in the table, no action is needed; if it is not present, the literal is added to LITTAB (leaving the address unassigned). When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address (unless such an address has already been filled in). As these addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

During Pass 2, the operand address for use in generating object code is obtained by searching LITTAB for each literal operand encountered. The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD statements. If a literal value represents an address in the program (for example, a location counter value), the assembler must also generate the appropriate Modification record.

To be sure you understand how LITTAB is created and used by the assembler, you may want to apply the procedure we just described to the source statements in Fig. 2.9. The object code and literal pools generated should be the same as those in Fig. 2.10.

2.3.2 Symbol-Defining Statements

Up to this point the only user-defined symbols we have seen in assembler language programs have appeared as labels on instructions or data areas. The value of such a label is the address assigned to the statement on which it appears. Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The assembler directive generally used is EQU (for "equate"). The general form of such a statement is

```
symbol    EQU    value
```

This statement defines the given symbol (i.e., enters it into SYMTAB) and assigns to it the value specified. The value may be given as a constant or as any expression involving constants and previously defined symbols. We discuss the formation and use of expressions in the next section.

One common use of EQU is to establish symbolic names that can be used for improved readability in place of numeric values. For example, on line 133 of the program in Fig. 2.5 we used the statement

```
        +LDT        #4096
```

to load the value 4096 into register T. This value represents the maximum-length record we could read with subroutine RDREC. The meaning is not, however, as clear as it might be. If we include the statement

```
MAXLEN    EQU        4096
```

in the program, we can write line 133 as

```
+LDT        #MAXLEN
```

When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB (with value 4096). During assembly of the LDT instruction, the assembler searches SYMTAB for the symbol MAXLEN, using its value as the operand in the instruction. The resulting object code is exactly the same as in the original version of the instruction; however, the source statement is easier to understand. It is also much easier to find and change the value of MAXLEN if this becomes necessary—we would not have to search through the source code looking for places where #4096 is used.

Another common use of EQU is in defining mnemonic names for registers. We have assumed that our assembler recognizes standard mnemonics for registers—A, X, L, etc. Suppose, however, that the assembler expected register *numbers* instead of names in an instruction like RMO. This would require the programmer to write (for example) RMO 0,1 instead of RMO A,X. In such a case the programmer could include a sequence of EQU statements like

```
A       EQU     0
X       EQU     1
L       EQU     2
        .
        .
        .
```

These statements cause the symbols A, X, L,... to be entered into SYMTAB with their corresponding values 0, 1, 2,... . An instruction like RMO A,X would then be allowed. The assembler would search SYMTAB, finding the values 0 and 1 for the symbols A and X, and assemble the instruction.

On a machine like SIC, there would be little point in doing this—it is just as easy to have the standard register mnemonics built into the assembler. Furthermore, the standard names (base, index, etc.) reflect the usage of the registers. Consider, however, a machine that has general-purpose registers. These registers are typically designated by 0, 1, 2,... (or R0, R1, R2,...). In a particular program, however, some of these may be used as base registers, some as index registers, some as accumulators, etc. Furthermore, this usage of registers changes from one program to the next. By writing statements like

```
BASE    EQU     R1
COUNT   EQU     R2
INDEX   EQU     R3
```

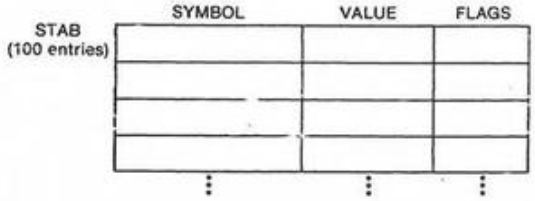
the programmer can establish and use names that reflect the logical function of the registers in the program.

There is another common assembler directive that can be used to indirectly assign values to symbols. This directive is usually called ORG (for "origin"). Its form is

```
ORG     value
```

where *value* is a constant or an expression involving constants and previously defined symbols. When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG.

Of course the location counter is used to control assignment of storage in the object program; in most cases, altering its value would result in an incorrect assembly. Sometimes, however, ORG can be useful in label definition. Suppose that we were defining a symbol table with the following structure:



In this table, the SYMBOL field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAGS is a 2-byte field that specifies symbol type and other information.

We could reserve space for this table with the statement

```
STAB    RESB    1100
```

We might want to refer to entries in the table using indexed addressing (placing in the index register the offset of the desired entry from the beginning of the table). Of course, we want to be able to refer to the fields SYMBOL, VALUE, and FLAGS individually, so we must also define these labels. One way of doing this would be with EQU statements:

```
SYMBOL  EQU     STAB
VALUE   EQU     STAB+6
FLAGS   EQU     STAB+9
```

This would allow us to write, for example,

```
        LDA     VALUE,X
```

to fetch the VALUE field from the table entry indicated by the contents of register X. However, this method of definition simply defines the labels; it does not make the structure of the table as clear as it might be.

We can accomplish the same symbol definition using ORG in the following way:

```
STAB    RESB    1100
        ORG     STAB
SYMBOL  RESB    6
VALUE   RESW    1
FLAGS   RESB    2
        ORG     STAB+1100
```

The first ORG resets the location counter to the value of STAB (i.e., the beginning address of the table). The label on the following RESB statement defines SYMBOL to have the current value in LOCCTR; this is the same address assigned to SYMTAB. LOCCTR is then advanced so the label on the RESW statement assigns to VALUE the address (STAB+6), and so on. The result is a set of labels with the same values as those defined with the EQU statements above. This method of definition makes it clear, however, that each entry in STAB consists of a 6-byte SYMBOL, followed by a one-word VALUE, followed by a 2-byte FLAGS.

The last ORG statement is very important. It sets LOCCTR back to its previous value—the address of the next unassigned byte of memory after the table STAB. This is necessary so that any labels on subsequent statements, which do not represent part of STAB, are assigned the proper addresses. In some assemblers the previous value of LOCCTR is automatically remembered, so we can simply write

```
        ORG
```

(with no value specified) to return to the normal use of LOCCTR.

The descriptions of the EQU and ORG statements contain restrictions that are common to all symbol-defining assembler directives. In the case of EQU, all symbols used on the right-hand side of the statement—that is, all terms used to specify the value of the new symbol—must have been defined previously in the program. Thus, the sequence

```
ALPHA   RESW    1
BETA    EQU     ALPHA
```


would be allowed, whereas the sequence

```
BETA    EQU     ALPHA
ALPHA   RESW    1
```

would not. The reason for this is the symbol definition process. In the second example above, BETA cannot be assigned a value when it is encountered during Pass 1 of the assembly (because ALPHA does not yet have a value). However, our two-pass assembler design requires that all symbols be defined during Pass 1.

A similar restriction applies to ORG: all symbols used to specify the new location counter value must have been previously defined. Thus, for example, the sequence

```
        ORG     ALPHA
BYTE1   RESB    1
BYTE2   RESB    1
BYTE3   RESB    1
        ORG
ALPHA   RESB    1
```

could not be processed. In this case, the assembler would not know (during Pass 1) what value to assign to the location counter in response to the first ORG statement. As a result, the symbols BYTE1, BYTE2, and BYTE3 could not be assigned addresses during Pass 1.

It may appear that this restriction is a result of the particular way in which we defined the two passes of our assembler. In fact, it is a more general product of the forward-reference problem. You can easily see, for example, that the sequence of statements

```
ALPHA   EQU     BETA
BETA    EQU     DELTA
DELTA   RESW    1
```

cannot be resolved by an ordinary two-pass assembler regardless of how the work is divided between the passes. In Section 2.4.2, we briefly consider ways of handling such sequences in a more complex assembler structure.

2.3.3 Expressions

Our previous examples of assembler language statements have used single terms (labels, literals, etc.) as instruction operands. Most assemblers allow the

use of expressions wherever such a single operand is permitted. Each such expression must, of course, be evaluated by the assembler to produce a single operand address or value.

Assemblers generally allow arithmetic expressions formed according to the normal rules using the operators +, –, *, and /. Division is usually defined to produce an integer result. Individual terms in the expression may be constants, user-defined symbols, or special terms. The most common such special term is the current value of the location counter (often designated by *). This term represents the value of the next unassigned memory location. Thus in Fig. 2.9 the statement

```
106     BUFEND    EQU     *
```

gives BUFEND a value that is the address of the next byte after the buffer area.

In Section 2.2 we discussed the problem of program relocation. We saw that some values in the object program are *relative* to the beginning of the program, while others are *absolute* (independent of program location). Similarly, the values of terms and expressions are either relative or absolute. A constant is, of course, an absolute term. Labels on instructions and data areas, and references to the location counter value, are relative terms. A symbol whose value is given by EQU (or some similar assembler directive) may be either an absolute term or a relative term depending upon the expression used to define its value.

Expressions are classified as either *absolute expressions* or *relative expressions* depending upon the type of value they produce. An expression that contains only absolute terms is, of course, an absolute expression. However, absolute expressions may also contain relative terms provided the relative terms occur in pairs and the terms in each such pair have opposite signs. It is not necessary that the paired terms be adjacent to each other in the expression; however, all relative terms must be capable of being paired in this way. None of the relative terms may enter into a multiplication or division operation.

A relative expression is one in which all of the relative terms except one can be paired as described above; the remaining unpaired relative term must have a positive sign. As before, no relative term may enter into a multiplication or division operation. Expressions that do not meet the conditions given for either absolute or relative expressions should be flagged by the assembler as errors.

Although the rules given above may seem arbitrary, they are actually quite reasonable. The expressions that are legal under these definitions include exactly those expressions whose value remains meaningful when the program is relocated. A relative term or expression represents some value that may be written as (S+ *r*), where S is the starting address of the program and *r* is the

value of the term or expression relative to the starting address. Thus a relative term usually represents some location within the program. When relative terms are paired with opposite signs, the dependency on the program starting address is canceled out; the result is an absolute value. Consider, for example, the program of Fig. 2.9. In the statement

```
107     MAXLEN    EQU     BUFEND-BUFFER
```

both BUFEND and BUFFER are relative terms, each representing an address within the program. However, the expression represents an absolute value: the *difference* between the two addresses, which is the length of the buffer area in bytes. Notice that the assembler listing in Fig. 2.10 shows the value calculated for this expression (hexadecimal 1000) in the Loc column. This value does not represent an address, as do most of the other entries in that column. However, it does show the value that is associated with the symbol that appears in the source statement (MAXLEN).

Expressions such as BUFEND + BUFFER, 100 – BUFFER, or 3 * BUFFER represent neither absolute values nor locations within the program. The values of these expressions depend upon the program starting address in a way that is unrelated to anything within the program itself. Because such expressions are very unlikely to be of any use, they are considered errors.

To determine the type of an expression, we must keep track of the types of all symbols defined in the program. For this purpose we need a flag in the symbol table to indicate type of value (absolute or relative) in addition to the value itself. Thus for the program of Fig. 2.10, some of the symbol table entries might be

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

With this information the assembler can easily determine the type of each expression used as an operand and generate Modification records in the object program for relative values.

In Section 2.3.5 we consider programs that consist of several parts that can be relocated independently of each other. As we discuss in the later section, our rules for determining the type of an expression must be modified in such instances.

2.3.4 Program Blocks

In all of the examples we have seen so far the program being assembled was treated as a unit. The source programs logically contained subroutines, data areas, etc. However, they were handled by the assembler as one entity, resulting in a single block of object code. Within this object program the generated machine instructions and data appeared in the same order as they were written in the source program.

Many assemblers provide features that allow more flexible handling of the source and object programs. Some features allow the generated machine instructions and data to appear in the object program in a different order from the corresponding source statements. Other features result in the creation of several independent parts of the object program. These parts maintain their identity and are handled separately by the loader. We use the term *program blocks* to refer to segments of code that are rearranged within a single object program unit, and *control sections* to refer to segments that are translated into independent object program units. (This terminology is, unfortunately, far from uniform. As a matter of fact, in some systems the same assembler language feature is used to accomplish both of these logically different functions.) In this section we consider the use of program blocks and how they are handled by the assembler. Section 2.3.5 discusses control sections and their uses.

Figure 2.11 shows our example program as it might be written using program blocks. In this case three blocks are used. The first (unnamed) program block contains the executable instructions of the program. The second (named CDATA) contains all data areas that are a few words or less in length. The third (named CBLKS) contains all data areas that consist of larger blocks of memory. Some possible reasons for making such a division are discussed later in this section.

The assembler directive USE indicates which portions of the source program belong to the various blocks. At the beginning of the program, statements are assumed to be part of the unnamed (default) block; if no USE statements are included, the entire program belongs to this single block. The USE statement on line 92 signals the beginning of the block named CDATA. Source statements are associated with this block until the USE statement on line 103, which begins the block named CBLKS. The USE statement may also indicate a continuation of a previously begun block. Thus the statement on line 123 resumes the default block, and the statement on line 183 resumes the block named CDATA.

As we can see, each program block may actually contain several separate segments of the source program. The assembler will (logically) rearrange these segments to gather together the pieces of each block. These blocks will then be assigned addresses in the object program, with the blocks appearing in the


```
Line        Source statement
  5   COPY     START    0                 COPY FILE FROM INPUT TO OUTPUT
 10   FIRST    STL      RETADR            SAVE RETURN ADDRESS
 15   CLOOP    JSUB     RDREC             READ INPUT RECORD
 20            LDA      LENGTH            TEST FOR EOF (LENGTH = 0)
 25            COMP     #0
 30            JEQ      ENDFIL            EXIT IF EOF FOUND
 35            JSUB     WRREC             WRITE OUTPUT RECORD
 40            J        CLOOP             LOOP
 45   ENDFIL   LDA      =C'EOF'           INSERT END OF FILE MARKER
 50            STA      BUFFER
 55            LDA      #3                SET LENGTH = 3
 60            STA      LENGTH
 65            JSUB     WRREC             WRITE EOF
 70            J        @RETADR           RETURN TO CALLER
 92            USE      CDATA
 95   RETADR   RESW     1
100   LENGTH   RESW     1                 LENGTH OF RECORD
103            USE      CBLKS
105   BUFFER   RESB     4096              4096-BYTE BUFFER AREA
106   BUFEND   EQU      *                 FIRST LOCATION AFTER BUFFER
107   MAXLEN   EQU      BUFEND-BUFFER     MAXIMUM RECORD LENGTH
110   .
115   .        SUBROUTINE TO READ RECORD INTO BUFFER
120   .
123            USE
125   RDREC    CLEAR    X                 CLEAR LOOP COUNTER
130            CLEAR    A                 CLEAR A TO ZERO
132            CLEAR    S                 CLEAR S TO ZERO
133            +LDT     #4096
135   RLOOP    TD       INPUT             TEST INPUT DEVICE
140            JEQ      RLOOP             LOOP UNTIL READY
145            RD       INPUT             READ CHARACTER INTO REGISTER A
150            COMPR    A,S               TEST FOR END OF RECORD (X'00')
155            JEQ      EXIT              EXIT LOOP IF EOR
160            STCH     BUFFER,X          STORE CHARACTER IN BUFFER
165            TIXR     T                 LOOP UNLESS MAX LENGTH
170            JLT      RLOOP               HAS BEEN REACHED
175   EXIT     STX      LENGTH            SAVE RECORD LENGTH
180            RSUB                       RETURN TO CALLER
183            USE      CDATA
185   INPUT    BYTE     X'F1'             CODE FOR INPUT DEVICE
195   .
200   .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205   .
208            USE
210   WRREC    CLEAR    X                 CLEAR LOOP COUNTER
212            LDT      LENGTH
215   WLOOP    TD       =X'05'            TEST OUTPUT DEVICE
220            JEQ      WLOOP             LOOP UNTIL READY
225            LDCH     BUFFER,X          GET CHARACTER FROM BUFFER
230            WD       =X'05'            WRITE CHARACTER
235            TIXR     T                 LOOP UNTIL ALL CHARACTERS
240            JLT      WLOOP               HAVE BEEN WRITTEN
245            RSUB                       RETURN TO CALLER
252            USE      CDATA
253            LTORG
255            END      FIRST
```

Figure 2.11 Example of a program with multiple program blocks.

same order in which they were first begun in the source program. The result is the same as if the programmer had physically rearranged the source statements to group together all the source lines belonging to each block.

The assembler accomplishes this logical rearrangement of code by maintaining, during Pass 1, a separate location counter for each program block. The location counter for a block is initialized to 0 when the block is first begun. The current value of this location counter is saved when switching to another block, and the saved value is restored when resuming a previous block. Thus during Pass 1 each label in the program is assigned an address that is relative to the start of the block that contains it. When labels are entered into the symbol table, the block name or number is stored along with the assigned relative address. At the end of Pass 1 the latest value of the location counter for each block indicates the length of that block. The assembler can then assign to each block a starting address in the object program (beginning with relative location 0).

For code generation during Pass 2, the assembler needs the address for each symbol relative to the start of the object program (not the start of an individual program block). This is easily found from the information in SYMTAB. The assembler simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address.

Figure 2.12 demonstrates this process applied to our sample program. The column headed Loc/Block shows the relative address (within a program block) assigned to each source line and a block number indicating which program block is involved (0 = default block, 1 = CDATA, 2 = CBLKS). This is essentially the same information that is stored in SYMTAB for each symbol. Notice that the value of the symbol MAXLEN (line 107) is shown without a block number. This indicates that MAXLEN is an absolute symbol, whose value is not relative to the start of any program block.

At the end of Pass 1 the assembler constructs a table that contains the starting addresses and lengths for all blocks. For our sample program, this table looks like

Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

Now consider the instruction

```
20    0006  0    LDA    LENGTH    032060
```

Line	Loc/Block		Source statement			Object code
5	0000	0	COPY	START	0	
10	0000	0	FIRST	STL	RETADR	172063
15	0003	0	CLOOP	JSUB	RDREC	4B2021
20	0006	0		LDA	LENGTH	032060
25	0009	0		COMP	#0	290000
30	000C	0		JEQ	ENDFIL	332006
35	000F	0		JSUB	WRREC	4B203B
40	0012	0		J	CLOOP	3F2FEE
45	0015	0	ENDFIL	LDA	=C'EOF'	032055
50	0018	0		STA	BUFFER	0F2056
55	001B	0		LDA	#3	010003
60	001E	0		STA	LENGTH	0F2048
65	0021	0		JSUB	WRREC	4B2029
70	0024	0		J	@RETADR	3E203F
92	0000	1		USE	CDATA	
95	0000	1	RETADR	RESW	1	
100	0003	1	LENGTH	RESW	1	
103	0000	2		USE	CBLKS	
105	0000	2	BUFFER	RESB	4096	
106	1000	2	BUFEND	EQU	*	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	
110			.			
115			.	SUBROUTINE TO READ RECORD INTO BUFFER		
120			.			
123	0027	0		USE		
125	0027	0	RDREC	CLEAR	X	B410
130	0029	0		CLEAR	A	B400
132	002B	0		CLEAR	S	B440
133	002D	0		+LDT	#MAXLEN	75101000
135	0031	0	RLOOP	TD	INPUT	E32038
140	0034	0		JEQ	RLOOP	332FFA
145	0037	0		RD	INPUT	DB2032
150	003A	0		COMPR	A,S	A004
155	003C	0		JEQ	EXIT	332008
160	003F	0		STCH	BUFFER,X	57A02F
165	0042	0		TIXR	T	B850
170	0044	0		JLT	RLOOP	3B2FEA
175	0047	0	EXIT	STX	LENGTH	13201F
180	004A	0		RSUB		4F0000
183	0006	1		USE	CDATA	
185	0006	1	INPUT	BYTE	X'F1'	F1
195			.			
200			.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205			.			
208	004D	0		USE		
210	004D	0	WRREC	CLEAR	X	B410
212	004F	0		LDT	LENGTH	772017
215	0052	0	WLOOP	TD	=X'05'	E3201B
220	0055	0		JEQ	WLOOP	332FFA
225	0058	0		LDCH	BUFFER,X	53A016
230	005B	0		WD	=X'05'	DF2012
235	005E	0		TIXR	T	B850
240	0060	0		JLT	WLOOP	3B2FEF
245	0063	0		RSUB		4F0000
252	0007	1		USE	CDATA	
253				LTORG		
	0007	1	*	=C'EOF'		454F46
	000A	1	*	=X'05'		05
255				END	FIRST	

Figure 2.12 Program from Fig. 2.11 with object code.

SYMTAB shows the value of the operand (the symbol LENGTH) as relative location 0003 within program block 1 (CDATA). The starting address for CDATA is 0066. Thus the desired target address for this instruction is 0003 + 0066 = 0069. The instruction is to be assembled using program-counter relative addressing. When the instruction is executed, the program counter contains the address of the following instruction (line 25). The address of this instruction is relative location 0009 within the default block. Since the default block starts at location 0000, this address is simply 0009. Thus the required displacement is 0069 – 0009 = 60. The calculation of the other addresses during Pass 2 follows a similar pattern.

We can immediately see that the separation of the program into blocks has considerably reduced our addressing problems. Because the large buffer area is moved to the end of the object program, we no longer need to use extended format instructions on lines 15, 35, and 65. Furthermore, the base register is no longer necessary; we have deleted the LDB and BASE statements previously on lines 13 and 14. The problem of placement of literals (and literal references) in the program is also much more easily solved. We simply include a LTORG statement in the CDATA block to be sure that the literals are placed ahead of any large data areas.

Of course the use of program blocks has not accomplished anything we could not have done by rearranging the statements of the source program. For example, program readability is often improved if the definitions of data areas are placed in the source program close to the statements that reference them. This could be accomplished in a long subroutine (without using program blocks) by simply inserting data areas in any convenient position. However, the programmer would need to provide Jump instructions to branch around the storage thus reserved.

In the situation just discussed, machine considerations suggested that the parts of the object program appear in memory in a particular order. On the other hand, human factors suggested that the source program should be in a different order. The use of program blocks is one way of satisfying both of these requirements, with the assembler providing the required reorganization.

It is not necessary to physically rearrange the generated code in the object program to place the pieces of each program block together. The assembler can simply write the object code as it is generated during Pass 2 and insert the proper load address in each Text record. These load addresses will, of course, reflect the starting address of the block as well as the relative location of the code within the block. This process is illustrated in Fig. 2.13. The first two Text records are generated from the source program lines 5 through 70. When the USE statement on line 92 is recognized, the assembler writes out the current Text record (even though there is still room left in it). The assembler then prepares to begin a new Text record for the new program block. As it happens, the statements on lines 95 through 105 result in no generated code, so no new Text


```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEB^032055^0F2056^010003
T^00001E^09^0F2048^4B2029^3E203F
T^000027^1D^B410^B400^B440^75101000^E32038^332FFA^DB2032^A004^332008^57A02F^B850
T^000044^09^3B2FEA^13201F^4F0000
T^00006C^01^F1
T^00004D^19^B410^77201F^E3201B^332FFA^53A016^DF2012^B8503B^2FEF^4F0000
T^00006D^04^454F46^05
E^000000
```

Figure 2.13 Object program corresponding to Fig. 2.11.

records are created. The next two Text records come from lines 125 through 180. This time the statements that belong to the next program block do result in the generation of object code. The fifth Text record contains the single byte of data from line 185. The sixth Text record resumes the default program block and the rest of the object program continues in similar fashion.

It does not matter that the Text records of the object program are not in sequence by address; the loader will simply load the object code from each record at the indicated address. When this loading is completed, the generated code from the default block will occupy relative locations 0000 through 0065; the generated code and reserved storage for CDATA will occupy locations 0066 through 0070; and the storage reserved for CBLKS will occupy locations 0071 through 1070. Figure 2.14 traces the blocks of the example program through this process of assembly and loading. Notice that the program segments marked CDATA(1) and CBLKS(1) are not actually present in the object program. Because of the way the addresses are assigned, storage will automatically be reserved for these areas when the program is loaded.

You should carefully examine the generated code in Fig. 2.12, and work through the assembly of several more instructions to be sure you understand how the assembler handles multiple program blocks. To understand how the pieces of each program block are gathered together, you may also want to simulate (by hand) the loading of the object program of Fig. 2.13.

2.3.5 Control Sections and Program Linking

In this section, we discuss the handling of programs that consist of multiple control sections. A *control section* is a part of the program that maintains its identity after assembly; each such control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions of a program. The programmer can assemble, load, and manipulate each of these control sections separately. The

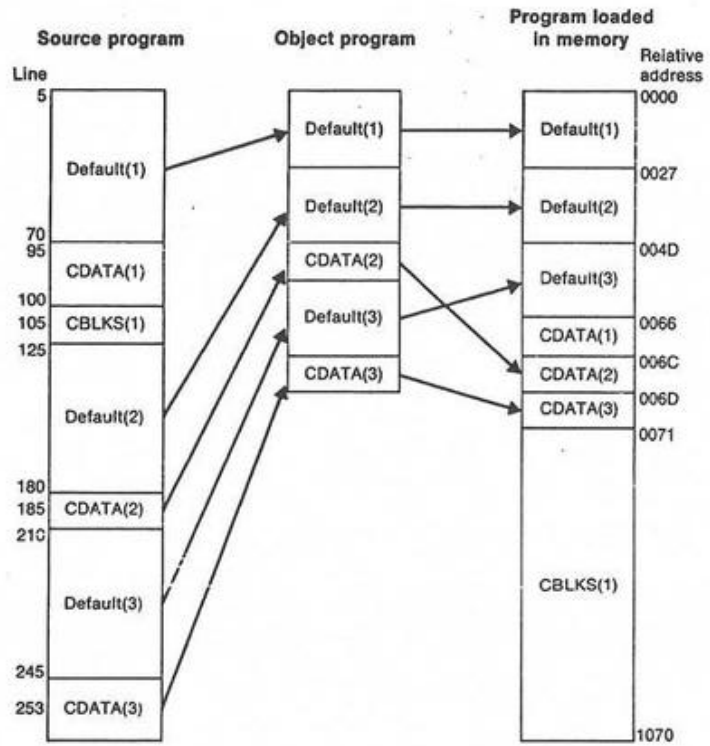


Figure 2.14 Program blocks from Fig. 2.11 traced through the assembly and loading processes.

resulting flexibility is a major benefit of using control sections. We consider examples of this when we discuss linkage editors in Chapter 3.

When control sections form logically related parts of a program, it is necessary to provide some means for *linking* them together. For example, instructions in one control section might need to refer to instructions or data located in another section. Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. The assembler has no idea where any other control section will be located at execution time. Such references between control sections are called *external references*. The assembler generates information for each external reference that will allow the loader to perform the required linking. In this section we describe how external references are handled by our assembler. Chapter 3 discusses in detail how the actual linking is performed.

Figure 2.15 shows our example program as it might be written using multiple control sections. In this case there are three control sections: one for the main program and one for each subroutine. The START statement identifies the beginning of the assembly and gives a name (COPY) to the first control section. The first section continues until the CSECT statement on line 109. This assembler directive signals the start of a new control section named RDREC. Similarly, the CSECT statement on line 193 begins the control section named WRREC. The assembler establishes a separate location counter (beginning at 0) for each control section, just as it does for program blocks.

Control sections differ from program blocks in that they are handled separately by the assembler. (It is not even necessary for all control sections in a program to be assembled at the same time.) Symbols that are defined in one control section may not be used directly by another control section; they must be identified as external references for the loader to handle. Figure 2.15 shows the use of two assembler directives to identify such references: EXTDEF (external definition) and EXTREF (external reference). The EXTDEF statement in a control section names symbols, called *external symbols*, that are defined in this control section and may be used by other sections. Control section names (in this case COPY, RDREC, and WRREC) do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols. The EXTREF statement names symbols that are used in this control section and are defined elsewhere. For example, the symbols BUFFER, BUFEND, and LENGTH are defined in the control section named COPY and made available to the other sections by the EXTDEF statement on line 6. The third control section (WRREC) uses two of these symbols, as specified in its EXTREF statement (line 207). The order in which symbols are listed in the EXTDEF and EXTREF statements is not significant.

Now we are ready to look at how external references are handled by the assembler. Figure 2.16 shows the generated object code for each statement in the program. Consider first the instruction

```
15    0003    CLOOP    +JSUB    RDREC          4B100000
```

The operand (RDREC) is named in the EXTREF statement for the control section, so this is an external reference. The assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address for this instruction. Instead the assembler inserts an address of zero and passes information to the loader, which will cause the proper address to be inserted at load time. The address of RDREC will have no predictable relationship to anything in this control section; therefore relative addressing is not possible. Thus an extended format instruction must be used to provide room for the actual address to be inserted. This is true of any instruction whose operand involves an external reference.

Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
6		EXTDEF	BUFFER,BUFEND,LENGTH	
7		EXTREF	RDREC,WRREC	
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	=C'EOF'	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
103		LTORG		
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
106	BUFEND	EQU	*	
107	MAXLEN	EQU	BUFEND-BUFFER	
109	RDREC	CSECT		
110	.			
115	.	SUBROUTINE TO READ RECORD INTO BUFFER		
120	.			
122		EXTREF	BUFFER,LENGTH,BUFEND	
125		CLEAR	X	CLEAR LOOP COUNTER
130		CLEAR	A	CLEAR A TO ZERO
132		CLEAR	S	CLEAR S TO ZERO
133		LDT	MAXLEN	
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		JEQ	RLOOP	LOOP UNTIL READY
145		RD	INPUT	READ CHARACTER INTO REGISTER A
150		COMPR	A,S	TEST FOR END OF RECORD (X'00')
155		JEQ	EXIT	EXIT LOOP IF EOR
160		+STCH	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIXR	T	LOOP UNLESS MAX LENGTH
170		JLT	RLOOP	HAS BEEN REACHED
175	EXIT	+STX	LENGTH	SAVE RECORD LENGTH
180		RSUB		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
190	MAXLEN	WORD	BUFEND-BUFFER	
193	WRREC	CSECT		
195	.			
200	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.			
207		EXTREF	LENGTH,BUFFER	
210		CLEAR	X	CLEAR LOOP COUNTER
212		+LDT	LENGTH	
215	WLOOP	TD	=X'05'	TEST OUTPUT DEVICE
220		JEQ	WLOOP	LOOP UNTIL READY
225		+LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
230		WD	=X'05'	WRITE CHARACTER
235		TIXR	T	LOOP UNTIL ALL CHARACTERS
240		JLT	WLOOP	HAVE BEEN WRITTEN
245		RSUB		RETURN TO CALLER
255		END	FIRST	

Figure 2.15 Illustration of control sections and program linking.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
109	0000	RDREC	CSECT		
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
122			EXTREF	BUFFER,LENGTH,BUFEND	
125	0000		CLEAR	X	B410
130	0002		CLEAR	A	B400
132	0004		CLEAR	S	B440
133	0006		LDT	MAXLEN	77201F
135	0009	RLOOP	TD	INPUT	E3201B
140	000C		JEQ	RLOOP	332FFA
145	000F		RD	INPUT	DB2015
150	0012		COMPR	A,S	A004
155	0014		JEQ	EXIT	332009
160	0017		+STCH	BUFFER,X	57900000
165	001B		TIXR	T	B850
170	001D		JLT	RLOOP	3B2FE9
175	0020	EXIT	+STX	LENGTH	13100000
180	0024		RSUB		4F0000
185	0027	INPUT	BYTE	X'F1'	F1
190	0028	MAXLEN	WORD	BUFEND-BUFFER	000000
193	0000	WRREC	CSECT		
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
207			EXTREF	LENGTH,BUFFER	
210	0000		CLEAR	X	B410
212	0002		+LDT	LENGTH	77100000
215	0006	WLOOP	TD	=X'05'	E32012
220	0009		JEQ	WLOOP	332FFA
225	000C		+LDCH	BUFFER,X	53900000
230	0010		WD	=X'05'	DF2008
235	0013		TIXR	T	B850
240	0015		JLT	WLOOP	3B2FEE
245	0018		RSUB		4F0000
255			END	FIRST	
	001B	*	=X'05'		05

Figure 2.16 Program from Fig. 2.15 with object code.

Similarly, the instruction

```
160     0017            +STCH    BUFFER,X          57900000
```

makes an external reference to BUFFER. The instruction is assembled using extended format with an address of zero. The x bit is set to 1 to indicate indexed addressing, as specified by the instruction. The statement

```
190     0028    MAXLEN   WORD     BUFEND-BUFFER     000000
```

is only slightly different. Here the value of the data word to be generated is specified by an expression involving two external references: BUFEND and BUFFER. As before, the assembler stores this value as zero. When the program is loaded, the loader will add to this data area the address of BUFEND and subtract from it the address of BUFFER, which results in the desired value.

Note the difference between the handling of the expression on line 190 and the similar expression on line 107. The symbols BUFEND and BUFFER are defined in the same control section with the EQU statement on line 107. Thus the value of the expression can be calculated immediately by the assembler. This could not be done for line 190; BUFEND and BUFFER are defined in another control section, so their values are unknown at assembly time.

As we can see from the above discussion, the assembler must remember (via entries in SYMTAB) in which control section a symbol is defined. Any attempt to refer to a symbol in another control section must be flagged as an error unless the symbol is identified (using EXTREF) as an external reference. The assembler must also allow the same symbol to be used in different control sections. For example, the conflicting definitions of MAXLEN on lines 107 and 190 should cause no problem. A reference to MAXLEN in the control section COPY would use the definition on line 107, whereas a reference to MAXLEN in RDREC would use the definition on line 190.

So far we have seen how the assembler leaves room in the object code for the values of external symbols. The assembler must also include information in the object program that will cause the loader to insert the proper values where they are required. We need two new record types in the object program and a change in a previously defined record type. As before, the exact format of these records is arbitrary; however, the same information must be passed to the loader in some form.

The two new record types are Define and Refer. A Define record gives information about external symbols that are defined in this control section—that is, symbols named by EXTDEF. A Refer record lists symbols that are used as external references by the control section—that is, symbols named by EXTREF. The formats of these records are as follows.

Define record:

Col. 1	D
Col. 2–7	Name of external symbol defined in this control section
Col. 8–13	Relative address of symbol within this control section (hexadecimal)
Col. 14–73	Repeat information in Col. 2–13 for other external symbols

Refer record:

Col. 1	R
Col. 2–7	Name of external symbol referred to in this control section
Col. 8–73	Names of other external reference symbols

The other information needed for program linking is added to the Modification record type. The new format is as follows.

Modification record (revised):

Col. 1	M
Col. 2–7	Starting address of the field to be modified, relative to the beginning of the control section (hexadecimal)
Col. 8–9	Length of the field to be modified, in half-bytes (hexadecimal)
Col. 10	Modification flag (+ or –)
Col. 11–16	External symbol whose value is to be added to or subtracted from the indicated field

The first three items in this record are the same as previously discussed. The two new items specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification may be defined either in this control section or in another one.

Figure 2.17 shows the object program corresponding to the source in Fig. 2.16. Notice that there is a separate set of object program records (from Header through End) for each control section. The records for each control section are exactly the same as they would be if the sections were assembled separately.

The Define and Refer records for each control section include the symbols named in the EXTDEF and EXTREF statements. In the case of Define, the record also indicates the relative address of each external symbol within the control section. For EXTREF symbols, no address information is available. These symbols are simply named in the Refer record.

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^0000001D^172027^4B1000000^3202329000^0332007^4B1000003^F2FEC^032016^0F2016
T^00001D^0D^0100030^F200A^4B1000003^E2000
T^000030^03^454F46
M^00000405^+RDREC
M^00001105^+WRREC
M^00002405^+WRREC
E^000000


H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^0000001D^B410^8400^B44077201^FE3201^B332FFA^DB2015^A004332009^57900000^B850
T^00001D^0E^3B2FE9^131000004^F0000F^1000000
M^00001805^+BUFFER
M^00002105^+LENGTH
M^00002806^+BUFEND
M^00002806^-BUFFER
E


H^WRREC ^000000^00001C
R^LENGTH^BUFFER
T^0000001C^B4107^7100000^E32012^332FFA^53900000^DF2008^B8503^B2FEE^4F0000^05
M^00000305^+LENGTH
M^00000D05^+BUFFER
E
```

Figure 2.17 Object program corresponding to Fig. 2.15.

Now let us examine the process involved in linking up external references, beginning with the source statements we discussed previously. The address field for the JSUB instruction on line 15 begins at relative address 0004. Its initial value in the object program is zero. The Modification record

```
M00000405+RDREC
```

in control section COPY specifies that the address of RDREC is to be added to this field, thus producing the correct machine instruction for execution. The other two Modification records in COPY perform similar functions for the

instructions on lines 35 and 65. Likewise, the first Modification record in control section RDREC fills in the proper address for the external reference on line 160.

The handling of the data word generated by line 190 is only slightly different. The value of this word is to be BUFEND–BUFFER, where both BUFEND and BUFFER are defined in another control section. The assembler generates an initial value of zero for this word (located at relative address 0028 within control section RDREC). The last two Modification records in RDREC direct that the address of BUFEND be added to this field, and the address of BUFFER be subtracted from it. This computation, performed at load time, results in the desired value for the data word.

In Chapter 3 we discuss in detail how the required modifications are performed by the loader. At this time, however, you should be sure that you understand the concepts involved in the linking process. You should carefully examine the other Modification records in Fig. 2.17, and reconstruct for yourself how they were generated from the source program statements.

Note that the revised Modification record may still be used to perform program relocation. In the case of relocation, the modification required is adding the beginning address of the control section to certain fields in the object program. The symbol used as the name of the control section has as its value the required address. Since the control section name is automatically an external symbol, it is available for use in Modification records. Thus, for example, the Modification records from Fig. 2.8 are changed from

```
M00000705
M00001405
M00002705
```

to

```
M00000705+COPY
M00001405+COPY
M00002705+COPY
```

In this way, exactly the same mechanism can be used for program relocation and for program linking. There are more examples in the next chapter.

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions slightly more complicated. Our earlier definitions required that all of the relative terms in an expression be paired (for an absolute expression), or that all except one be paired (for a relative expression). We must now extend this restriction to specify that both terms in each pair must be relative within the same control sec-

tion. The reason is simple—if the two terms represent relative locations in the same control section, their difference is an absolute value (regardless of where the control section is located). On the other hand, if they are in different control sections, their difference has a value that is unpredictable (and therefore probably useless). For example, the expression

```
BUFEND-BUFFER
```

has as its value the length of BUFFER in bytes. On the other hand, the value of the expression

```
RDREC-COPY
```

is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use whatsoever to an application program.

When an expression involves external references, the assembler cannot in general determine whether or not the expression is legal. The pairing of relative terms to test legality cannot be done without knowing which of the terms occur in the same control sections, and this is unknown at assembly time. In such a case, the assembler evaluates all of the terms it can, and combines these to form an initial expression value. It also generates Modification records so the loader can finish the evaluation. The loader can then check the expression for errors. We discuss this further in Chapter 3 when we examine the design of a linking loader.

2.4 ASSEMBLER DESIGN OPTIONS

In this section we discuss two alternatives to the standard two-pass assembler logic. Section 2.4.1 describes the structure and logic of one-pass assemblers. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program. Section 2.4.2 introduces the notion of a multi-pass assembler, an extension to the two-pass logic that allows an assembler to handle forward references during symbol definition.

2.4.1 One-Pass Assemblers

In this section we examine the structure and design of one-pass assemblers. As we discussed in Section 2.1, the main problem in trying to assemble a program in one pass involves forward references. Instruction operands often are symbols that have not yet been defined in the source program. Thus the assembler does not know what address to insert in the translated instruction.

It is easy to eliminate forward references to data items; we can simply require that all such areas be defined in the source program before they are referenced. This restriction is not too severe. The programmer merely places all storage reservation statements at the start of the program rather than at the end. Unfortunately, forward references to labels on instructions cannot be eliminated as easily. The logic of the program often requires a forward jump—for example, in escaping from a loop after testing some condition. Requiring that the programmer eliminate all such forward jumps would be much too restrictive and inconvenient. Therefore, the assembler must make some special provision for handling forward references. To reduce the size of the problem, many one-pass assemblers do, however, prohibit (or at least discourage) forward references to data items.

There are two main types of one-pass assembler. One type produces object code directly in memory for immediate execution; the other type produces the usual kind of object program for later execution. We use the program in Fig. 2.18 to illustrate our discussion of both types. This example is the same as in Fig. 2.2, with all data item definitions placed ahead of the code that references them. The generated object code shown in Fig. 2.18 is for reference only; we will discuss how each type of one-pass assembler would actually generate the object program required.

We first discuss one-pass assemblers that generate their object code in memory for immediate execution. No object program is written out, and no loader is needed. This kind of *load-and-go* assembler is useful in a system that is oriented toward program development and testing. A university computing system for student use is a typical example of such an environment. In such a system, a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration. A load-and-go assembler avoids the overhead of writing the object program out and reading it back in. This can be accomplished with either a one- or a two-pass assembler. However, a one-pass assembler also avoids the overhead of an additional pass over the source program.

Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward references becomes less difficult. The assembler simply generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled. The symbol used as an operand is entered into the symbol table (unless such an entry is already present). This entry is flagged to indicate that the symbol is undefined. The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry. When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists), and the proper address is inserted into any instructions previously generated.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
121	2039	INPUT	BYTE	X'F1'	F1
122	203A	MAXLEN	WORD	4096	001000
124		.			
125	203D	RDREC	LDX	ZERO	041006
130	2040		LDA	ZERO	001006
135	2043	RLOOP	TD	INPUT	E02039
140	2046		JEQ	RLOOP	302043
145	2049		RD	INPUT	D82039
150	204C		COMP	ZERO	281006
155	204F		JEQ	EXIT	30205B
160	2052		STCH	BUFFER,X	54900F
165	2055		TIX	MAXLEN	2C203A
170	2058		JLT	RLOOP	382043
175	205B	EXIT	STX	LENGTH	10100C
180	205E		RSUB		4C0000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
206	2061	OUTPUT	BYTE	X'05'	05
207		.			
210	2062	WRREC	LDX	ZERO	041006
215	2065	WLOOP	TD	OUTPUT	E02061
220	2068		JEQ	WLOOP	302065
225	206B		LDCH	BUFFER,X	50900F
230	206E		WD	OUTPUT	DC2061
235	2071		TIX	LENGTH	2C100C
240	2074		JLT	WLOOP	382065
245	2077		RSUB		4C0000
255			END	FIRST	

Figure 2.18 Sample program for a one-pass assembler.

An example should help to make this process clear. Figure 2.19(a) shows the object code and symbol table entries as they would be after scanning line 40 of the program in Fig. 2.18. The first forward reference occurred on line 15. Since the operand (RDREC) was not yet defined, the instruction was assembled with no value assigned as the operand address (denoted in the figure by —). RDREC was then entered into SYMTAB as an undefined symbol (indicated by *); the address of the operand field of the instruction (2013) was inserted in a list associated with RDREC. A similar process was followed with the instructions on lines 30 and 35.

Now consider Fig. 2.19(b), which corresponds to the situation after scanning line 160. Some of the forward references have been resolved by this time, while others have been added. When the symbol ENDFIL was defined (line 45), the assembler placed its value in the SYMTAB entry; it then inserted this value into the instruction operand field (at address 201C) as directed by the forward reference list. From this point on, any references to ENDFIL would not be forward references, and would not be entered into a list. Similarly, the definition of RDREC (line 125) resulted in the filling in of the operand address at location 2013. Meanwhile, two new forward references have been added: to WRREC (line 65) and EXIT (line 155). You should continue tracing through this process to the end of the program to show yourself that all of the forward

Memory address	Contents			
1000	454F4600	00030000	00xxxxxx	xxxxxxxx
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
•				
•				
•				
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14
2010	100948—	—00100C	28100630	——48—
2020	—3C2012			
•				
•				
•				

Symbol	Value
LENGTH	100C
RDREC	* → 2013 0
THREE	1003
ZERO	1006
WRREC	* → 201F 0
EOF	1000
ENDFIL	* → 201C 0
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F

Figure 2.19(a) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 40.

Memory address	Contents			
1000	454F4600	00030000	00xxxxxx	xxxxxxxx
1010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
•				
•				
•				
2000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxx14
2010	10094820	3D00100C	28100630	202448—
2020	—3C2012	0010000C	100F0010	030C100C
2030	48——08	10094C00	00F10010	00041006
2040	001006E0	20393020	43D82039	28100630
2050	——5490	0F		
•				
•				
•				

Symbol	Value
LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	* → 201F → 2031 | 0
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F
MAXLEN	203A
INPUT	2039
EXIT	* → 2050 | 0
RLOOP	2043

Figure 2.19(b) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 160.

references will be filled in properly. At the end of the program, any SYMTAB entries that are still marked with * indicate undefined symbols. These should be flagged by the assembler as errors.

When the end of the program is encountered, the assembly is complete. If no errors have occurred, the assembler searches SYMTAB for the value of the symbol named in the END statement (in this case, FIRST) and jumps to this location to begin execution of the assembled program.

We used an absolute program as our example because, for a load-and-go assembler, the actual address must be known at assembly time. Of course it is not necessary for this address to be specified by the programmer; it might be assigned by the system. In either case, however, the assembly process would be the same—the location counter would be initialized to the actual program starting address.

One-pass assemblers that produce object programs as output are often used on systems where external working-storage devices (for the intermediate file between the two passes) are not available. Such assemblers may also be

useful when the external storage is slow or is inconvenient to use for some other reason. One-pass assemblers that produce object programs follow a slightly different procedure from that previously described. Forward references are entered into lists as before. Now, however, when the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer be available in memory for modification. In general, they will already have been written out as part of a Text record in the object program. In this case the assembler must generate another Text record with the correct operand address. When the program is loaded, this address will be inserted into the instruction by the action of the loader.

Figure 2.20 illustrates this process. The second Text record contains the object code generated from lines 10 through 40 in Fig. 2.18. The operand addresses for the instructions on lines 15, 30, and 35 have been generated as 0000. When the definition of ENDFIL on line 45 is encountered, the assembler generates the third Text record. This record specifies that the value 2024 (the address of ENDFIL) is to be loaded at location 201C (the operand address field of the JEQ instruction on line 30). When the program is loaded, therefore, the value 2024 will replace the 0000 previously loaded. The other forward references in the program are handled in exactly the same way. In effect, the services of the loader are being used to complete forward references that could not be handled by the assembler. Of course, the object program records must be kept in their original order when they are presented to the loader.

In this section we considered only simple one-pass assemblers that handled absolute programs. Instruction operands were assumed to be single symbols, and the assembled instructions contained the actual (not relative) addresses of the operands. More advanced assembler features such as literals

```
H^COPY  ^00100^00107A
T^00100^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

Figure 2.20 Object program from one-pass assembler for program in Fig. 2.18.

were not allowed. You are encouraged to think about ways of removing some of these restrictions (see the Exercises for this section for some suggestions).

2.4.2 Multi-Pass Assemblers

In our discussion of the EQU assembler directive, we required that any symbol used on the right-hand side (i.e., in the expression giving the value of the new symbol) be defined previously in the source program. A similar requirement was imposed for ORG. As a matter of fact, such a restriction is normally applied to all assembler directives that (directly or indirectly) define symbols.

The reason for this is the symbol definition process in a two-pass assembler. Consider, for example, the sequence

```
ALPHA   EQU     BETA
BETA    EQU     DELTA
DELTA   RESW    1
```

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined. As a result, ALPHA cannot be evaluated during the second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.

Restrictions such as prohibiting forward references in symbol definition are not normally a serious inconvenience for the programmer. As a matter of fact, such forward references tend to create difficulty for a person reading the program as well as for the assembler. Nevertheless, some assemblers are designed to eliminate the need for such restrictions. The general solution is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols. It is not necessary for such an assembler to make more than two passes over the entire program. Instead, the portions of the program that involve forward references in symbol definition are saved during Pass 1. Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

There are several ways of accomplishing the task outlined above. The method we describe involves storing those symbol definitions that involve forward references in the symbol table. This table also indicates which symbols are dependent on the values of others, to facilitate symbol evaluation.

Figure 2.21(a) shows a sequence of symbol-defining statements that involve forward references; the other parts of the source program are not important for our discussion, and have been omitted. The following parts of Fig. 2.21 show information in the symbol table as it might appear after processing each of the source statements shown.

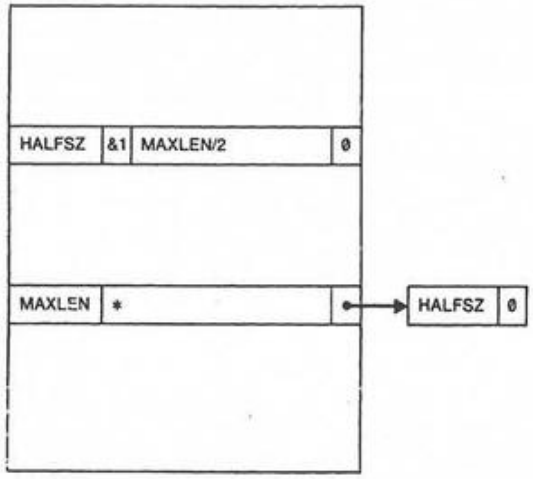
Figure 2.21(b) displays symbol table entries resulting from Pass 1 processing of the statement


```
HALFSZ   EQU   MAXLEN/2
```

MAXLEN has not yet been defined, so no value for HALFSZ can be computed. The defining expression for HALFSZ is stored in the symbol table in place of its value. The entry &1 indicates that one symbol in the defining expression is undefined. In an actual implementation, of course, this definition might be stored at some other location. SYMTAB would then simply contain a pointer to the defining expression. The symbol MAXLEN is also entered in the symbol table, with the flag * identifying it as undefined. Associated with this entry is a list of the symbols whose values depend on MAXLEN (in this case, HALFSZ). (Note the similarity to the way we handled forward references in a one-pass assembler.)

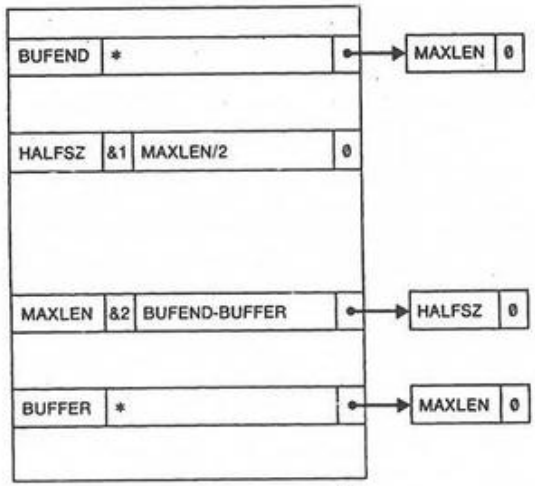
```
1   HALFSZ   EQU   MAXLEN/2
2   MAXLEN   EQU   BUFEND-BUFFER
3   PREVBT   EQU   BUFFER-1
                   .
                   .
                   .
4   BUFFER   RESB  4096
5   BUFEND   EQU   *
```

(a)

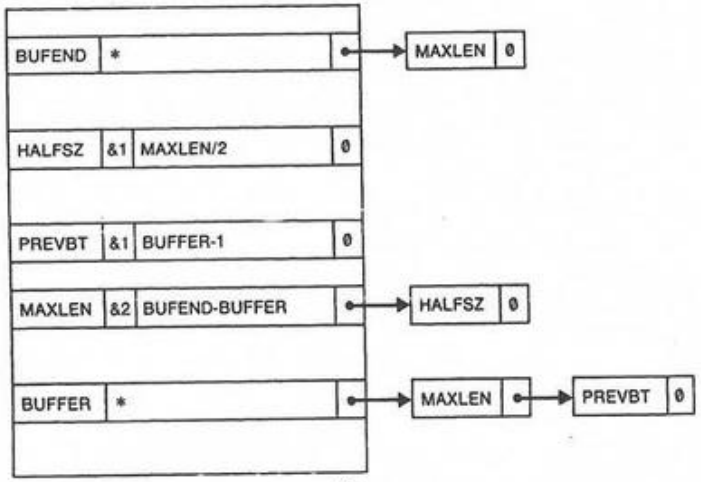


(b)

Figure 2.21 Example of multi-pass assembler operation.

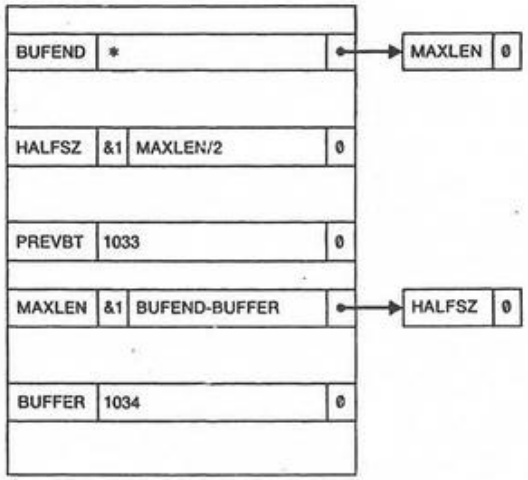


(c)

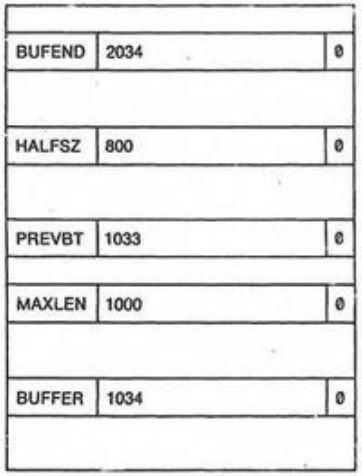


(d)

Figure 2.21 (*cont'd*)



(e)



(f)

Figure 2.21 (*con'd*)

The same procedure is followed with the definition of MAXLEN [see Fig. 2.21(c)]. In this case there are two undefined symbols involved in the definition: BUFEND and BUFFER. Both of these are entered into SYMTAB with lists indicating the dependence of MAXLEN upon them. Similarly, the definition of PREVBT causes this symbol to be added to the list of dependencies on BUFFER [as shown in Fig. 2.21(d)].

So far we have simply been saving symbol definitions for later processing. The definition of BUFFER on line 4 lets us begin evaluation of some of these symbols. Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034. This address is stored as the value of BUFFER. The assembler then examines the list of symbols that are dependent on BUFFER. The symbol table entry for the first symbol in this list (MAXLEN) shows that it depends on two currently undefined symbols; therefore, MAXLEN cannot be evaluated immediately. Instead, the &2 is changed to &1 to show that only one symbol in the definition (BUFEND) remains undefined. The other symbol in the list (PREVBT) can be evaluated because it depends only on BUFFER. The value of the defining expression for PREVBT is calculated and stored in SYMTAB. The result is shown in Fig. 2.21(e).

The remainder of the processing follows the same pattern. When BUFEND is defined by line 5, its value is entered into the symbol table. The list associated with BUFEND then directs the assembler to evaluate MAXLEN, and entering a value for MAXLEN causes the evaluation of the symbol in its list (HALFSZ). As shown in Fig. 2.21(f), this completes the symbol definition process. If any symbols remained undefined at the end of the program, the assembler would flag them as errors.

The procedure we have just described applies to symbols defined by assembler directives like EQU. You are encouraged to think about how this method could be modified to allow forward references in ORG statements as well.

2.5 IMPLEMENTATION EXAMPLES

We discussed many of the most common assembler features in the preceding sections. However, the variety of machines and assembler languages is very great. Most assemblers have at least some unusual features that are related to machine architecture or language design. In this section we discuss three examples of assemblers for real machines. We are obviously unable to give a full description of any of these in the space available. Instead we focus on some of the most interesting or unusual features of each assembler. We are also particularly interested in areas where the assembler design differs from the basic algorithm and data structures described earlier.

The assembler examples we discuss are for the Pentium (x86), SPARC, and PowerPC architectures. You may want to review the descriptions of these architectures in Chapter 1 before proceeding.

2.5.1 MASM Assembler

This section describes some of the features of the Microsoft MASM assembler for Pentium and other x86 systems. Further information about MASM can be found in Barkakati (1992).

As we discussed in Section 1.4.2, the programmer of an x86 system views memory as a collection of segments. An MASM assembler language program is written as a collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST, and STACK.

During program execution, segments are addressed via the x86 segment registers. In most cases, code segments are addressed using register CS, and stack segments are addressed using register SS. These segment registers are automatically set by the system loader when a program is loaded for execution. Register CS is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is set to indicate the last stack segment processed by the loader.

Data segments (including constant segments) are normally addressed using DS, ES, FS, or GS. The segment register to be used can be specified explicitly by the programmer (by writing it as part of the assembler language instruction). If the programmer does not specify a segment register, one is selected by the assembler.

By default, the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME. For example, the directive

```
ASSUME  ES:DATASEG2
```

tells the assembler to assume that register ES indicates the segment DATASEG2. Thus, any references to labels that are defined in DATASEG2 will be assembled using register ES. It is also possible to collect several segments into a group and use ASSUME to associate a segment register with the group.

Registers DS, ES, FS and GS must be loaded by the program before they can be used to address data segments. For example, the instructions

```
MOV     AX,DATASEG2
MOV     ES,AX
```

would set ES to indicate the data segment DATASEG2. Notice the similarities between the ASSUME directive and the BASE directive we discussed for SIC/XE. The BASE directive tells a SIC/XE assembler the contents of register B; the programmer must provide executable instructions to load this value into the register. Likewise, ASSUME tells MASM the contents of a segment register; the programmer must provide instructions to load this register when the program is executed.

Jump instructions are assembled in two different ways, depending on whether the target of the jump is in the same code segment as the jump instruction. A *near jump* is a jump to a target in the same code segment; a *far jump* is a jump to a target in a different code segment. A near jump is assembled using the current code segment register CS. A far jump must be assembled using a different segment register, which is specified in an instruction prefix. The assembled machine instruction for a near jump occupies 2 or 3 bytes (depending upon whether the jump address is within 128 bytes of the current instruction). The assembled instruction for a far jump requires 5 bytes.

Forward references to labels in the source program can cause problems. For example, consider a jump instruction like

```
JMP     TARGET
```

If the definition of the label TARGET occurs in the program before the JMP instruction, the assembler can tell whether this is a near jump or a far jump. However, if this is a forward reference to TARGET, the assembler does not know how many bytes to reserve for the instruction.

By default, MASM assumes that a forward jump is a near jump. If the target of the jump is in another code segment, the programmer must warn the assembler by writing

```
JMP     FAR PTR TARGET
```

If the jump address is within 128 bytes of the current instruction, the programmer can specify the shorter (2-byte) near jump by writing

```
JMP     SHORT TARGET
```

If the JMP to TARGET is a far jump, and the programmer does not specify FAR PTR, a problem occurs. During Pass 1, the assembler reserves 3 bytes for the jump instruction. However, the actual assembled instruction requires 5 bytes. In the earlier versions of MASM, this caused an assembly error (called a phase error). In later versions of MASM, the assembler can repeat Pass 1 to generate the correct location counter values.

Notice the similarities between the far jump and the forward references in SIC/XE that require the use of extended format instructions.

There are also many other situations in which the length of an assembled instruction depends on the operands that are used. For example, the operands of an ADD instruction may be registers, memory locations, or immediate operands. Immediate operands may occupy from 1 to 4 bytes in the instruction. An operand that specifies a memory location may take varying amounts of space in the instruction, depending upon the location of the operand.

This means that Pass 1 of an x86 assembler must be considerably more complex than Pass 1 of a SIC assembler. The first pass of the x86 assembler must analyze the operands of an instruction, in addition to looking at the operation code. The operation code table must also be more complicated, since it must contain information on which addressing modes are valid for each operand.

Segments in an MASM source program can be written in more than one part. If a SEGMENT directive specifies the same name as a previously defined segment, it is considered to be a continuation of that segment. All of the parts of a segment are gathered together by the assembly process. Thus, segments can perform a similar function to the program blocks we discussed for SIC/XE.

References between segments that are assembled together are automatically handled by the assembler. External references between separately assembled modules must be handled by the linker. The MASM directive PUBLIC has approximately the same function as the SIC/XE directive EXTDEF. The MASM directive EXTRN has approximately the same function as EXTREF. We will consider the action of the linker in more detail in the next chapter.

The object program from MASM may be in several different formats, to allow easy and efficient execution of the program in a variety of operating environments. MASM can also produce an instruction timing listing that shows the number of clock cycles required to execute each machine instruction. This allows the programmer to exercise a great deal of control in optimizing timing-critical sections of code.

2.5.2 SPARC Assembler

This section describes some of the features of the SunOS SPARC assembler. Further information about this assembler can be found in Sun Microsystems (1994a).

A SPARC assembler language program is divided into units called *sections*. The assembler provides a set of predefined section names. Some examples of these are

.TEXT	Executable instructions
.DATA	Initialized read/write data
.RODATA	Read-only data
.BSS	Uninitialized data areas

It is also possible to define other sections, specifying section attributes such as "executable" and "writeable."

The programmer can switch between sections at any time in the source program by using assembler directives. The assembler maintains a separate location counter for each named section. Each time the assembler switches to a different section, it also switches to the location counter associated with that section. In this way, sections are similar to the program blocks we discussed for SIC. However, references between different sections are resolved by the linker, not by the assembler.

By default, symbols used in a source program are assumed to be local to that program. (However, a section may freely refer to local symbols defined in another section of the same program.) Symbols that are used in linking separately assembled programs may be declared to be either *global* or *weak*. A global symbol is either a symbol that is defined in the program and made accessible to others, or a symbol that is referenced in a program and defined externally. (Notice that this combines the functions of the EXTDEF and EXTREF directives we discussed for SIC.) A weak symbol is similar to a global symbol. However, the definition of a weak symbol may be overridden by a global symbol with the same name. Also, weak symbols may remain undefined when the program is linked, without causing an error.

The object file written by the SPARC assembler contains translated versions of the segments of the program and a list of relocation and linking operations that need to be performed. References between different segments of the same program are resolved when the program is linked. The object program also includes a symbol table that describes the symbols used during relocation and linking (global symbols, weak symbols, and section names).

SPARC assembler language has an unusual feature that is directly related to the machine architecture. As we discussed in Section 1.5.1, SPARC branch instructions (including subroutine calls) are *delayed branches*. The instruction immediately following a branch instruction is actually executed before the branch is taken. For example, in the instruction sequence


```
        CMP     %L0, 10
        BLE     LOOP
        ADD     %L2, %L3, %L4
```

the ADD instruction is executed *before* the conditional branch BLE. This ADD instruction is said to be in the *delay slot* of the branch; it is executed regardless of whether or not the conditional branch is taken.

To simplify debugging, SPARC assembly language programmers often place NOP (no-operation) instructions in delay slots when a program is written. The code is later rearranged to move useful instructions into the delay slots. For example, the instruction sequence illustrated above might originally have been

```
LOOP:   .
        .
        .
        ADD     %L2, %L3, %L4
        CMP     %L0, 10
        BLE     LOOP
        NOP
```

Moving the ADD instruction into the delay slot would produce the version discussed earlier. (Notice that the CMP instruction could not be moved into the delay slot, because it sets the condition codes that must be tested by the BLE.)

However, there is another possibility. Suppose that the original version of the loop had been

```
LOOP:   ADD     %L2, %L3, %L4
        .
        .
        CMP     %L0, 10
        BLE     LOOP
        NOP
```

Now the ADD instruction is logically the *first* instruction in the loop. It could still be moved into the delay slot, as previously described. However, this would create a problem. On the last execution of the loop, the ADD instruction (which is the beginning of the next loop iteration) should not be executed.

The SPARC architecture defines a solution to this problem. A conditional branch instruction like BLE can be *annulled*. If a branch is annulled, the instruction in its delay slot is executed if the branch is taken, but *not* executed if the branch is not taken. Annulled branches are indicated in SPARC assembler

language by writing ",A" following the operation code. Thus the loop just discussed could be rewritten as

```
LOOP: .
      .
      .
      CMP     %L0, 10
      BLE,A   LOOP
      ADD     %L2, %L3, %L4
```

The SPARC assembler provides warning messages to alert the programmer to possible problems with delay slots. For example, a label on an instruction in a delay slot usually indicates an error. A segment that ends with a branch instruction (with nothing in the delay slot) is also likely to be incorrect. Before the branch is executed, the machine will attempt to execute whatever happens to be stored at the memory location immediately following the branch.

2.5.3 AIX Assembler

This section describes some of the features of the AIX assembler for PowerPC and other similar systems. Further information about this assembler can be found in IBM (1994b).

The AIX assembler includes support for various models of PowerPC microprocessors, as well as earlier machines that implement the original POWER architecture. The programmer can declare which architecture is being used with the assembler directive .MACHINE. The assembler automatically checks for POWER or PowerPC instructions that are not valid for the specified environment. When the object program is generated, the assembler includes a flag that indicates which processors are capable of running the program. This flag depends on which instructions are actually used in the program, not on the .MACHINE directive. For example, a PowerPC program that contains only instructions that are also in the original POWER architecture would be executable on either type of system.

As we discussed in Section 1.5.2, PowerPC load and store instructions use a base register and a displacement value to specify an address in memory. Any of the general-purpose registers (except GPR0) can be used as a base register. Decisions about which registers to use in this way are left to the programmer. In a long program, it is not unusual to have several different base registers in use at the same time. The programmer specifies which registers are available for use as base registers, and the contents of these registers, with the .USING

assembler directive. This is similar in function to the BASE statement in our SIC/XE assembler language. Thus the statements

```
.USING    LENGTH,1
.USING    BUFFER,4
```

would identify GPR1 and GPR4 as base registers. GPR1 would be assumed to contain the address of LENGTH, and GPR4 would be assumed to contain the address of BUFFER. As with SIC/XE, the programmer must provide instructions to place these values into the registers at execution time. Additional .USING statements may appear at any point in the program. If a base register is to be used later for some other purpose, the programmer indicates with the .DROP statement that this register is no longer available for addressing purposes.

This additional flexibility in register usage means more work for the assembler. A *base register table* is used to remember which of the general-purpose registers are currently available as base registers, and what base addresses they contain. Processing a .USING statement causes an entry to be made in this table (or an existing entry to be modified); processing a .DROP statement removes the corresponding table entry. For each instruction whose operand is an address in memory, the assembler scans the table to find a base register that can be used to address that operand. If more than one register can be used, the assembler selects the base register that results in the smallest signed displacement. If no suitable base register is available, the instruction cannot be assembled. The process of displacement calculation is the same as we described for SIC/XE.

The AIX assembler language also allows the programmer to write base registers and displacements explicitly in the source program. For example, the instruction

```
L    2,8(4)
```

specifies an operand address that is 8 bytes past the address contained in GPR4. This form of addressing may be useful when some register is known to contain the starting address of a table or data record, and the programmer wishes to refer to a fixed location within that table or record. The assembler simply inserts the specified values into the object code instruction: in this case base register GPR4 and displacement 8. The base register table is not involved, and the register used in this way need not have appeared in a .USING statement.

An AIX assembler language program can be divided into *control sections* using the .CSECT assembler directive. Each control section has an associated storage mapping class that describes the kind of data it contains. Some of the most commonly used storage mapping classes are PR (executable instructions), RO (read-only data), RW (read/write data), and BS (uninitialized read/write data). AIX control sections combine some of the features of the SIC control sections and program blocks that we discussed in Section 2.3. One control section may consist of several different parts of the source program. These parts are gathered together by the assembler, as with SIC program blocks. The control sections themselves remain separate after assembly, and are handled independently by the loader or linkage editor.

The AIX assembler language provides a special type of control section called a *dummy section*. Data items included in a dummy section do not actually become part of the object program; they serve only to define labels within the section. Dummy sections are most commonly used to describe the layout of a record or table that is defined externally. The labels define symbols that can be used to address fields in the record or table (after an appropriate base register is established). AIX also provides *common blocks*, which are uninitialized blocks of storage that can be shared between independently assembled programs.

Linking of control sections can be accomplished using methods like the ones we discussed for SIC. The assembler directive .GLOBL makes a symbol available to the linker, and the directive .EXTERN declares that a symbol is defined in another source module. These directives are essentially the same as the SIC directives EXTDEF and EXTREF. Expressions that involve relocatable and external symbols are classified and handled using rules similar to those discussed in Sections 2.3.3 and 2.3.5.

The AIX assembler also provides a different method for linking control sections. By using assembler directives, the programmer can create a *table of contents* (TOC) for the assembled program. The TOC contains addresses of control sections and global symbols defined within the control sections. To refer to one of these symbols, the program retrieves the needed address from the TOC, and then uses that address to refer to the needed data item or procedure. (Some types of frequently used data items can be stored directly in the TOC for efficiency of retrieval.) If all references to external symbols are done in this way, then the TOC entries are the only parts of the program involved in relocation and linking when the program is loaded.

The AIX assembler itself has a two-pass structure similar to the one we discussed for SIC. However, there are some significant differences. The first pass of the AIX assembler writes a listing file that contains warnings and error messages. If errors are found during the first pass, the assembler terminates and

does not continue to the second pass. In this case, the assembly listing contains only errors that could be detected during Pass 1.

If no errors are detected during the first pass, the assembler proceeds to Pass 2. The second pass reads the source program again, instead of using an intermediate file as we discussed for SIC. This means that location counter values must be recalculated during Pass 2. It also means that any warning messages that were generated during Pass 1 (but were not serious enough to terminate the assembly) are lost. The assembly listing will contain only errors and warnings that are generated during Pass 2.

Assembled control sections are placed into the object program according to their storage mapping class. Executable instructions, read-only data, and various kinds of debugging tables are assigned to an object program section named .TEXT. Read/write data and TOC entries are assigned to an object program section named .DATA. Uninitialized data is assigned to a section named .BSS. When the object program is generated, the assembler first writes all of the .TEXT control sections, followed by all of the .DATA control sections except for the TOC. The TOC is written after the other .DATA control sections. Relocation and linking operations are specified by entries in a relocation table, similar to the Modification records we discussed for SIC.

EXERCISES

Section 2.1

1. Apply the algorithm described in Fig. 2.4 to assemble the source program in Fig. 2.1. Your results should be the same as those shown in Figs. 2.2 and 2.3.

2. Apply the algorithm described in Fig. 2.4 to assemble the following SIC source program:

```
SUM     START   4000
FIRST   LDX     ZERO
        LDA     ZERO
LOOP    ADD     TABLE,X
        TIX     COUNT
        JLT     LOOP
        STA     TOTAL
        RSUB
TABLE   RESW    2000
COUNT   RESW    1
ZERO    WORD    0
TOTAL   RESW    1
        END     FIRST
```

3. As mentioned in the text, a number of operations in the algorithm of Fig. 2.4 are not explicitly spelled out. (One example would be scanning the instruction operand field for the modifier ",X".) List as many of these implied operations as you can, and think about how they might be implemented.

4. Suppose that you are to write a "disassembler"—that is, a system program that takes an ordinary object program as input and produces a listing of the source version of the program. What tables and data structures would be required, and how would they be used? How many passes would be needed? What problems would arise in recreating the source program?

5. Many assemblers use free-format input. Labels must start in Column 1 of the source statement, but other fields (opcode, operands, comments) may begin in any column. The various fields are separated by blanks. How could our assembler logic be modified to allow this?

6. The algorithm in Fig. 2.4 provides for the detection of some assembly errors; however, there are many more such errors that might occur. List error conditions that might arise during the assembly of a SIC program. When and how would each type of error be detected, and what action should the assembler take for each?

7. Suppose that the SIC assembler language is changed to include a new form of the RESB statement, such as

```
        RESB   n'c'
```

which reserves *n* bytes of memory and initializes all of these bytes to the character 'c'. For example, line 105 in Fig. 2.5 could be changed to

```
BUFFER     RESB   4096' '
```

This feature could be implemented by simply generating the required number of bytes in Text records. However, this could lead to a large increase in the size of the object program—for example, the object program in Fig. 2.8 would be about 40 times its previous size. Propose a way to implement this new form of RESB without such a large increase in object program size.

8. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. In the case of a duplicate symbol, this assembler would give an error message only for the second (i.e., duplicate) definition. For example, it would give an error message only for line 5 of the program below.

```
1        P3        START     1000
2                  LDA       ALPHA
.                  .         .
3                  STA       ALPHA
.                  .         .
4        ALPHA     RESW      1
.                  .         .
5        ALPHA     WORD      0
6                  END
```

Suppose that you want to change the assembler to give error messages for all definitions of a doubly defined symbol (e.g., lines 4 and 5), and also for all references to a doubly defined symbol (e.g., lines 2 and 3). Describe the changes you would make to accomplish this. In making this modification, you should change the existing assembler as little as possible.

9. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. You want to change this assembler so that it gives a warning message for labels that are not referenced in the program, as illustrated by the following example.

```
P3       START     1000
         LDA       DELTA
         ADD       BETA
LOOP     STA       DELTA
Warning: label is never referenced
         RSUB
ALPHA    RESW      1
Warning: label is never referenced
BETA     RESW      1
DELTA    RESW      1
         END
```

The warning messages should appear in the assembly listing directly below the line that contains the unreferenced label, as shown above. Describe the changes you would make in the assembler to add this

new diagnostic feature. In making this modification, you should change the existing assembler as little as possible.

Section 2.2

1. Could the assembler decide for itself which instructions need to be assembled using extended format? (This would avoid the necessity for the programmer to code + in such instructions.)

2. As we have described it, the BASE statement simply gives information to the assembler. The programmer must also write an instruction like LDB to load the correct value into the base register. Could the assembler automatically generate the LDB instruction from the BASE statement? If so, what would be the advantages and disadvantages of doing this?

3. Generate the object code for each statement in the following SIC/XE program:

```
SUM      START   0
FIRST    LDX     #0
         LDA     #0
        +LDB     #TABLE2
         BASE    TABLE2
LOOP     ADD     TABLE,X
         ADD     TABLE2,X
         TIX     COUNT
         JLT     LOOP
        +STA     TOTAL
         RSUB
COUNT    RESW    1
TABLE    RESW    2000
TABLE2   RESW    2000
TOTAL    RESW    1
         END     FIRST
```

4. Generate the complete object program for the source program given in Exercise 3.

5. Modify the algorithm described in Fig. 2.4 to handle all of the SIC/XE addressing modes discussed. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

6. Modify the algorithm described in Fig. 2.4 to handle relocatable programs. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

7. Suppose that you are writing a disassembler for SIC/XE (see Exercise 2.1.4.) How would your disassembler deal with the various addressing modes and instruction formats?

8. Our discussion of SIC/XE Format 4 instructions specified that the 20-bit "address" field should contain the actual target address, and that addressing mode bits *b* and *p* should be set to 0. (That is, the instruction should contain a direct address—it should not use base relative or program-counter relative addressing.)

   However, it would be possible to use program-counter relative addressing with Format 4. In that case, the "address" field would actually contain a displacement, and bit *p* would be set to 1. For example, the instruction on line 15 in Fig. 2.6 could be assembled as

```
0006  CLOOP  +JSUB  RDREC      4B30102C
```

   (using program-counter relative addressing with displacement 102C).

   What would be the advantages (if any) of assembling Format 4 instructions in this way? What would be the disadvantages (if any)? Are there any situations in which it would *not* be possible to assemble a Format 4 instruction using program-counter relative addressing?

9. Our Modification record format is well suited for SIC/XE programs because all address fields in instructions and data words fall neatly into half-bytes. What sort of Modification record could we use if this were not the case (that is, if address fields could begin anywhere within a byte and could be of any length)?

10. Suppose that we made the program in Fig. 2.1 a relocatable program. This program is written for the *standard* version of SIC, so all operand addresses are actual addresses, and there is only one instruction format. Nearly every instruction in the object program would need to have its operand address modified at load time. This would mean a large number of Modification records (more than doubling the size of the object program). How could we include the required relocation information without this large increase in object program size?

11. Suppose that you are writing an assembler for a machine that has *only* program-counter relative addressing. (That is, there are no direct-addressing instruction formats and no base relative addressing.) Suppose that you wish to assemble an instruction whose operand is an absolute address in memory—for example,

```
        LDA  100
```

to load register A from address (hexadecimal) 100 in memory. How might such an instruction be assembled in a relocatable program? What relocation operations would be required?

12. Suppose that you are writing an assembler for a machine on which the length of an assembled instruction depends upon the type of the operand. Consider, for example, the following three fragments of code:

```
a.        ADD  ALPHA
          .
          .
ALPHA     DC   I(3)

b.        ADD  ALPHA
          .
          .
ALPHA     DC   F(3.1)

c.        ADD  ALPHA
          .
          .
ALPHA     DC   D(3.14159)
```

In case (a), ALPHA is an integer operand; the ADD instruction generates 2 bytes of object code. In case (b), ALPHA is a single-precision floating-point operand; the ADD instruction generates 3 bytes of object code. In case (c), ALPHA is a double-precision floating-point operand; the ADD instruction generates 4 bytes of object code.

What special problems does such a machine present for an assembler? Briefly describe how you would solve these problems—that is, how your assembler for this machine would be different from the assembler structure described in Section 2.1.

Section 2.3

1. Modify the algorithm described in Fig. 2.4 to handle literals.
2. In the program of Fig. 2.9, could we have used literals on lines 135 and 145? Why might we prefer *not* to use a literal here?
3. With a minor extension to our literal notation, we could write the instruction on line 55 of Fig. 2.9 as

```
        LDA   =W'3'
```

specifying as the literal operand a word with the value 3. Would this be a good idea?

4. Immediate operands and literals are both ways of specifying an operand value in a source statement. What are the advantages and disadvantages of each? When might each be preferable to the other?
5. Suppose that you have a two-pass SIC/XE assembler that does not support literals. Now you want to modify the assembler to handle literals. However, you want to place the literal pool at the *beginning* of the assembled program, not at the end as is commonly done. (You do not have to worry about LTORG statements—your assembler should always place all literals in a pool at the beginning of the program.) Describe how you could accomplish this. If possible, you should do so without adding another pass to the assembler. Be sure to describe any data structures that you may need, and explain how they are used in the assembler.
6. Suppose we made the following changes to the program in Fig. 2.9:
   a. Delete the LTORG statement on line 93.
   b. Change the statement on line 45 to +LDA... .
   c. Change the operands on lines 135 and 145 to use literals (and delete line 185).

   Show the resulting object code for lines 45, 135, 145, 215, and 230. Also show the literal pool with addresses and data values. Note: you do not need to retranslate the entire program to do this.
7. Assume that the symbols ALPHA and BETA are labels in a source program. What is the difference between the following two sequences of statements?

```
a.          LDA   ALPHA-BETA
b.          LDA   ALPHA
            SUB   BETA
```

8. What is the difference between the following sequences of statements?

```
a.          LDA   #3
b. THREE    EQU    3
            .
            .
            LDA   #THREE
c. THREE    EQU    3
            .
            .
            LDA    THREE
```

9. Modify the algorithm described in Fig. 2.4 to handle multiple program blocks.

10. Modify the algorithm described in Fig. 2.4 to handle multiple control sections.

11. Suppose all the features we described in Section 2.3 were to be implemented in an assembler. How would the symbol table required be different from the one discussed in Section 2.1?

12. Which of the features described in Section 2.3 would create additional problems in the writing of a disassembler (see Exercise 2.1.4)? Describe these problems, and discuss possible solutions.

13. When different control sections are assembled together, some references between them could be handled by the assembler (instead of being passed on to the loader). In the program of Fig. 2.15, for example, the expression on line 190 could be evaluated directly by the assembler because its symbol table contains all of the required information. What would be the advantages and disadvantages of doing this?

14. In the program of Fig. 2.11, suppose we used only two program blocks: the default block and CBLKS. Assume that the data items in CDATA are to be included in the default block. What changes in the source program would accomplish this? Show the object program (corresponding to Fig. 2.13) that would result.

15. Suppose that for some reason it is desirable to separate the parts of an assembler language program that require initialization (e.g., instructions and data items defined with WORD or BYTE) from the parts that do not require initialization (e.g., storage reserved with RESW or RESB). Thus, when the program is loaded for execution it should look like

Instructions and initialized data items
Reserved storage (uninitialized data items)

Suppose that it is considered too restrictive to require the programmer to perform this separation. Instead, the assembler should take the source program statements in whatever order they are written, and automatically perform the rearrangement as described above.

Describe a way in which this separation of the program could be accomplished by a two-pass assembler.

16. Suppose LENGTH is defined as in the program of Fig. 2.9. What would be the difference between the following sequences of statements?

```
a.          LDA    LENGTH
            SUB    #1

b.          LDA    LENGTH-1
```

17. Referring to the definitions of symbols in Fig. 2.10, give the value, type, and intuitive meaning (if any) of each of the following expressions:

a. BUFFER-FIRST

b. BUFFER+4095

c. MAXLEN-1

d. BUFFER+MAXLEN-1

e. BUFFER-MAXLEN

f. 2*LENGTH

g. `2*MAXLEN-1`

h. `MAXLEN-BUFFER`

i. `FIRST+BUFFER`

j. `FIRST-BUFFER+BUFEND`

18. In the program of Fig. 2.9, what is the advantage of writing (on line 107)

```
MAXLEN   EQU    BUFEND-BUFFER
```

instead of

```
MAXLEN   EQU    4096  ?
```

19. In the program of Fig. 2.15, could we change line 190 to

```
MAXLEN   EQU    BUFEND-BUFFER
```

and line 133 to

```
         +LDT   #MAXLEN
```

as we did in Fig. 2.9?

20. The assembler could simply assume that any reference to a symbol not defined within a control section is an external reference. This change would eliminate the need for the EXTREF statement. Would this be a good idea?

21. How could an assembler that allows external references avoid the need for an EXTDEF statement? What would be the advantages and disadvantages of doing this?

22. The assembler could automatically use extended format for instructions whose operands involve external references. This would eliminate the need for the programmer to code + in such statements. What would be the advantages and disadvantages of doing this?

23. On some systems, control sections can be composed of several different parts, just as program blocks can. What problems does this pose for the assembler? How might these problems be solved?

24. Assume that the symbols RDREC and COPY are defined as in Fig. 2.15. According to our rules, the expression

```
RDREC-COPY
```

would be illegal (that is, the assembler and/or the loader would reject it). Suppose that for some reason the program really needs the value of this expression. How could such a thing be accomplished without changing the rules for expressions?

25. We discussed a large number of assembler directives, and many more could be implemented in an actual assembler. Checking for them one at a time using comparisons might be quite inefficient. How could we use a table, perhaps similar to OPTAB, to speed recognition and handling of assembler directives? (Hint: the answer to this problem may depend upon the language in which the assembler itself is written.)

26. Other than the listing of the source program with generated object code, what assembler outputs might be useful to the programmer? Suggest some optional listings that might be generated and discuss any data structures or algorithms involved in producing them.

Section 2.4

1. The process of fixing up a few forward references should involve less overhead than making a complete second pass of the source program. Why don't all assemblers use the one-pass technique for efficiency?

2. Suppose we wanted our assembler to produce a cross-reference listing for all symbols used in the program. For the program of Fig. 2.5, such a listing might look like

Symbol	Defined on line	Used on lines
COPY	5	
FIRST	10	255
CLOOP	15	40
ENDFIL	45	30
EOF	80	45
RETADR	95	10,70
LENGTH	100	12,13,20,60,175,212
.		
.		

How might this be done by the assembler? Indicate changes to the logic and tables discussed in Section 2.1 that would be required.

3. Could a one-pass assembler produce a relocatable object program and handle external references? Describe the processing logic that would be involved and identify any potential difficulties.
4. How could literals be implemented in a one-pass assembler?
5. We discussed one-pass assemblers as though instruction operands could only be single symbols. How could a one-pass assembler handle an instruction like

```
JEQ     ENDFIL+3
```

where ENDFIL has not yet been defined?

6. Outline the logic flow for a simple one-pass load-and-go assembler.
7. Using the methods outlined in Chapter 8, develop a modular design for a one-pass assembler that produces object code in memory.
8. Suppose that an instruction involving a forward reference is to be assembled using program-counter relative addressing. How might this be handled by a one-pass assembler?
9. The process of fixing up forward references in a one-pass assembler that produces an object program is very similar to the linking process described in Section 2.3.5. Why didn't we just use Modification records to fix up the forward references?
10. How could we extend the methods of Section 2.4.2 to handle forward references in ORG statements?

Section 2.5

1. Consider the description of the VAX architecture in Section 1.4.1. What characteristics would you expect to find in a VAX assembler?
2. Consider the description of the T3E architecture in Section 1.5.3. What characteristics would you expect to find in a T3E assembler?

Chapter 3

Loaders and Linkers

As we have seen, an object program contains translated instructions and data values from the source program, and specifies addresses in memory where these items are to be loaded. Our discussions in Chapter 2 introduced the following three processes:

1. *Loading*, which brings the object program into memory for execution.
2. *Relocation*, which modifies the object program so that it can be loaded at an address different from the location originally specified (see Section 2.2.2).
3. *Linking*, which combines two or more separate object programs and supplies the information needed to allow references between them (see Section 2.3.5).

A *loader* is a system program that performs the loading function. Many loaders also support relocation and linking. Some systems have a *linker* (or *linkage editor*) to perform the linking operations and a separate loader to handle relocation and loading. In most cases all the program translators (i.e., assemblers and compilers) on a particular system produce object programs in the same format. Thus one system loader or linker can be used regardless of the original source programming language.

In this chapter we study the design and implementation of loaders and linkers. For simplicity we often use the term *loader* in place of *loader and/or linker*. Because the processes of assembly and loading are closely related, this chapter is similar in structure to the preceding one. Many of the same examples used in our study of assemblers are carried forward in this chapter. During our discussion of assemblers, we studied a number of features and capabilities that are of concern to both the assembler and the loader. In the present chapter we encounter many of the same concepts again. This time, of course, we are primarily concerned with the operation of the loader; however, it is important to remember the close connections between program translation and loading.

As in the preceding chapter, we begin by discussing the most basic software function—in this case, loading an object program into memory for execution. Section 3.1 presents the design of an *absolute loader* and illustrates its operation. Such a loader might be found on a simple SIC machine that uses the sort of assembler described in Section 2.1.

Section 3.2 examines the issues of relocation and linking from the loader's point of view. We consider some possible alternatives for object program representation and examine how these are related to issues of machine architecture. We also present the design of a *linking loader*, a more advanced type of loader that is typical of those found on most modern computing systems.

Section 3.3 presents a selection of commonly encountered loader features that are not directly related to machine architecture. As before, our purpose is not to cover all possible options, but to introduce some of the concepts and techniques most frequently found in loaders.

Section 3.4 discusses alternative ways of accomplishing loader functions. We consider the various times at which relocation and linking can be performed, and the advantages and disadvantages associated with each. In this context we study linkage editors (which perform linking before loading) and dynamic linking schemes (which delay linking until execution time).

Finally, in Section 3.5 we briefly discuss some examples of actual loaders and linkers. As before, we are primarily concerned with aspects of each piece of software that are related to hardware or software design decisions.

3.1 BASIC LOADER FUNCTIONS

In this section we discuss the most fundamental functions of a loader—bringing an object program into memory and starting its execution. You are probably already familiar with how these basic functions are performed. This section is intended as a review to set the stage for our later discussion of more advanced loader functions. Section 3.1.1 discusses the functions and design of an absolute loader and gives the outline of an algorithm for such a loader. Section 3.1.2 presents an example of a very simple absolute loader for SIC/XE, to clarify the coding techniques that are involved.

3.1.1 Design of an Absolute Loader

We consider the design of an absolute loader that might be used with the sort of assembler described in Section 2.1. The object program format used is the same as that described in Section 2.1.1. An example of such an object program is shown in Fig. 3.1(a).


```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

Memory address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D302[illegible]	3FD8205D	28103030	30575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

← COPY

(b) Program loaded in memory

Figure 3.1 Loading of an absolute program.

Because our loader does not need to perform such functions as linking and program relocation, its operation is very simple. All functions are accomplished in a single pass. The Header record is checked to verify that the correct program has been presented for loading (and that it will fit into the available memory). As each Text record is read, the object code it contains is moved to the indicated address in memory. When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program. Figure 3.1(b) shows a representation of the program from Fig. 3.1(a) after loading. The contents of memory locations for which there is no Text record are shown as *xxxx*. This indicates that the previous contents of these locations remain unchanged.

Figure 3.2 shows an algorithm for the absolute loader we have discussed. Although this process is extremely simple, there is one aspect that deserves comment. In our object program, each byte of assembled code is given using its hexadecimal representation in character form. For example, the machine operation code for an STL instruction would be represented by the *pair of characters* "1" and "4". When these are read by the loader (as part of the object program), they will occupy *two* bytes of memory. In the instruction as loaded for execution, however, this operation code must be stored in a *single* byte with *hexadecimal value* 14. Thus each pair of bytes from the object program record must be packed together into one byte during loading. It is very important to realize that in Fig. 3.1(a), each printed character represents one *byte* of the object program record. In Fig. 3.1(b), on the other hand, each printed character represents one *hexadecimal digit* in memory (i.e., a half-byte).

This method of representing an object program is inefficient in terms of both space and execution time. Therefore, most machines store object programs in a *binary* form, with each byte of object code stored as a single byte in the object program. In this type of representation, of course, a byte may contain any binary value. We must be sure that our file and device conventions do not cause some of the object program bytes to be interpreted as control characters. For example, the convention described in Section 2.1—indicating the end of a record with a byte containing hexadecimal 00—would clearly be unsuitable for use with a binary object program.

Obviously object programs stored in binary form do not lend themselves well to printing or to reading by human beings. Therefore, we continue to use character representations of object programs in our examples in this book.

```
begin
   read Header record
   verify program name and length
   read first Text record
   while record type ≠ 'E' do
     begin
        {if object code is in character form, convert into
           internal representation}
        move object code to specified location in memory
        read next object program record
     end
   jump to address specified in End record
end
```

Figure 3.2 Algorithm for an absolute loader.

3.1.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called a *bootstrap loader*, is executed. This bootstrap loads the first program to be run by the computer—usually an operating system. (Bootstrap loaders are discussed in more detail in Section 3.4.3.) In this section, we examine a very simple bootstrap loader for SIC/XE. In spite of its simplicity, this program illustrates almost all of the logic and coding techniques that are used in an absolute loader.

Figure 3.3 shows the source code for our bootstrap loader. The bootstrap itself begins at address 0 in the memory of the machine. It loads the operating system (or some other program) starting at address 80. Because this loader is used in a unique situation (the initial program load for the system), the program to be loaded can be represented in a very simple format. Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits (just as it is in a Text record of a SIC object program). However, there is no Header record, End record, or control information (such as addresses or lengths). The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

Much of the work of the bootstrap loader is performed by the subroutine GETC. This subroutine reads one character from device F1 and converts it from the ASCII character code to the value of the hexadecimal digit that is represented by that character. For example, the ASCII code for the character "0" (hexadecimal 30) is converted to the numeric value 0. Likewise, the ASCII codes for "1" through "9" (hexadecimal 31 through 39) are converted to the numeric values 1 through 9, and the codes for "A" through "F" (hexadecimal 41 through 46) are converted to the values 10 through 15. This is accomplished by subtracting 48 (hexadecimal 30) from the character codes for "0" through "9", and subtracting 55 (hexadecimal 37) from the codes for "A" through "F". The subroutine GETC jumps to address 80 when an end-of-file (hexadecimal 04) is read from device F1. It skips all other input characters that have ASCII codes less than hexadecimal 30. This causes the bootstrap to ignore any control bytes (such as end-of-line) that are read.

The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X. GETC is used to read and convert a pair of characters from device F1 (representing 1 byte of object code to be loaded). These two hexadecimal digit values are combined into a single byte by shifting the first one left 4 bit positions and adding the second to it. The resulting byte is stored at the address currently in register X, using a STCH instruction that refers to location 0 using indexed addressing. The TIXR instruction is then used to add 1 to the value in register X. (Because we are not interested in the result of the comparison performed by TIXR, register X is also used as the second operand for this instruction.)

```
BOOT     START    0        BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
         CLEAR    A        CLEAR REGISTER A TO ZERO
         LDX      #128     INITIALIZE REGISTER X TO HEX 80
LOOP     JSUB     GETC     READ HEX DIGIT FROM PROGRAM BEING LOADED
         RMO      A,S      SAVE IN REGISTER S
         SHIFTL   S,4      MOVE TO HIGH-ORDER 4 BITS OF BYTE
         JSUB     GETC     GET NEXT HEX DIGIT
         ADDR     S,A      COMBINE DIGITS TO FORM ONE BYTE
         STCH     0,X      STORE AT ADDRESS IN REGISTER X
         TIXR     X,X      ADD 1 TO MEMORY ADDRESS BEING LOADED
         J        LOOP     LOOP UNTIL END OF INPUT IS REACHED
.
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC     TD       INPUT    TEST INPUT DEVICE
         JEQ      GETC     LOOP UNTIL READY
         RD       INPUT    READ CHARACTER
         COMP     #4       IF CHARACTER IS HEX 04 (END OF FILE),
         JEQ      80           JUMP TO START OF PROGRAM JUST LOADED
         COMP     #48      COMPARE TO HEX 30 (CHARACTER '0')
         JLT      GETC     SKIP CHARACTERS LESS THAN '0'
         SUB      #48      SUBTRACT HEX 30 FROM ASCII CODE
         COMP     #10      IF RESULT IS LESS THAN 10, CONVERSION IS
         JLT      RETURN       COMPLETE. OTHERWISE, SUBTRACT 7 MORE
         SUB      #7           (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN   RSUB             RETURN TO CALLER
INPUT    BYTE     X'F1'    CODE FOR INPUT DEVICE
         END      LOOP
```

Figure 3.3 Bootstrap loader for SIC/XE.

You should work through the execution of this bootstrap routine by hand with several bytes of sample input, keeping track of the exact contents of all registers and memory locations as you go. This will help you become familiar with the machine-level details of how loading is performed.

For simplicity, the bootstrap routine in Fig. 3.3 does not do any error checking it assumes that its input is correct. You are encouraged to think about the

different kinds of error conditions that might arise during the loading, and how these could be handled.

3.2 MACHINE-DEPENDENT LOADER FEATURES

The absolute loader described in Section 3.1 is certainly simple and efficient; however, this scheme has several potential disadvantages. One of the most obvious is the need for the programmer to specify (when the program is assembled) the actual address at which it will be loaded into memory. If we are considering a very simple computer with a small memory (such as the standard version of SIC), this does not create much difficulty. There is only room to run one program at a time, and the starting address for this single user program is known in advance. On a larger and more advanced machine (such as SIC/XE), the situation is not quite as easy. We would often like to run several independent programs together, sharing memory (and other system resources) between them. This means that we do not know in advance where a program will be loaded. Efficient sharing of the machine requires that we write relocatable programs instead of absolute ones.

Writing absolute programs also makes it difficult to use subroutine libraries efficiently. Most such libraries (for example, scientific or mathematical packages) contain many more subroutines than will be used by any one program. To make efficient use of memory, it is important to be able to select and load exactly those routines that are needed. This could not be done effectively if all of the subroutines had preassigned absolute addresses.

In this section we consider the design and implementation of a more complex loader. The loader we present is one that is suitable for use on a SIC/XE system and is typical of those that are found on most modern computers. This loader provides for program relocation and linking, as well as for the simple loading functions described in the preceding section. As part of our discussion, we examine the effect of machine architecture on the design of the loader.

The need for program relocation is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics. Section 3.2.1 discusses these dependencies by examining different implementation techniques and the circumstances in which they might be used.

Section 3.2.2 examines program linking from the loader's point of view. Linking is not a machine-dependent function in the sense that relocation is; however, the same implementation techniques are often used for these two functions. In addition, the process of linking usually involves relocation of some of the routines being linked together. (See, for example, the previous discussion concerning the use of subroutine libraries.) For these reasons we discuss linking together with relocation in this section.

Section 3.2.3 discusses the data structures used by a typical linking (and relocating) loader, and gives a description of the processing logic involved. The algorithm presented here serves as a starting point for discussion of some of the more advanced loader features in the following sections.

3.2.1 Relocation

Loaders that allow for program relocation are called *relocating loaders* or *relative loaders*. The concept of program relocation was introduced in Section 2.2.2; you may want to briefly review that discussion before reading further. In this section we discuss two methods for specifying relocation as part of the object program.

The first method we discuss is essentially the same as that introduced in Chapter 2. A Modification record is used to describe each part of the object code that must be changed when the program is relocated. (The format of the Modification record is given in Section 2.3.5.) Figure 3.4 shows a SIC/XE program we use to illustrate this first method of specifying relocation. The program is the same as the one in Fig. 2.6; it is reproduced here for convenience. Most of the instructions in this program use relative or immediate addressing. The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

Figure 3.5 displays the object program corresponding to the source in Fig.3.4. Notice that there is one Modification record for each value that must be changed during relocation (in this case, the three instructions previously mentioned). Each Modification record specifies the starting address and length of the field whose value is to be altered. It then describes the modification to be performed. In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program. The algorithm the loader uses to perform these modifications is discussed in Section 3.2.3. More examples of relocation specified in this manner appear in the next section when we examine the relationship between relocation and linking.

The Modification record scheme is a convenient means for specifying program relocation; however, it is not well suited for use with all machine architectures. Consider, for example, the program in Fig. 3.6. This is a relocatable program written for the standard version of SIC. The important difference between this example and the one in Fig. 3.4 is that the standard SIC machine does not use relative addressing. In this program the addresses in all the instructions except RSUB must be modified when the program is relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig. 3.5.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 3.4 Example of a SIC/XE program (from Fig. 2.6).

```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05^+COPY
M^000014^05^+COPY
M^000027^05^+COPY
E^000000
```

Figure 3.5 Object program with relocation by Modification records.

On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique. Figure 3.7 shows this method applied to our SIC program example. There are no Modification records. The Text records are the same as before except that there is a *relocation bit* associated with each word of object code. Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction. The relocation bits are gathered together into a *bit mask* following the length indicator in each Text record. In Fig. 3.7 this mask is represented (in character form) as three hexadecimal digits. These characters are underlined for easier identification in the figure.

If the relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the program is relocated. A bit value of 0 indicates that no modification is necessary. If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0. Thus the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation. These words contain the instructions corresponding to lines 10 through 55 in Fig. 3.6. The mask E00 in the second Text record specifies that the first three words are to be modified. The remainder of the object code in this record represents data constants (and the RSUB instruction) and thus does not require modification.

The other Text records follow the same pattern. Note that the object code generated from the LDX instruction on line 210 begins a new Text record even though there is room for it in the preceding record. This occurs because each relocation bit is associated with a 3-byte segment of object code in the Text record. Any value that is to be modified during relocation must coincide with one of these 3-byte segments so that it corresponds to a relocation bit. The assembled LDX instruction does require modification because of the direct address. However, if it were placed in the preceding Text record, it would not be

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061
70	0024		LDL	RETADR	080033
75	0027		RSUB		4C0000
80	002A	EOF	BYTE	C'EOF'	454F46
85	002D	THREE	WORD	3	000003
90	0030	ZERO	WORD	0	000000
95	0033	RETADR	RESW	1	
100	0036	LENGTH	RESW	1	
105	0039	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1039	RDREC	LDX	ZERO	040030
130	103C		LDA	ZERO	000030
135	103F	RLOOP	TD	INPUT	E0105D
140	1042		JEQ	RLOOP	30103F
145	1045		RD	INPUT	D8105D
150	1048		COMP	ZERO	280030
155	104B		JEQ	EXIT	301057
160	104E		STCH	BUFFER,X	548039
165	1051		TIX	MAXLEN	2C105E
170	1054		JLT	RLOOP	38103F
175	1057	EXIT	STX	LENGTH	100036
180	105A		RSUB		4C0000
185	105D	INPUT	BYTE	X'F1'	F1
190	105E	MAXLEN	WORD	4096	001000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	1061	WRREC	LDX	ZERO	040030
215	1064	WLOOP	TD	OUTPUT	E01079
220	1067		JEQ	WLOOP	301064
225	106A		LDCH	BUFFER,X	508039
230	106D		WD	OUTPUT	DC1079
235	1070		TIX	LENGTH	2C0036
240	1073		JLT	LOOP	381064
245	1076		RSUB		4C0000
250	1079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 3.6 Relocatable program for a standard SIC machine.

```
H^COPY  ^00000^00107A
T^000000^1E^FFC^14003^348103^9000036^280030^300015^481061^3C0003^00002A^0C0039^00002D
T^00001E^15^E000^C0036^481061^080033^4C0000^454F46^000003^000000
T^001039^1E^FFC^040030^000030^E0105D^30103F^D8105D^280030^301057^548039^2C105E^38103F
T^001057^0A^800^100036^4C0000^F1^001000
T^00106119^FE^0040030^E01079^301064^508039^DC1079^2C0036^381064^4C0000^05
E^000000
```

Figure 3.7 Object program with relocation by bit mask.

properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185. Therefore, this instruction must begin a new Text record in the object program.

You should carefully examine the remainder of the object program in Fig. 3.7. Make sure you understand how the relocation bits are generated by the assembler and used by the loader.

Some computers provide a hardware relocation capability that eliminates some of the need for the loader to perform program relocation. For example, some such machines consider all memory references to be relative to the beginning of the user's assigned area of memory. The conversion of these relative addresses to actual addresses is performed as the program is executed. (We discuss this further when we study memory management in Chapter 6.) As the next section illustrates, however, the loader must still handle relocation of subprograms in connection with linking.

3.2.2 Program Linking

The basic concepts involved in program linking were introduced in Section 2.3.5. Before proceeding you may want to review that discussion and the examples in that section. In this section we consider more complex examples of external references between programs and examine the relationship between relocation and linking. The next section gives an algorithm for a linking and relocating loader.

Figure 2.15 in Section 2.3.5 showed a program made up of three control sections. These control sections could be assembled together (that is, in the same invocation of the assembler), or they could be assembled independently of one another. In either case, however, they would appear as separate segments of object code after assembly (see Fig. 2.17). The programmer has a natural inclination to think of a program as a logical entity that combines all of the related control sections. From the loader's point of view, however, there is no such thing as a program in this sense—there are only control sections that

are to be linked, relocated, and loaded. The loader has no way of knowing (and no need to know) which control sections were assembled at the same time.

Consider the three (separately assembled) programs in Fig. 3.8, each of which consists of a single control section. Each program contains a list of items (LISTA, LISTB, LISTC); the ends of these lists are marked by the labels ENDA, ENDB, ENDC. The labels on the beginnings and ends of the lists are external symbols (that is, they are available for use in linking). Note that each program contains exactly the same set of references to these external symbols. Three of these are instruction operands (REF1 through REF3), and the others are the values of data words (REF4 through REF8). In considering this example, we examine the differences in the way these identical expressions are handled within the three programs. This emphasizes the relationship between the relocation and linking processes. To focus on these issues, we have not attempted to make these programs appear realistic. All portions of the programs not involved in the relocation and linking process are omitted. The same applies to the generated object programs shown in Fig. 3.9.

Consider first the reference marked REF1. For the first program (PROGA), REF1 is simply a reference to a label within the program. It is assembled in the usual way as a program-counter relative instruction. No modification for relocation or linking is necessary. In PROGB, on the other hand, the same operand refers to an external symbol. The assembler uses an extended-format instruction with address field set to 00000. The object program for PROGB (see Fig. 3.9) contains a Modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked. This reference is handled in exactly the same way for PROGC.

The reference marked REF2 is processed in a similar manner. For PROGA, the operand expression consists of an external reference plus a constant. The assembler stores the value of the constant in the address field of the instruction and a Modification record directs the loader to add to this field the value of LISTB. In PROGB, the same expression is simply a local reference and is assembled using a program-counter relative instruction with no relocation or linking required.

REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes). In PROGA, the assembler has all of the information necessary to compute this value. During the assembly of PROGB (and PROGC), however, the values of the labels are unknown. In these programs, the expression must be assembled as an external reference (with two Modification records) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

Loc		Source statement		Object code
0000	PROGA	START	0	
		EXTDEF	LISTA, ENDA	
		EXTREF	LISTB, ENDB, LISTC, ENDC	
		.		
		.		
		.		
0020	REF1	LDA	LISTA	03201D
0023	REF2	+LDT	LISTB+4	77100004
0027	REF3	LDX	#ENDA-LISTA	050014
		.		
		.		
		.		
0040	LISTA	EQU	*	
		.		
		.		
0054	ENDA	EQU	*	
0054	REF4	WORD	ENDA-LISTA+LISTC	000014
0057	REF5	WORD	ENDC-LISTC-10	FFFFF6
005A	REF6	WORD	ENDC-LISTC+LISTA-1	00003F
005D	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	

Loc		Source statement		Object code
0000	PROGB	START	0	
		EXTDEF	LISTB, ENDB	
		EXTREF	LISTA, ENDA, LISTC, ENDC	
		.		
		.		
		.		
0036	REF1	+LDA	LISTA	03100000
003A	REF2	LDT	LISTB+4	772027
003D	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0060	LISTB	EQU	*	
		.		
		.		
0070	ENDB	EQU	*	
0070	REF4	WORD	ENDA-LISTA+LISTC	000000
0073	REF5	WORD	ENDC-LISTC-10	FFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1	FFFFFF
0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	FFFFF0
007C	REF8	WORD	LISTB-LISTA	000060
		END		

Figure 3.8 Sample programs illustrating linking and relocation

Loc		Source statement		Object code
0000	PROGC	START	0	
		EXTDEF	LISTC, ENDC	
		EXTREF	LISTA, ENDA, LISTB, ENDB	
		.		
		.		
		.		
0018	REF1	+LDA	LISTA	03100000
001C	REF2	+LDT	LISTB+4	77100004
0020	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0030	LISTC	EQU	*	
		.		
		.		
0042	ENDC	EQU	*	
0042	REF4	WORD	ENDA-LISTA+LISTC	000030
0045	REF5	WORD	ENDC-LISTC-10	000008
0048	REF6	WORD	ENDC-LISTC+LISTA-1	000011
004B	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000000
004E	REF8	WORD	LISTB-LISTA	000000
		END		

Figure 3.8 *(cont'd)*

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
:
T^0000200A^03201D^77100004^050014
:
T^0000540F^000014^FFFFF6^00003F^000014^FFFFC0
M^00002405^+LISTB
M^00005406^+LISTC
M^00005706^+ENDC
M^00005706^-LISTC
M^00005A06^+ENDC
M^00005A06^-LISTC
M^00005A06^+PROGA
M^00005D06^-ENDB
M^00005D06^+LISTB
M^00006006^+LISTB
M^00006006^-PROGA
E^000020
```

Figure 3.9 Object programs corresponding to Fig. 3.8.

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
:
T000036 0B 03100000 77202705100000
:
T000070 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05 +LISTA
M00003E 05 +ENDA
M00003E 05 -LISTA
M000070 06 +ENDA
M000070 06 -LISTA
M000070 06 +LISTC
M000073 06 +ENDC
M000073 06 -LISTC
M000076 06 +ENDC
M000076 06 -LISTC
M000076 06 +LISTA
M000079 06 +ENDA
M000079 06 -LISTA
M00007C 06 +PROGB
M00007C 06 -LISTA
E

HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
:
T000018 0C 03100000 77100004 05100000
:
T000042 0F 000030 000008 000011 000000 000000
M000019 05 +LISTA
M00001D 05 +LISTB
M000021 05 +ENDA
M000021 05 -LISTA
M000042 06 +ENDA
M000042 06 -LISTA
M000042 06 +PROGC
M000048 06 +LISTA
M00004B 06 +ENDA
M00004B 06 -LISTA
M00004B 06 -ENDB
M00004B 06 +LISTB
M00004E 06 +LISTB
M00004E 06 -LISTA
E
```

Figure 3.9 *(cont'd)*

The remaining references illustrate a variety of other possibilities. The general approach taken is for the assembler to evaluate as much of the expression as it can. The remaining terms are passed on to the loader via Modification records. To see this, consider REF4. The assembler for PROGA can evaluate all

of the expression in REF4 except for the value of LISTC. This results in an initial value of (hexadecimal) 000014 and one Modification record. However, the same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and three Modification records. For PROGC, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded). The initial value of this data word contains the relative address of LISTC (hexadecimal 000030). Modification records instruct the loader to add the beginning address of the program (i.e., the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA. Thus the expression in REF4 represents a simple external reference for PROGA, a more complicated external reference for PROGB, and a combination of relocation and external references for PROGC.

You should work through references REF5 through REF8 for yourself to be sure you understand how the object code and Modification records in Fig. 3.9 were generated.

Figure 3.10(a) shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following. Note that each of REF4 through REF8 has resulted (after relocation and linking is performed) in the same value in each of the three programs. This is as it should be, since the same source expression appeared in each program.

For example, the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). Figure 3.10(b) shows the details of how this value is computed. The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). In PROGB, the value for REF4 is located at relative address 70 (actual address 40D3). To the initial value (000000), the loader adds the values of ENDA (4054) and LISTC (4112), and subtracts the value of LISTA (4040). The result, 004126, is the same as was obtained in PROGA. Similarly, the computation for REF4 in PROGC results in the same value. The same is also true for each of the other references REF5 through REF8.

For the references that are instruction operands, the calculated values after loading do not always appear to be equal. This is because there is an additional address calculation step involved for program-counter relative (or base relative) instructions. In these cases it is the *target addresses* that are the same. For example, in PROGA the reference REF1 is a program-counter relative instruction with displacement 01D. When this instruction is executed, the program counter contains the value 4023 (the actual address of the next instruction). The resulting target address is 4040. No relocation is necessary for this instruction since the program counter will always contain the actual (not

relative) address of the next instruction. We could also think of this process as automatically providing the needed relocation at execution time through the target address calculation. In PROGB, on the other hand, reference REF1 is an extended format instruction that contains a direct (actual) address. This address, after linking, is 4040—the same as the target address for the same reference in PROGA.

You should work through the details of the other references to see that the target addresses (for REF2 and REF3) or the data values (for REF5 through REF8) are the same in each of the three programs. You do not need to worry about how these calculations are actually performed by the loader because the algorithm and data structures for doing this are discussed in the next section. It is important, however, that you *understand* the calculations to be performed, and that you are able to carry out the computations by hand (following the instructions that are contained in the object programs).

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
3FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4000	.....	.....	.....	.....	
4010	.....	.....	.....	.....	
4020	03201D77	1040C705	0014....	.....	←PROGA
4030	.....	.....	.....	.....	
4040	.....	.....	.....	.....	
4050	.....	00412600	00080040	51000004	
4060	000083..	.....	.....	.....	
4070	.....	.....	.....	.....	
4080	.....	.....	.....	.....	
4090	.....	.....	..031040	40772027	←PROGB
40A0	05100014	.....	.....	.....	
40B0	.....	.....	.....	.....	
40C0	.....	.....	.....	.....	
40D0	00	41260000	08004051	00000400	
40E0	0083....	.....	.....	.....	
40F0	.....	.....	0310	40407710	←PROGC
4100	40C70510	0014....	.....	.....	
4110	.....	.....	.....	.....	
4120	.....	00412600	00080040	51000004	
4130	000083xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

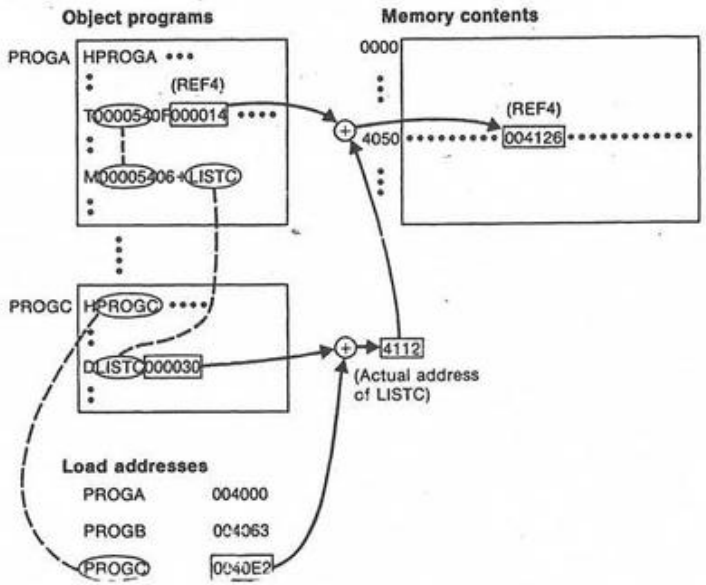


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

3.2.3 Algorithm and Data Structures for a Linking Loader

Now we are ready to present an algorithm for a linking (and relocating) loader. We use Modification records for relocation so that the linking and relocation functions are performed using the same mechanism. As mentioned previously, this type of loader is often found on machines (like SIC/XE) whose relative addressing makes relocation unnecessary for most instructions.

The algorithm for a linking loader is considerably more complicated than the absolute loader algorithm discussed in Section 3.1. The input to such a loader consists of a set of object programs (i.e., control sections) that are to be linked together. It is possible (and common) for a control section to make an external reference to a symbol whose definition does not appear until later in this input stream. In such a case the required linking operation cannot be performed until an address is assigned to the external symbol involved (that is,

until the later control section is read). Thus a linking loader usually makes two passes over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler: Pass 1 assigns addresses to all external symbols, and Pass 2 performs the actual loading, relocation, and linking.

The main data structure needed for our linking loader is an external symbol table ESTAB. This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the name and address of each external symbol in the set of control sections being loaded. The table also often indicates in which control section the symbol is defined. A hashed organization is typically used for this table. Two other important variables are PROGADDR (program load address) and CSADDR (control section address). PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the operating system. (In Chapter 6 we discuss how PROGADDR might be generated within the operating system.) CSADDR contains the starting address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

The algorithm itself is presented in Fig. 3.11. As we discuss this algorithm, you may find it useful to refer to the example of loading and linking in the preceding section (Figs. 3.9 and 3.10).

During the first pass [Fig. 3.11(a)], the loader is concerned only with Header and Define record types in the control sections. The beginning load address for the linked program (PROGADDR) is obtained from the operating system. This becomes the starting address (CSADDR) for the first control section in the input sequence. The control section name from the Header record is entered into ESTAB, with value given by CSADDR. All external symbols appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR. When the End record is read, the control section length CSLTH (which was saved from the Header record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each. Many loaders include as an option the ability to print a *load map* that shows these symbols and their addresses. This information is often useful in program debugging. For the example in Figs. 3.9 and 3.10, such a load map might look like the following. This is essentially the same information contained in ESTAB at the end of Pass 1.

Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR (for first control section)
while not end of input do
   begin
      read next input record (Header record for control section)
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag (duplicate external symbol)
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end (for)
         end (while ≠ 'E')
      add CSLTH to CSADDR (starting address for next control section)
   end (while not EOF)
end (Pass 1)
```

Figure 3.11(a) Algorithm for Pass 1 of a linking loader.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record  (Header record)
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end  {if 'M'}
         end {while ≠ 'E'}
      if an address is specified (in End record) then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR (to start execution of loaded program)
end (Pass 2)
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

Pass 2 of our loader [Fig. 3.11(b)] performs the actual loading, relocation, and linking of the program. CSADDR is used in the same way it was in Pass 1—it always contains the actual starting address of the control section currently being loaded. As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR). When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB. This value is then added to or subtracted from the indicated location in memory.

The last step performed by the loader is usually the transferring of control to the loaded program to begin execution. (On some systems, the address where execution is to begin is simply passed back to the operating system. The user must then enter a separate Execute command.) The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered. If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point. This convention is typical of those found in most linking loaders. Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine. Thus the correct execution address would be specified regardless of the order in which the control sections were presented for loading. (See Fig. 2.17 for an example of this.)

You should apply this algorithm (by hand) to load and link the object programs in Fig. 3.9. If PROGADDR is taken to be 4000, the result should be the same as that shown in Fig. 3.10.

This algorithm can be made more efficient if a slight change is made in the object program format. This modification involves assigning a *reference number* to each external symbol referred to in a control section. This reference number is used (instead of the symbol name) in Modification records.

Suppose we always assign the reference number 01 to the control section name. The other external reference symbols may be assigned numbers as part of the Refer record for the control section. Figure 3.12 shows the object

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
:
T000020A03201D77100004050014
:
T000054E000014FFFFF600003E000014FFFFC0
M000024 05+02
M000054 06+04
M000057 06+05
M000057 06-04
M00005A 06+05
M00005A 06-04
M00005A 06+01
M00005D 06-03
M00005D 06+02
M000060 06+02
M000060 06-01
E000020
```

Figure 3.12 Object programs corresponding to Fig. 3.8 using reference numbers for code modification. (Reference numbers are underlined for easier reading.)

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
:
T0000360B03100000772027 05100000
:
T0000700F000000FFFFF6FFFFFFFFFFF0000060
M000037 05+02
M00003E 05+03
M00003E 05-02
M000070 06+03
M000070 06-02
M000070 06+04
M000073 06+03
M000073 06-04
M000076 06+05
M000076 06-04
M000076 06+02
M000079 06+03
M000079 06-02
M00007C 06+01
M00007C 06-02
E

HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
:
T0000180C03100000771000040 5100000
:
T0000420F000030000008000011000000000000
M000019 05+02
M00001D 05+04
M000021 05+03
M000021 05-02
M000042 06+03
M000042 06-02
M000042 06+01
M000048 06+02
M00004B 06+03
M00004B 06-02
M00004B 06-05
M00004E 06+04
M00004E 06+04
M00004E 06-02
E
```

Figure 3.12 (cont'd)

programs from Fig. 3.9 with this change. The reference numbers are underlined in the Refer and Modification records for easier reading. The common use of a technique such as this is one reason we included Refer records in our object programs. You may have noticed that these records were not used in the algorithm of Fig. 3.11.

The main advantage of this reference-number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section. An external reference symbol can be looked up in ESTAB once for each control section that uses it. The values for code modification can then be obtained by simply indexing into an array of these values. You are encouraged to develop an algorithm that includes this technique, together with any additional data structures you may require.

3.3 MACHINE-INDEPENDENT LOADER FEATURES

In this section we discuss some loader features that are not directly related to machine architecture and design. Loading and linking are often thought of as operating system service functions. The programmer's connection with such services is not as direct as it is with, for example, the assembler during program development. Therefore, most loaders include fewer different features (and less varied capabilities) than are found in a typical assembler.

Section 3.3.1 discusses the use of an automatic library search process for handling external references. This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking.

Section 3.3.2 presents some common options that can be selected at the time of loading and linking. These include such capabilities as specifying alternative sources of input, changing or deleting external references, and controlling the automatic processing of external references.

3.3.1 Automatic Library Search

Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. In most cases there is a standard system library that is used in this way. Other libraries may be specified by control statements or by parameters to the loader. This feature allows the programmer to use subroutines from one or more libraries (for example, mathematical or statistical routines) almost as if they were a part of the programming language. The subroutines called by the program being loaded are

automatically fetched from the library, linked with the main program, and loaded. The programmer does not need to take any action beyond mentioning the subroutine names as external references in the source program. On some systems, this feature is referred to as *automatic library call*. We use the term *library search* to avoid confusion with the call feature found in most programming languages.

Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader. One easy way to do this is to enter symbols from each Refer record into the symbol table (ESTAB) unless these symbols are already present. These entries are marked to indicate that the symbol has not yet been defined. When the definition is encountered, the address assigned to the symbol is filled in to complete the entry. At the end of Pass 1, the symbols in ESTAB that remain undefined represent *unresolved* external references. The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.

Note that the subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved (or until no further resolution can be made). If unresolved external references remain after the library search is completed, these must be treated as errors.

The process just described allows the programmer to override the standard subroutines in the library by supplying his or her own routines. For example, suppose that the main program refers to a standard subroutine named SQRT. Ordinarily the subroutine with this name would automatically be included via the library search function. A programmer who for some reason wanted to use a different version of SQRT could do so simply by including it as input to the loader. By the end of Pass 1 of the loader, SQRT would already be defined, so it would not be included in any library search that might be necessary.

The libraries to be searched by the loader ordinarily contain assembled or compiled versions of the subroutines (that is, object programs). It is possible to search these libraries by scanning the Define records for all of the object programs on the library, but this might be quite inefficient. In most cases a special file structure is used for the libraries. This structure contains a *directory* that gives the name of each routine and a pointer to its address within the file. If a subroutine is to be callable by more than one name (using different entry points), both names are entered into the directory. The object program itself, of course, is only stored once. Both directory entries point to the same copy of the routine. Thus the library search itself really involves a search of the directory, followed by reading the object programs indicated by this search. Some operating systems can keep the directory for commonly used libraries permanently

in memory. This can expedite the search process if a large number of external references are to be resolved.

The process of library search has been discussed as the resolution of a call to a subroutine. Obviously the same technique applies equally well to the resolution of external references to data items.

3.3.2 Loader Options

Many loaders allow the user to specify options that modify the standard processing described in Section 3.2. In this section we discuss some typical loader options and give examples of their use. Many loaders have a special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. Sometimes these same statements can also be embedded in the primary input stream between object programs. On a few systems the programmer can even include loader control statements in the source program, and the assembler or compiler retains these commands as a part of the object program.

We discuss loader options in this section as though they were specified using a command language, but there are other possibilities. On some systems options are specified as a part of the job control language that is processed by the operating system. When this approach is used, the operating system incorporates the options specified into a control block that is made available to the loader when it is invoked. The implementation of such options is, of course, the same regardless of the means used to select them.

One typical loader option allows the selection of alternative sources of input. For example, the command

```
INCLUDE  program-name(library-name)
```

might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

Other commands allow the user to delete external symbols or entire control sections. It may also be possible to change external references within the programs being loaded and linked. For example, the command

```
DELETE  csect-name
```

might instruct the loader to delete the named control section(s) from the set of programs being loaded. The command

```
CHANGE  name1,name2
```

might cause the external symbol *name1* to be changed to *name2* wherever it appears in the object programs. An illustration of the use of such commands is given in the following example.

Consider the source program in Fig. 2.15 and the corresponding object program in Fig. 2.17. There is a main program (COPY) that uses two subprograms (RDREC and WRREC); each of these is a separate control section. If RDREC and WRREC are designed only for use with COPY, it is likely that the three control sections will be assembled at the same time. This means that the three control sections of the object program will appear in the same file (or as part of the same library member).

Suppose now that a set of utility subroutines is made available on the computer system. Two of these, READ and WRITE, are designed to perform the same functions as RDREC and WRREC. It would probably be desirable to change the source program of COPY to use these utility routines. As a temporary measure, however, a sequence of loader commands could be used to make this change without reassembling the program. This might be done, for example, to test the utility routines before the final conversion is made.

Suppose that a file containing the object programs in Fig. 2.17 is the primary loader input with the loader commands

```
INCLUDE  READ(UTLIB)
INCLUDE  WRITE(UTLIB)
DELETE   RDREC, WRREC
CHANGE   RDREC, READ
CHANGE   WRREC, WRITE
```

These commands would direct the loader to include control sections READ and WRITE from the library UTLIB, and to delete the control sections RDREC and WRREC from the load. The first CHANGE command would cause all external references to symbol RDREC to be changed to refer to symbol READ. Similarly, references to WRREC would be changed to WRITE. The result would be exactly the same as if the source program for COPY had been changed to use READ and WRITE. You are encouraged to think for yourself about how the loader might handle such commands to perform the specified processing.

Another common loader option involves the automatic inclusion of library routines to satisfy external references (as described in the preceding section). Most loaders allow the user to specify alternative libraries to be searched, using a statement such as

```
LIBRARY  MYLIB
```


Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

Loaders that perform automatic library search to satisfy external references often allow the user to specify that some references not be resolved in this way. Suppose, for example, that a certain program has as its main function the gathering and storing of data. However, the program can also perform an analysis of the data using the routines STDDEV, PLOT, and CORREL from a statistical library. The user may request this analysis at execution time. Since the program contains external references to these three routines, they would ordinarily be loaded and linked with the program. If it is known that the statistical analysis is not to be performed in a particular execution of this program, the user could include a command such as

```
NOCALL  STDDEV, PLOT, CORREL
```

to instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

It is also possible to specify that *no* external references be resolved by library search. Of course, this means an error will result if the program attempts to make such an external reference during execution. This option is more useful when programs are to be linked but not executed immediately. It is often desirable to postpone the resolution of external references in such a case. In Section 3.4.1 we discuss linkage editors that perform this sort of function.

Another common option involves output from the loader. In Section 3.2.3 we gave an example of a load map that might be generated during the loading process. Through control statements the user can often specify whether or not such a map is to be printed at all. If a map is desired, the level of detail can be selected. For example, the map may include control section names and addresses only. It may also include external symbol addresses or even a cross-reference table that shows references to each external symbol.

Loaders often include a variety of other options. One such option is the ability to specify the location at which execution is to begin (overriding any information given in the object programs). Another is the ability to control whether or not the loader should attempt to execute the program if errors are detected during the load (for example, unresolved external references).

3.4 LOADER DESIGN OPTIONS

In this section we discuss some common alternatives for organizing the loading functions, including relocation and linking. Linking loaders, as described

جامعة الإسكندرية
مكتبة
كلية الآداب

in Section 3.2.3, perform all linking and relocation at load time. We discuss two alternatives to this: linkage editors, which perform linking prior to load time, and dynamic linking, in which the linking function is performed at execution time.

Section 3.4.1 discusses linkage editors, which are found on many computing systems instead of or in addition to the linking loader. A linkage editor performs linking and some relocation; however, the linked program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.

Section 3.4.2 introduces dynamic linking, which uses facilities of the operating system to load and link subprograms at the time they are first called. By delaying the linking process in this way, additional flexibility can be achieved. However, this approach usually involves more overhead than does a linking loader.

In Section 3.4.3 we discuss bootstrap loaders. Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

3.4.1 Linkage Editors

The essential difference between a linkage editor and a linking loader is illustrated in Fig. 3.13. The source program is first assembled or compiled, producing an object program (which may contain several different control sections). A linking loader performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution. A *linkage editor*, on the other hand, produces a linked version of the program (often called a *load module* or an *executable image*), which is written to a file or library for later execution.

When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory. The only object code modification necessary is the addition of an actual load address to relative values within the program. The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required. This involves much less overhead than using a linking loader.

If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required. Resolution

of external references and library searching are only performed once (when the program is link edited). In contrast, a linking loader searches libraries and resolves external references every time the program is executed.

Sometimes, however, a program is reassembled for nearly every execution. This situation might occur in a program development and testing environment (for example, student programs). It also occurs when a program is used so infrequently that it is not worthwhile to store the assembled version in a library. In such cases it is more efficient to use a linking loader, which avoids the steps of writing and reading the linked program.

The linked program produced by the linkage editor is generally in a form that is suitable for processing by a relocating loader. All external references are resolved, and relocation is indicated by some mechanism such as Modification records or a bit mask. Even though all linking has been performed, information concerning external references is often retained in the linked program. This allows subsequent relinking of the program to replace control sections, modify external references, etc. If this information is not retained, the linked program cannot be reprocessed by the linkage editor; it can only be loaded and executed.

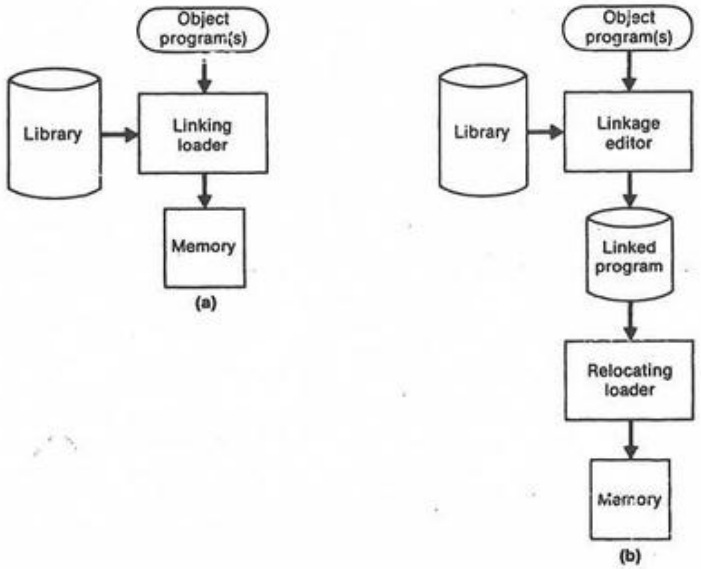


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

If the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation. The result is a linked program that is an exact image of the way the program will appear in memory during execution. The content and processing of such an image are the same as for an absolute object program. Normally, however, the added flexibility of being able to load the program at any location is easily worth the slight additional overhead for performing relocation at load time.

Linkage editors can perform many useful functions besides simply preparing an object program for execution. Consider, for example, a program (PLANNER) that uses a large number of subroutines. Suppose that one subroutine (PROJECT) used by the program is changed to correct an error or to improve efficiency. After the new version of PROJECT is assembled or compiled, the linkage editor can be used to replace this subroutine in the linked version of PLANNER. It is not necessary to go back to the original (separate) versions of all of the other subroutines. The following is a typical sequence of linkage editor commands used to accomplish this. The command language is similar to that discussed in Section 3.3.2.

```
INCLUDE   PLANNER(PROGLIB)
DELETE    PROJECT            {DELETE from existing PLANNER}
INCLUDE   PROJECT(NEWLIB)    {INCLUDE new version}
REPLACE   PLANNER(PROGLIB)
```

Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages. In a typical implementation of FORTRAN, for example, there are a large number of subroutines that are used to handle formatted input and output. These include routines to read and write data blocks, to block and deblock records, and to encode and decode data items according to format specifications. There are a large number of cross-references between these subprograms because of their closely related functions. However, it is desirable that they remain as separate control sections for reasons of program modularity and maintainability.

If a program using formatted I/O were linked in the usual way, all of the cross-references between these library subroutines would have to be processed individually. Exactly the same set of cross-references would need to be processed for almost every FORTRAN program linked. This represents a substantial amount of overhead. The linkage editor could be used to combine the appropriate subroutines into a package with a command sequence like the following:

```
INCLUDE   READR(FTNLIB)
INCLUDE   WRITER(FTNLIB)
```



```
INCLUDE   BLOCK(FTNLIB)
INCLUDE   DEBLOCK(FTNLIB)
INCLUDE   ENCODE(FTNLIB)
INCLUDE   DECODE(FTNLIB)
.
.
.
SAVE      FTNIO(SUBLIB)
```

The linked module named FTNIO could be indexed in the directory of SUBLIB under the same names as the original subroutines. Thus a search of SUBLIB before FTNLIB would retrieve FTNIO instead of the separate routines. Since FTNIO already has all of the cross-references between subroutines resolved, these linkages would not be reprocessed when each user's program is linked. The result would be a much more efficient linkage editing operation for each program and a considerable overall savings for the system.

Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search. Suppose, for example, that 100 FORTRAN programs using the I/O routines described above were to be stored on a library. If all external references were resolved, this would mean that a total of 100 copies of FTNIO would be stored. If library space were an important resource, this might be highly undesirable. Using commands like those discussed in Section 3.3.2, the user could specify that no library search be performed during linkage editing. Thus only the external references between user-written routines would be resolved. A linking loader could then be used to combine the linked user routines with FTNIO at execution time. Because this process involves two separate linking operations, it would require slightly more overhead; however, it would result in a large savings in library space.

Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control, with a corresponding increase in complexity and overhead.

3.4.2 Dynamic Linking

Linkage editors perform linking operations before the program is loaded for execution. Linking loaders perform these same operations at load time. In this section we discuss a scheme that postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called. This type of function is usually called *dynamic linking*, *dynamic loading*, or *load on call*.

Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. For example, run-time support routines for a high-level language like C could be stored in a *dynamic link library*. A single copy of the routines in this library could be loaded into the memory of the computer. All C programs currently in execution could be linked to this one copy, instead of linking a separate copy into each object program.

In an object-oriented system, dynamic linking is often used for references to software objects. This allows the implementation of the object and its methods to be determined at the time the program is run. The implementation can be changed at any time, without affecting the program that makes use of the object. Dynamic linking also makes it possible for one object to be shared by several programs, as discussed previously. (See Section 8.4 for an introduction to object-oriented programming and design.)

Dynamic linking also offers some other advantages over the other types of linking we have discussed. Suppose, for example, that a program contains subroutines that correct or clearly diagnose errors in the input data during execution. If such errors are rare, the correction and diagnostic routines may not be used at all during most executions of the program. However, if the program were completely linked before execution, these subroutines would need to be loaded and linked every time the program is run. Dynamic linking provides the ability to load the routines only when (and if) they are needed. If the subroutines involved are large, or have many external references, this can result in substantial savings of time and memory space.

Similarly, suppose that in any one execution a program uses only a few out of a large number of possible subroutines, but the exact routines needed cannot be predicted until the program examines its input. This situation could occur, for example, with a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library. Input data could be supplied by the user, and results could be displayed at the terminal. In this case, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution. Dynamic linking avoids the necessity of loading the entire library for each execution. As a matter of fact, dynamic linking may make it unnecessary for the program even to know the possible set of subroutines that might be used. The subroutine name might simply be treated as another input item.

There are a number of different mechanisms that can be used to accomplish the actual loading and linking of a called subroutine. Figure 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an operating system service request. This method could also be thought of as a request to a part of the loader that is kept in memory during execution of the program.

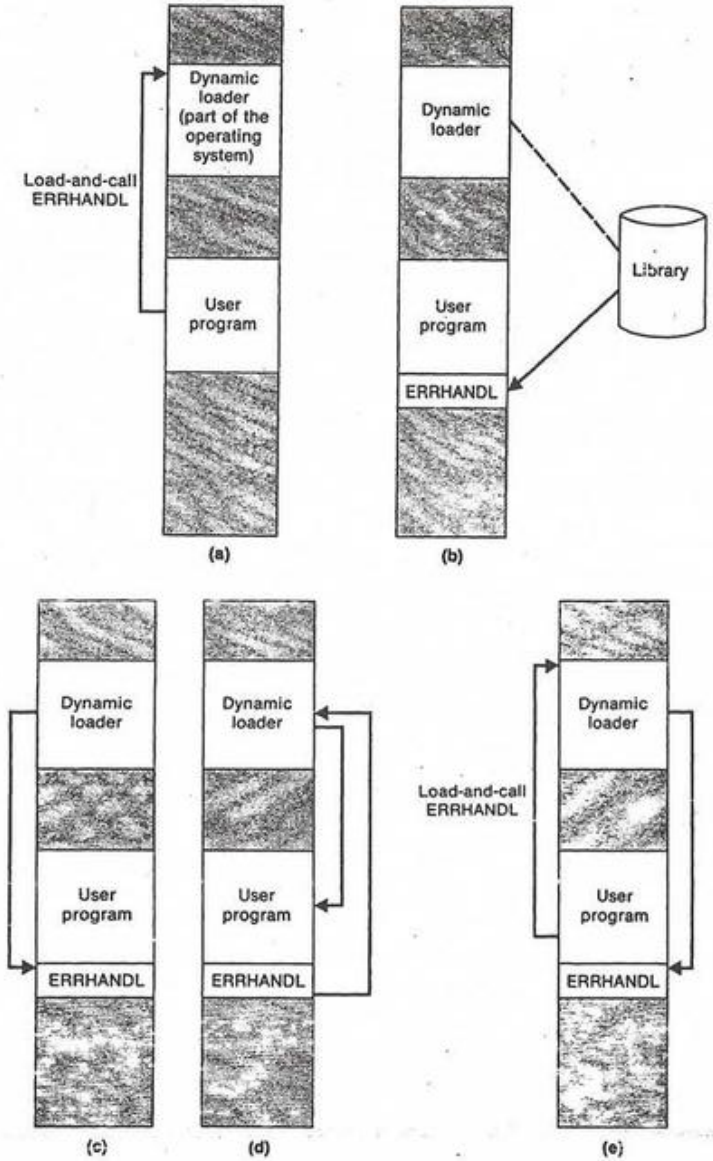


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the operating system. The parameter of this request is the symbolic name of the routine to be called. [See Fig. 3.14(a).] The operating system examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries as shown in Fig. 3.14(b). Control is then passed from the operating system to the routine being called [Fig. 3.14(c)].

When the called subroutine completes its processing, it returns to its caller (that is, to the operating system routine that handles the load-and-call service request). The operating system then returns control to the program that issued the request. This process is illustrated in Fig. 3.14(d). It is important that control be returned in this way so that the operating system knows when the called routine has completed its execution. After the subroutine is completed, the memory that was allocated to load it may be released and used for other purposes. However, this is not always done immediately. Sometimes it is desirable to retain the routine in memory for later use as long as the storage space is not needed for other processing. If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine, as shown in Fig. 3.14(e).

When dynamic linking is used, the association of an actual address with the symbolic name of the called routine is not made until the call statement is executed. Another way of describing this is to say that the *binding* of the name to an actual address is delayed from load time until execution time. This delayed binding results in greater flexibility, as we have discussed. It also requires more overhead since the operating system must intervene in the calling process. In later chapters we see other examples of delayed binding. In those examples, too, delayed binding gives more capabilities at a higher cost.

3.4.3 Bootstrap Loaders

In our discussions of loaders we have neglected to answer one important question: How is the loader itself loaded into memory? Of course, we could say that the operating system loads the loader; however, we are then left with the same question with respect to the operating system. More generally, the question is this: Given an idle computer with no program in memory, how do we get things started?

In this situation, with the machine empty and idle, there is no need for program relocation. We can simply specify the absolute address for whatever program is first loaded. Most often, this program will be the operating system, which occupies a predefined location in memory. This means that we need some means of accomplishing the functions of an absolute loader. Some early computers required the operator to enter into memory the object code for an absolute loader, using switches on the computer console. However, this process is much too inconvenient and error-prone to be a good solution to the problem.

On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs (for example, the operator pressing a "system start" switch), the machine begins to execute this ROM program. On some computers, the program is executed directly in the ROM; on others, the program is copied from ROM to main memory and executed there. However, some machines do not have such read-only storage. In addition, it can be inconvenient to change a ROM program if modifications in the absolute loader are required.

An intermediate solution is to have a built-in hardware function (or a very short ROM program) that reads a fixed-length record from some device into memory at a fixed location. The particular device to be used can often be selected via console switches. After the read operation is complete, control is automatically transferred to the address in memory where the record was stored. This record contains machine instructions that load the absolute program that follows. If the loading process requires more instructions than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of still more records—hence the term *bootstrap*. The first record (or records) is generally referred to as a *bootstrap loader*. (A simple example of such a bootstrap loader was given in Section 3.1.2.) Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system. This includes, for example, the operating system itself and all stand-alone programs that are to be run without an operating system.

3.5 IMPLEMENTATION EXAMPLES

In this section we briefly examine linkers and loaders for actual computers. As in our previous discussions, we make no attempt to give a full description of the linkers and loaders used as examples. Instead we concentrate on any particularly interesting or unusual features, and on differences between these implementations and the more general model discussed earlier in this chapter. We also point out areas in which the linker or loader design is related to the assembler design or to the architecture and characteristics of the machine.

The loader and linker examples we discuss are for the Pentium, SPARC, and T3E architectures. You may want to review the descriptions of these architectures in Chapter 1, and the related assembler examples in Section 2.5.

3.5.1 MS-DOS Linker

This section describes some of the features of the Microsoft MS-DOS linker for Pentium and other x86 systems. Further information can be found in Simrin (1991) and Microsoft (1988).

Most MS-DOS compilers and assemblers (including MASM) produce object modules, not executable machine language programs. By convention, these object modules have the file name extension .OBJ. Each object module contains a binary image of the translated instructions and data of the program. It also describes the structure of the program (for example, the grouping of segments and the use of external references in the program).

MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program. By convention, this executable program has the file name extension .EXE. LINK can also combine the translated programs with other modules from object code libraries, as we discussed previously.

Figure 3.15 illustrates a typical MS-DOS object module. There are also several other possible record types (such as comment records), and there is some flexibility in the order of the records.

The THEADR record specifies the name of the object module. The MODEND record marks the end of the module, and can contain a reference to the entry point of the program. These two records generally correspond to the Header and End records we discussed for SIC/XE.

Record Types	Description
THEADR	Translator header
TYPDEF PUBDEF EXTDEF	External symbols and references
LNAMES SEGDEF GRPDEF	Segment definition and grouping
LEDATA LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module

Figure 3.15 MS-DOS object module.

The PUBDEF record contains a list of the external symbols (called public names) that are defined in this object module. The EXTDEF record contains a list of the external symbols that are referred to in this object module. These records are similar in function to the SIC/XE Define and Refer records. Both PUBDEF and EXTDEF can contain information about the data type designated by an external name. These types are defined in the TYPDEF record.

SEGDEF records describe the segments in the object module, including their name, length, and alignment. GRPDEF records specify how these segments are combined into groups. (See Section 2.5.1 for a discussion of the use of segmentation in the MASM assembler.) The LNAMES record contains a list of all the segment and class names used in the program. SEGDEF and GRPDEF records refer to a segment by giving the position of its name in the LNAMES record. (This approach to specifying names is similar to the "reference number" technique described near the end of Section 3.2.3.)

LEDATA records contain translated instructions and data from the source program, similar to the SIC/XE Text record. LIDATA records specify translated instructions and data that occur in a repeating pattern. (See Exercise 2.1.7.)

FIXUPP records are used to resolve external references, and to carry out address modifications that are associated with relocation and grouping of segments within the program. This is similar to the function performed by the SIC/XE Modification records. However, FIXUPP records are substantially more complex, because of the more complicated object program structure. A FIXUPP record must immediately follow the LEDATA or LIDATA record to which it applies.

LINK performs its processing in two passes, following a similar approach to that described in Section 3.2.3. Pass 1 computes a starting address for each segment in the program. In general, segments are placed into the executable program in the same order that the SEGDEF records are processed. However, in some cases segments from different object modules that have the same segment name and class are combined. Segments with the same class, but different names, are concatenated. The starting address initially associated with a segment is updated during Pass 1 as these combinations and concatenations are performed.

Pass 1 constructs a symbol table that associates an address with each segment (using the LNAMES, SEGDEF, and GRPDEF records) and each external symbol (using the EXTDEF and PUBDEF records). If unresolved external symbols remain after all object modules have been processed, LINK searches the specified libraries as described in Section 3.3.1.

During Pass 2, LINK extracts the translated instructions and data from the object modules, and builds an image of the executable program in memory. It does this because the executable program is organized by segment, not by the

order of the object modules. Building a memory image is the most efficient way to handle the rearrangements caused by combining and concatenating segments. If there is not enough memory available to contain the entire executable image, LINK uses a temporary disk file in addition to all of the available memory.

Pass 2 of LINK processes each LEDATA and LIDATA record along with the corresponding FIXUPP record (if there is one). It places the binary data from LEDATA and LIDATA records into the memory image at locations that reflect the segment addresses computed during Pass 1. (Repeated data specified in LIDATA records is expanded at this time.) Relocations within a segment (caused by combining or grouping segments) are performed, and external references are resolved. Relocation operations that involve the starting address of a segment are added to a table of segment fixups. This table is used to perform relocations that reflect the actual segment addresses when the program is loaded for execution.

After the memory image is complete, LINK writes it to the executable (.EXE) file. This file also includes a header that contains the table of segment fixups, information about memory requirements and entry points, and the initial contents for registers CS and SP.

3.5.2 SunOS Linkers

This section describes some of the features of the SunOS linkers for SPARC systems. Further information can be found in Sun Microsystems (1994b).

SunOS actually provides two different linkers, called the *link-editor* and the *run-time linker*. The link-editor is most commonly invoked in the process of compiling a program. It takes one or more object modules produced by assemblers and compilers, and combines them to produce a single output module. This output module may be one of the following types:

1. A *relocatable object module*, suitable for further link-editing
2. A *static executable*, with all symbolic references bound and ready to run
3. A *dynamic executable*, in which some symbolic references may need to be bound at run time
4. A *shared object*, which provides services that can be bound at run time to one or more dynamic executables

An object module contains one or more *sections*, which represent the instructions and data areas from the source program. Each section has a set of

attributes, such as "executable" and "writeable." (See Section 2.5.2 for a discussion of how sections are defined in an assembler language program.) The object module also includes a list of the relocation and linking operations that need to be performed, and a symbol table that describes the symbols used in these operations.

The SunOS link-editor begins by reading the object modules (or other files) that are presented to it to process. Sections from the input files that have the same attributes are concatenated to form new sections within the output file. The symbol tables from the input files are processed to match symbol definitions and references, and relocation and linking operations within the output file are performed. The linker normally generates a new symbol table, and a new set of relocation instructions, within the output file. These represent symbols that must be bound at run time, and relocations that must be performed when the program is loaded.

Relocation and linking operations are specified using a set of processor-specific codes. These codes describe the size of the field that is to be modified, and the calculation that must be performed. Thus, the set of codes reflects the instruction formats and addressing modes that are found on a particular machine. For example, there are 24 different relocation codes that are used on SPARC systems. SunOS linker implementations on x86 systems use a different set of 11 codes.

Symbolic references from the input files that do not have matching definitions are processed by referring to *archives* and *shared objects*. An archive is a collection of relocatable object modules. A directory stored with the archive associates symbol names with the object modules that contain their definitions. Selected modules from an archive are automatically included to resolve symbolic references, as described in Section 3.3.1.

A shared object is an indivisible unit that was generated by a previous link-edit operation. When the link-editor encounters a reference to a symbol defined in a shared object, the entire contents of the shared object become a *logical* part of the output file. All symbols defined in the object are made available to the link-editing process. However, the shared object is not physically included in the output file. Instead, the link-editor records the dependency on the shared object. The actual inclusion of the shared object is deferred until run time. (This is an example of the dynamic linking approach we discussed in Section 3.4.2. In this case, the use of dynamic linking allows several executing programs to share one copy of a shared object.)

The SunOS run-time linker is used to bind dynamic executables and shared objects at execution time. The linker determines what shared objects are required by the dynamic executable, and ensures that these objects are included. It also inspects the shared objects to detect and process any additional dependencies on other shared objects.

After it locates and includes the necessary shared objects, the linker performs relocation and linking operations to prepare the program for execution. These operations are specified in the relocation and linking sections of the dynamic executable and shared objects. They bind symbols to the actual memory addresses at which the segments are loaded. Binding of data references is performed before control is passed to the executable program. Binding of procedure calls is normally deferred until the program is in execution. During link-editing, calls to globally defined procedures are converted to references to a procedure linkage table. When a procedure is called for the first time, control is passed via this table to the run-time linker. The linker looks up the actual address of the called procedure and inserts it into the linkage table. Thus subsequent calls will go directly to the called procedure, without intervention by the linker. This process is sometimes referred to as *lazy binding*.

The run-time linker also provides an additional level of flexibility. During execution, a program can dynamically bind to new shared objects by requesting the same services of the linker that we have just described. This feature allows a program to choose between a number of shared objects, depending on the exact services required. It also reduces the amount of overhead required for starting a program. If a shared object is not needed during a particular run, it is not necessary to bind it at all. These advantages are similar to those that we discussed for dynamic linking in Section 3.4.2.

3.5.3 Cray MPP Linker

This section describes some of the features of the MPP linker for Cray T3E systems. Further information can be found in Cray Research (1995b).

As we discussed in Chapter 1, a T3E system contains a large number of processing elements (PEs). Each PE has its own local memory. In addition, any PE can access the memory of all other PEs (this is sometimes referred to as *remote memory*). However, the fastest access time always results from a PE accessing its own local memory.

An application program on a T3E system is normally allocated a *partition* that consists of several PEs. (It is possible to run a program in a partition of one PE, but this does not take advantage of the parallel architecture of the machine.) The work to be done by the program is divided between the PEs in the partition. One common method for doing this is to distribute the elements of an array among the PEs. For example, if a partition consists of 16 PEs, the elements of a one-dimensional array might be distributed as shown in Fig. 3.16.

The processing of such an array can also be divided among the PEs. Suppose, for example, that the program contains a loop that processes all 256 array elements. PE0 could execute this loop for subscripts 1 through 16, PE1

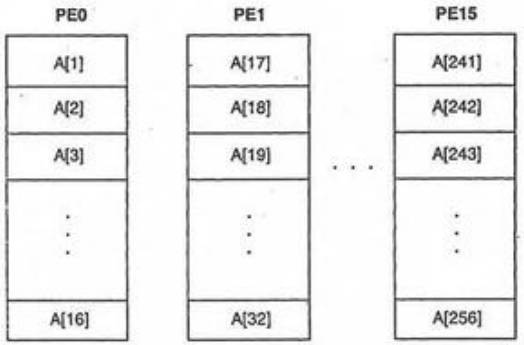


Figure 3.16 Example of data shared between PEs.

could execute the loop for subscripts 17 through 32, and so on. In this way, all of the PEs would share in the array processing, with each PE handling the array elements from its own local memory. Section 5.5.3 describes how this kind of sharing of data and work between PEs can be specified in a program. Section 6.5.4 discusses some of the operating system functions that are used to support the parallel operation of PEs.

Data that is divided among a number of PEs, as in the example just discussed, is called *shared data*. Data that is not shared in this way is called *private data*. In most cases, private data is replicated on each PE in the partition—that is, each PE has its own copy. It is also possible for a PE to have private data items that exist only in its own local memory.

When a program is loaded, each PE gets a copy of the executable code for the program, its private data, and its portion of the shared data. There are a number of possible arrangements of these items, but the overall situation can be visualized as shown in Fig. 3.17. In this diagram, *shared data-i* indicates the portion of the shared data that is assigned to PEi.

The MPP linker organizes blocks of code or data from the object programs into lists. The blocks on a given list all share some common property—for example, executable code, private data, or shared data. The blocks on each list are collected together, an address is assigned to each block, and relocation and linking operations are performed. The linker then writes an executable file that contains the relocated and linked blocks. This executable file also specifies the number of PEs required and other control information.

Notice that the distribution of shared data depends on the number of PEs in the partition. For example, if the partition in Fig. 3.16 contained only 8 PEs, each PE would receive 32 elements of the shared array. If the number of PEs in the partition is specified at compile time, it cannot be overridden later. If the

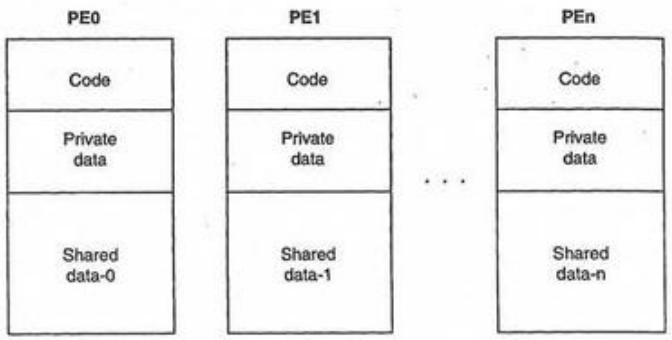


Figure 3.17 T3E program loaded on multiple PEs.

partition size is not specified at compile time, there are two possibilities. The linker can create an executable file that is targeted for a fixed number of PEs, or one that allows the partition size to be chosen at run time. This latter type is called a *plastic* executable. A plastic executable file must contain a copy of all relocatable object modules, and all linker directives that are needed to produce the final executable. Thus, a plastic executable is often considerably larger than one targeted for a fixed number of PEs.

EXERCISES

Section 3.1

1. Define a binary object program format for SIC and write an absolute loader (in SIC assembler language) to load programs in this format.
2. Describe a method for performing the packing required when loading an object program such as that in Fig. 3.1(a), which uses character representation of assembled code. How could you implement this method in SIC assembler language?
3. What would be the advantages and disadvantages of writing a loader using a high-level programming language? What problems might you encounter, and how might these be solved?

3.2 Section Exercises

1. Modify the algorithm given in Fig. 3.11 to use the bit-mask approach to relocation. Linking will still be performed using Modification records.
2. Suppose that a computer primarily uses direct addressing, but has several different instruction formats. What problems does this create for the relocation-bit approach to program relocation? How might these problems be solved?
3. Apply the algorithm described in Fig. 3.11 to link and load the object programs in Fig. 3.9. Compare your results with those shown in Fig. 3.10.
4. Assume that PROGA, PROGB, and PROGC are the same as in Fig. 3.8. Show how the object programs would change (including Text and Modification records) if the following statements were added to each program:

```
REF9    WORD    LISTC
REF10   WORD    LISTB-3
REF11   WORD    LISTA+LISTB
REF12   WORD    ENDC-LISTC-100
REF13   WORD    LISTA-LISTB-ENDA+ENDB
```

5. Apply the algorithm described in Fig. 3.11 to link and load the revised object programs you generated in Exercise 4.
6. Using the methods outlined in Chapter 8, develop a modular design for a relocating and linking loader.
7. Extend the algorithm in Fig. 3.11 to include the detection of improper external reference expressions as suggested in the text. (See Section 2.3.5 for the set of rules to be applied.) What problems arise in performing this kind of error checking?
8. Modify the algorithm in Fig. 3.11 to use the reference-number technique for code modification that is described in Section 3.2.3.
9. Suppose that you are implementing an assembler and loader and want to allow *absolute-valued* external symbols. For example, one control section might contain the statements

```
        EXTDEF  MAXLEN
        .
        .
        .
MAXLEN  EQU     4096
```

and other control sections could refer to the value of MAXLEN as an external symbol. Describe a way of implementing this new feature, including any needed changes in the loader logic and object program format.

10. Suppose that you have been given the task of writing an "unloader"—that is, a piece of software that can take the image of a program that has been loaded and write out an object program that could later be loaded and executed. The computer system uses a relocating loader, so the object program you produce must be capable of being loaded at a location in memory that is different from where your unloader took it. What problems do you see that would prevent you from accomplishing this task?

11. Suppose that you are given two images of a program as it would appear after loading at two *different* locations in memory. Assume that the images represent the program *after* it is loaded and relocated, but *before* any of the program's instructions are actually executed. Describe how this information could be used to accomplish the "unloading" task mentioned in Exercise 10.

12. Some loaders have used an indirect linking scheme. To use such a technique with SIC/XE, the assembler would generate a list of pointer words from the EXTREF directive (one pointer word for each external reference symbol). Modification records would direct the loader to insert the external symbol addresses into the corresponding words in the pointer list. External references would then be accomplished with indirect addressing using these pointers. Thus, for example, an instruction like

```
        LDA     XYZ
```

(where XYZ is an external reference) would be assembled as if it were

```
        LDA     @PXYZ
```


where PXYZ is the pointer word containing the address of XYZ. What would be the advantages and disadvantages of using such a method?

13. Suggest a design for a *one-pass* linking loader. What restrictions (if any) would be required? What would be the advantages and disadvantages of such a one-pass loader?

14. Some programming languages allow data items to be placed in common areas. There may be more than one common area (with different names) in a source program. We may think of each common area as being a separate control section in the object program.

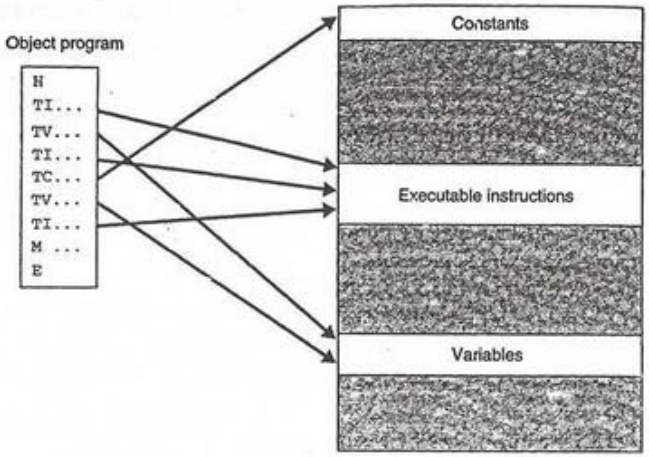
    When object programs are linked and loaded, all of the common areas with the same name are assigned the same starting address in memory. (These common areas may be of different lengths in the different programs declaring them.) This assignment of memory establishes an equivalence between the variables that were declared in common by the different programs. Any data value stored into a common area by one program is thus available to the others.

    How might the loader handle such common areas? (Suggest modifications to the algorithm of Fig. 3.11 that will perform the necessary processing.)

15. Suppose that you have a one-pass assembler that produces object code directly in memory, as described in Section 2.4. This assembler was designed to assemble and run only one control section. Now you want to change it so that it can assemble and run a program that consists of several different control sections (as illustrated in Fig. 2.15).

    Describe the changes you would make to implement this new capability. Your modified assembler should still run in one pass, and should still produce object code in memory, without using any other files.

16. Suppose that a relocatable SIC/XE program is to be loaded in three different parts. One part contains the assembled instructions of the program (LDA, JSUB, etc.). Another part contains the data variables used in the program (which are defined by RESW, RESB, BYTE, and WORD). The third part contains data constants (which are defined by a new assembler directive named CONST).



In the object program, the assembled instructions are contained in type TI records, the variables in type TV records, and the constants in type TC records. (These new record types take the place of the normal Text records in the object program.) The three parts of the object program will be loaded into separate areas of memory, as illustrated above. The starting address for each of the three segments of the program will be supplied to the loader at the time the program is being loaded.

Describe how the assembler could separate the object program into TI, TV, and TC records as described above. Describe how the loader would use the information in these records in loading the program.

17. Consider an extended version of SIC/XE that has a new register R. The contents of R cannot be accessed or changed by the user program. When a program is loaded, however, the loader sets register R so that it contains the starting address of the program. For simplicity, assume that this version of SIC has no program-counter or base relative addressing—thus, all instructions that refer to memory must use Format 4.

    Each time the program refers to an address in memory, the contents of register R are automatically added into the target address calculation. Suppose, for example, that an assembled instruction specifies an address of 800 (hexadecimal). If R contains 5000, executing this instruction would actually refer to memory address 5800. If R contains 8000, executing the same instruction would actually refer to memory address 8800.

Consider the control sections shown in Fig. 3.8. Assume that these control sections are being loaded and linked at the addresses shown in Fig. 3.10; thus the loader will set register R to the value 4000. What value should appear in the External Symbol Table of the loader for the symbol LISTB? What should the instruction labeled REF2 in control section PROGC look like after all loading and linking operations have been performed?

جامعة الإسكندرية
مكتبة

Section 3.3

1. Modify the algorithm in Fig. 3.11 to include automatic library search to resolve external references. You may assume that the details of library access are handled by operating system service routines.
2. Modify the algorithm in Fig. 3.11 to implement CHANGE, DELETE, and INCLUDE directives as described in Section 3.3.2. If you need to place any restrictions on the use of these commands, be sure to state what they are.
3. Suppose that the loader is to produce a listing that shows not only the addresses assigned to external symbols, but also the cross-references between control sections in the program being loaded. What information might be useful in such a listing? Briefly describe how you might implement this feature and include a description of any data structures needed.

Section 3.4

1. Define a module format suitable for representing linked programs produced by a linkage editor. Assume that the linked program is not to be reprocessed by the linkage editor. Describe an algorithm for a relocating loader that would be suitable for the loading of linked programs in this format.

2. Define a module format suitable for representing linked programs produced by a linkage editor. This format should allow for the loading of the linked program by a one-pass relocating loader, as in Exercise 1. However, it should also allow for the linked program to be reprocessed by the linkage editor. Describe how your format allows for both one-pass loading and relinking.

3. Consider the following possibilities for the storage, linking, and execution of a user's program:

   a. Store the source program only; reassemble the program and use a linking loader each time it is to be executed.

   b. Store the source and object versions of the program; use a linking loader each time the program is to be executed.

   c. Store the source program and the linked version with external references to library subroutines left unresolved. Use a linking loader each time the program is to be executed.

   d. Store the source program and the linked version with all external references resolved. Use a relocating loader each time the program is to be executed.

   e. Store the source program and a linked version that has all external references resolved and all relocation performed. Use an absolute loader each time the program is to be executed.

   Under what conditions might each of these approaches be appropriate? Assume that no changes are required in the source program from one execution to the next.

4. Dynamic linking, as described in Section 3.4.2, works for transfers of control only. How could the implementation be extended so that data references could also cause dynamic loading to occur?

5. Suppose that routines that are brought into memory by dynamic loading need not be removed until the termination of the main program. Suggest a way to improve the efficiency of dynamic linking by making it unnecessary for the operating system to be involved in the transfer of control after the routine is loaded.

6. Suppose that it may be necessary to remove from memory routines that were dynamically loaded (to reuse the space). Will the method that you suggested in Exercise 5 still work? What problems arise, and how might they be solved?

7. What kinds of errors might occur during bootstrap loading? What action should the bootstrap loader take for such errors? Modify the SIC/XE bootstrap loader shown in Fig. 3.3 to include such error checking.

Section 3.5

1. Consider the description of the VAX architecture in Section 1.4.1. What characteristics would you expect to find in a VAX linker and loader?

2. Consider the description of the PowerPC architecture in Section 1.5.2, and the description of the PowerPC assembler in Section 2.5.3. What characteristics would you expect to find in a PowerPC linker and loader?

Chapter 4

Macro Processors

In this chapter we study the design and implementation of macro processors. A *macro instruction* (often abbreviated to *macro*) is simply a notational convenience for the programmer. A macro represents a commonly used group of statements in the source programming language. The macro processor replaces each macro instruction with the corresponding group of source language statements. This is called *expanding* the macros. Thus macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macro processor.

For example, suppose that it is necessary to save the contents of all registers before calling a subprogram. On SIC/XE, this would require a sequence of seven instructions (STA, STB, etc.). Using a macro instruction, the programmer could simply write one statement like SAVEREGS. This macro instruction would be expanded into the seven assembler language instructions needed to save the register contents. A similar macro instruction (perhaps named LOADREGS) could be used to reload the register contents after returning from the subprogram.

The functions of a macro processor essentially involve the substitution of one group of characters or lines for another. Except in a few specialized cases, the macro processor performs no analysis of the text it handles. The design and capabilities of a macro processor may be influenced by the *form* of the programming language statements involved. However, the *meaning* of these statements, and their translation into machine language, are of no concern whatsoever during macro expansion. This means that the design of a macro processor is not directly related to the architecture of the computer on which it is to run.

The most common use of macro processors is in assembler language programming. We use SIC assembler language examples to illustrate most of the concepts being discussed. However, macro processors can also be used with high-level programming languages, operating system command languages, etc. In addition, there are general-purpose macro processors that are not tied to any particular language. In the later sections of this chapter, we briefly discuss these more general uses of macros.

Section 4.1 introduces the basic concepts of macro processing, including macro definition and expansion. We also present an algorithm for a simple macro processor. Section 4.2 discusses extended features that are commonly found in macro processors. These features include the generation of unique labels within macro expansions, conditional macro expansion, and the use of keyword parameters in macros. All these features are machine-independent. Because the macro processor is not directly related to machine architecture, this chapter contains no section on machine-dependent features.

Section 4.3 describes some macro processor design options. One of these options (recursive macro expansion) involves the internal structure of the macro processor itself. The other options are concerned with how the macro processor is related to other pieces of system software such as assemblers or compilers.

Finally, Section 4.4 briefly presents three examples of actual macro processors. One of these is a macro processor designed for use by assembler language programmers. Another is intended to be used with a high-level programming language. The third is a general-purpose macro processor, which is not tied to any particular language. Additional examples may be found in the references cited throughout this chapter.

4.1 BASIC MACRO PROCESSOR FUNCTIONS

In this section we examine the fundamental functions that are common to all macro processors. Section 4.1.1 discusses the processes of macro definition, invocation, and expansion with substitution of parameters. These functions are illustrated with examples using the SIC/XE assembler language. Section 4.1.2 presents a one-pass algorithm for a simple macro processor together with a description of the data structures needed for macro processing. Later sections in this chapter discuss extensions to the basic capabilities introduced in this section.

4.1.1 Macro Definition and Expansion

Figure 4.1 shows an example of a SIC/XE program using macro instructions. This program has the same functions and logic as the sample program in Fig. 2.5; however, the numbering scheme used for the source statements has been changed.

This program defines and uses two macro instructions, RDBUFF and WRBUFF. The functions and logic of the RDBUFF macro are similar to those of the RDREC subroutine in Fig. 2.5; likewise, the WRBUFF macro is similar to the WRREC subroutine. The definitions of these macro instructions appear in the source program following the START statement.

Two new assembler directives (MACRO and MEND) are used in macro definitions. The first MACRO statement (line 10) identifies the beginning of a macro definition. The symbol in the label field (RDBUFF) is the name of the macro, and the entries in the operand field identify the *parameters* of the macro instruction. In our macro language, each parameter begins with the character &, which facilitates the substitution of parameters during macro expansion. The macro name and parameters define a pattern or *prototype* for the macro instructions used by the programmer. Following the MACRO directive are the statements that make up the *body* of the macro definition (lines 15 through 90). These are the statements that will be generated as the expansion of the macro. The MEND assembler directive (line 95) marks the end of the macro definition. The definition of the WRBUFF macro (lines 100 through 160) follows a similar pattern.

The main program itself begins on line 180. The statement on line 190 is a *macro invocation* statement that gives the name of the macro instruction being invoked and the *arguments* to be used in expanding the macro. (A macro invocation statement is often referred to as a *macro call*. To avoid confusion with the call statements used for procedures and subroutines, we prefer to use the term *invocation*. As we shall see, the processes of macro invocation and subroutine call are quite different.) You should compare the logic of the main program in Fig. 4.1 with that of the main program in Fig. 2.5, remembering the similarities in function between RDBUFF and RDREC and between WRBUFF and WRREC.

The program in Fig. 4.1 could be supplied as input to a macro processor. Figure 4.2 shows the output that would be generated. The macro instruction definitions have been deleted since they are no longer needed after the macros are expanded. Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, and so on. In expanding the macro invocation on line 190, for example, the argument F1 is substituted for the parameter &INDEV wherever it occurs in the body of the macro. Similarly, BUFFER is substituted for &BUFADR, and LENGTH is substituted for &RECLTH.

```
Line      Source statement

5    COPY     START   0                      COPY FILE FROM INPUT TO OUTPUT
10   RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
15   .
20   .        MACRO TO READ RECORD INTO BUFFER
25   .
30            CLEAR   X                      CLEAR LOOP COUNTER
35            CLEAR   A
40            CLEAR   S
45            +LDT    #4096                  SET MAXIMUM RECORD LENGTH
50            TD      =X'&INDEV'             TEST INPUT DEVICE
55            JEQ     *-3                    LOOP UNTIL READY
60            RD      =X'&INDEV'             READ CHARACTER INTO REG A
65            COMPR   A,S                    TEST FOR END OF RECORD
70            JEQ     *+11                   EXIT LOOP IF EOR
75            STCH    &BUFADR,X              STORE CHARACTER IN BUFFER
80            TIXR    T                      LOOP UNLESS MAXIMUM LENGTH
85            JLT     *-19                     HAS BEEN REACHED
90            STX     &RECLTH                SAVE RECORD LENGTH
95            MEND
100  WRBUFF   MACRO   &OUTDEV,&BUFADR,&RECLTH
105  .
110  .        MACRO TO WRITE RECORD FROM BUFFER
115  .
120           CLEAR   X                      CLEAR LOOP COUNTER
125           LDT     &RECLTH
130           LDCH    &BUFADR,X              GET CHARACTER FROM BUFFER
135           TD      =X'&OUTDEV'            TEST OUTPUT DEVICE
140           JEQ     *-3                    LOOP UNTIL READY
145           WD      =X'&OUTDEV'            WRITE CHARACTER
150           TIXR    T                      LOOP UNTIL ALL CHARACTERS
155           JLT     *-14                     HAVE BEEN WRITTEN
160           MEND
165  .
170  .        MAIN PROGRAM
175  .
180  FIRST    STL     RETADR                 SAVE RETURN ADDRESS
190  CLOOP    RDBUFF  F1,BUFFER,LENGTH       READ RECORD INTO BUFFER
195           LDA     LENGTH                 TEST FOR END OF FILE
200           COMP    #0
205           JEQ     ENDFIL                 EXIT IF EOF FOUND
210           WRBUFF  05,BUFFER,LENGTH       WRITE OUTPUT RECORD
215           J       CLOOP                  LOOP
220  ENDFIL   WRBUFF  05,EOF,THREE           INSERT EOF MARKER
225           J       @RETADR
230  EOF      BYTE    C'EOF'
235  THREE    WORD    3
240  RETADR   RESW    1
245  LENGTH   RESW    1                      LENGTH OF RECORD
250  BUFFER   RESB    4096                   4096-BYTE BUFFER AREA
255           END     FIRST
```

Figure 4.1 Use of macros in a SIC/XE program.


```
Line      Source statement

5    COPY     START   0                      COPY FILE FROM INPUT TO OUTPUT
180  FIRST    STL     RETADR                 SAVE RETURN ADDRESS
190  .CLOOP   RDBUFF  F1,BUFFER,LENGTH       READ RECORD INTO BUFFER
190a CLOOP    CLEAR   X                      CLEAR LOOP COUNTER
190b          CLEAR   A
190c          CLEAR   S
190d          +LDT    #4096                  SET MAXIMUM RECORD LENGTH
190e          TD      =X'F1'                 TEST INPUT DEVICE
190f          JEQ     *-3                    LOOP UNTIL READY
190g          RD      =X'F1'                 READ CHARACTER INTO REG A
190h          COMPR   A,S                    TEST FOR END OF RECORD
190i          JEQ     *+11                   EXIT LOOP IF EOR
190j          STCH    BUFFER,X               STORE CHARACTER IN BUFFER
190k          TIXR    T                      LOOP UNLESS MAXIMUM LENGTH
190l          JLT     *-19                     HAS BEEN REACHED
190m          STX     LENGTH                 SAVE RECORD LENGTH
195           LDA     LENGTH                 TEST FOR END OF FILE
200           COMP    #0
205           JEQ     ENDFIL                 EXIT IF EOF FOUND
210           WRBUFF  05,BUFFER,LENGTH       WRITE OUTPUT RECORD
210a          CLEAR   X                      CLEAR LOOP COUNTER
210b          LDT     LENGTH
210c          LDCH    BUFFER,X               GET CHARACTER FROM BUFFER
210d          TD      =X'05'                 TEST OUTPUT DEVICE
210e          JEQ     *-3                    LOOP UNTIL READY
210f          WD      =X'05'                 WRITE CHARACTER
210g          TIXR    T                      LOOP UNTIL ALL CHARACTERS
210h          JLT     *-14                     HAVE BEEN WRITTEN
215           J       CLOOP                  LOOP
220  .ENDFIL  WRBUFF  05,EOF,THREE           INSERT EOF MARKER
220a ENDFIL   CLEAR   X                      CLEAR LOOP COUNTER
220b          LDT     THREE
220c          LDCH    EOF,X                  GET CHARACTER FROM BUFFER
220d          TD      =X'05'                 TEST OUTPUT DEVICE
220e          JEQ     *-3                    LOOP UNTIL READY
220f          WD      =X'05'                 WRITE CHARACTER
220g          TIXR    T                      LOOP UNTIL ALL CHARACTERS
220h          JLT     *-14                     HAVE BEEN WRITTEN
225           J       @RETADR
230  EOF      BYTE    C'EOF'
235  THREE    WORD    3
240  RETADR   RESW    1
245  LENGTH   RESW    1                      LENGTH OF RECORD
250  BUFFER   RESB    4096                   4096-BYTE BUFFER AREA
255           END     FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

Lines 190a through 190m show the complete expansion of the macro invocation on line 190. The comment lines within the macro body have been deleted, but comments on individual statements have been retained. Note that the macro invocation statement itself has been included as a comment line. This serves as documentation of the statement written by the programmer. The label on the macro invocation statement (CLOOP) has been retained as a label on the first statement generated in the macro expansion. This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic. The macro invocations on lines 210 and 220 are expanded in the same way. Note that the two invocations of WRBUFF specify different arguments, so they produce different expansions.

After macro processing, the expanded file (Fig. 4.2) can be used as input to the assembler. The macro invocation statements will be treated as comments, and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

A comparison of the expanded program in Fig. 4.2 with the program in Fig. 2.5 shows the most significant differences between macro invocation and subroutine call. In Fig. 4.2, the statements from the body of the macro WRBUFF are generated twice: lines 210a through 210h and lines 220a through 220h. In the program of Fig. 2.5, the corresponding statements appear only once: in the subroutine WRREC (lines 210 through 240). In general, the statements that form the expansion of a macro are generated (and assembled) each time the macro is invoked. Statements in a subroutine appear only once, regardless of how many times the subroutine is called.

Note also that our macro instructions have been written so that the body of the macro contains no labels. In Fig. 4.1, for example, line 140 contains the statement "JEQ *–3" and line 155 contains "JLT *–14." The corresponding statements in the WRREC subroutine (Fig. 2.5) are "JEQ WLOOP" and "JLT WLOOP," where WLOOP is a label on the TD instruction that tests the output device. If such a label appeared on line 135 of the macro body, it would be generated twice—on lines 210d and 220d of Fig. 4.2. This would result in an error (a duplicate label definition) when the program is assembled. To avoid duplication of symbols, we have eliminated labels from the body of our macro definitions.

The use of statements like "JLT *–14" is generally considered to be a poor programming practice. It is somewhat less objectionable within a macro definition; however, it is still an inconvenient and error-prone method. In Section 4.2.2 we discuss ways of avoiding this problem.

4.1.2 Macro Processor Algorithm and Data Structures

It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass, and all macro invocation statements are expanded during the second pass. However, such a two-pass macro processor would not allow the body of one macro instruction to contain definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

Such definitions of macros by other macros can be useful in certain cases. Consider, for example, the two macro instruction definitions in Fig. 4.3. The body of the first macro (MACROS) contains statements that define RDBUFF, WRBUFF, and other macro instructions for a SIC system (standard version). The body of the second macro instruction (MACROX) defines these same macros for a SIC/XE system. A program that is to be run on a standard SIC system could invoke MACROS to define the other utility macro instructions. A program for a SIC/XE system could invoke MACROX to define these same macros in their XE versions. In this way, the same program could run on either a standard SIC machine or a SIC/XE machine (taking advantage of the extended features). The only change required would be the invocation of either MACROS or MACROX. It is important to understand that *defining* MACROS or MACROX does not define RDBUFF and the other macro instructions. These definitions are processed only when an invocation of MACROS or MACROX is *expanded*.

A one-pass macro processor that can alternate between macro definition and macro expansion is able to handle macros like those in Fig. 4.3. In this section we present an algorithm and a set of data structures for such a macro processor. Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro. This restriction does not create any real inconvenience for the programmer. In fact, a macro invocation statement that preceded the definition of the macro would be confusing for anyone reading the program.

There are three main data structures involved in our macro processor. The macro definitions themselves are stored in a definition table (DEFTAB), which contains the macro prototype and the statements that make up the macro body (with a few modifications). Comment lines from the macro definition are not entered into DEFTAB because they will not be part of the macro expansion. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments. The macro names are also entered into NAMTAB, which serves as an index to DEFTAB. For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in DEFTAB.

```
1  MACROS   MACRO    {Defines SIC standard version macros}
2  RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                .
                .    {SIC  standard version}
                .
3           MEND     {End of RDBUFF}
4  WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                .
                .    {SIC standard version}
                .
5           MEND     {End of WRBUFF}
                .
                .
                .
6           MEND     {End of MACROS}
```

(a)

```
1  MACROX   MACRO    {Defines  SIC/XE macros}
2  RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                .
                .    {SIC/XE version}
                .
3           MEND     {End of RDBUFF}
4  WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                .
                .    {SIC/XE version}
                .
5           MEND     {End of WRBUFF}
                .
                .
                .
6           MEND     {End of MACROX}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.

The third data structure is an argument table (ARGTAB), which is used during the expansion of macro invocations. When a macro invocation statement is recognized, the arguments are stored in ARGTAB according to their position in the argument list. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.

Figure 4.4 shows portions of the contents of these tables during the processing of the program in Fig. 4.1. Figure 4.4(a) shows the definition of RDBUFF stored in DEFTAB, with an entry in NAMTAB identifying the beginning and end of the definition. Note the positional notation that has been used

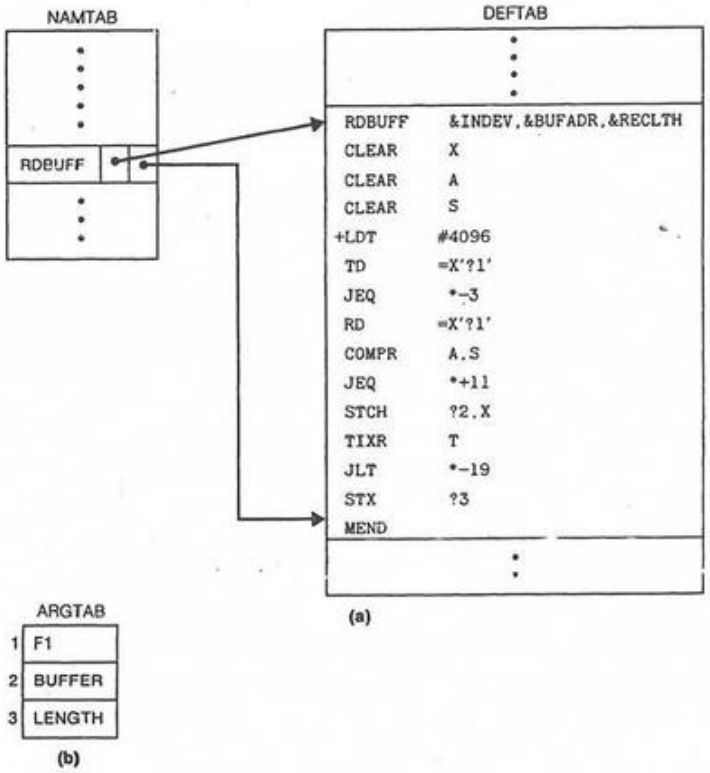


Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

for the parameters: the parameter &INDEV has been converted to *?1* (indicating the first parameter in the prototype), &BUFADR has been converted to *?2*, and so on. Figure 4.4(b) shows ARGTAB as it would appear during expansion of the RDBUFF statement on line 190. For this invocation, the first argument is F1, the second is BUFFER, etc. This scheme makes substitution of macro arguments much more efficient. When the *?n* notation is recognized in a line from DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

The macro processor algorithm itself is presented in Fig. 4.5. The procedure DEFINE, which is called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB. EXPAND is called to set up the argument values in ARGTAB and expand a macro invocation statement. The procedure GETLINE, which is called at several points in the algorithm, gets the next line to be processed. This line may come from DEFTAB (the next line of a macro being expanded), or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

One aspect of this algorithm deserves further comment: the handling of macro definitions within macros (as illustrated in Fig. 4.3). When a macro definition is being entered into DEFTAB, the normal approach would be to continue until an MEND directive is reached. This would not work for the example in Fig. 4.3, however. The MEND on line 3 (which actually marks the end of the definition of RDBUFF) would be taken as the end of the definition of MACROS. To solve this problem, our DEFINE procedure maintains a counter named LEVEL. Each time a MACRO directive is read, the value of LEVEL is increased by 1; each time an MEND directive is read, the value of LEVEL is decreased by 1. When LEVEL reaches 0, the MEND that corresponds to the original MACRO directive has been found. This process is very much like matching left and right parentheses when scanning an arithmetic expression.

```
begin {macro processor}
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}
```

Figure 4.5 Algorithm for a one-pass macro processor.


```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL := 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}


procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition (prototype) from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}


procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

Figure 4.5 (*cont'd*)

You may want to apply this algorithm by hand to the program in Fig. 4.1 to be sure you understand its operation. The result should be the same as shown in Fig. 4.2.

Most macro processors allow the definitions of commonly used macro instructions to appear in a standard system library, rather than in the source program. This makes the use of such macros much more convenient. Definitions are retrieved from this library as they are needed during macro processing. The extension of the algorithm in Fig. 4.5 to include this sort of processing appears as an exercise at the end of this chapter.

4.2 MACHINE-INDEPENDENT MACRO PROCESSOR FEATURES

In this section we discuss several extensions to the basic macro processor functions presented in Section 4.1. As we have mentioned before, these extended features are not directly related to the architecture of the computer for which the macro processor is written. Section 4.2.1 describes a method for concatenating macro instruction parameters with other character strings. Section 4.2.2 discusses one method for generating unique labels within macro expansions, which avoids the need for extensive use of relative addressing at the source statement level. Section 4.2.3 introduces the important topic of conditional macro expansion and illustrates the concepts involved with several examples. This ability to alter the expansion of a macro by using control statements makes macro instructions a much more powerful and useful tool for the programmer. Section 4.2.4 describes the definition and use of keyword parameters in macro instructions.

4.2.1 Concatenation of Macro Parameters

Most macro processors allow parameters to be concatenated with other character strings. Suppose, for example, that a program contains one series of variables named by the symbols XA1, XA2, XA3, ..., another series named by XB1, XB2, XB3, ..., etc. If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction. The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

```
LDA     X&ID1
```

in which the parameter &ID is concatenated after the character string X and before the character string 1. Closer examination, however, reveals a problem with such a statement. The beginning of the macro parameter is identified by the starting symbol &; however, the end of the parameter is not marked. Thus the operand in the foregoing statement could equally well represent the character string X followed by the parameter &ID1. In this particular case, the macro processor could potentially deduce the meaning that was intended. However, if the macro definition contained both &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.

Most macro processors deal with this problem by providing a special *concatenation operator*. In the SIC macro language, this operator is the character →. Thus the previous statement would be written as

```
LDA     X&ID→1
```

so that the end of the parameter &ID is clearly identified. The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so the character → will not appear in the macro expansion.

Figure 4.6(a) shows a macro definition that uses the concatenation operator as previously described. Figure 4.6(b) and (c) shows macro invocation statements and the corresponding macro expansions. You should work through the generation of these macro expansions for yourself to be sure you understand how the concatenation operators are handled. You are also encouraged to think about how the concatenation operator would be handled in a macro processing algorithm like the one given in Fig. 4.5.

4.2.2 Generation of Unique Labels

As we discussed in Section 4.1, it is in general not possible for the body of a macro instruction to contain labels of the usual kind. This leads to the use of relative addressing at the source statement level. Consider, for example, the definition of WRBUFF in Fig. 4.1. If a label were placed on the TD instruction on line 135, this label would be defined twice—once for each invocation of

```
1 SUM    MACRO    &ID
2        LDA      X&ID→1
3        ADD      X&ID→2
4        ADD      X&ID→3
5        STA      X&ID→S
6        MEND
```

(a)

```
SUM      A
↓
LDA      XA1
ADD      XA2
ADD      XA3
STA      XAS
```

(b)

```
SUM      BETA
↓
LDA      XBETA1
ADD      XBETA2
ADD      XBETA3
STA      XBETAS
```

(c)

Figure 4.6 Concatenation of macro parameters.

WRBUFF. This duplicate definition would prevent correct assembly of the resulting expanded program.

Because it was not possible to place a label on line 135 of this macro definition, the Jump instructions on lines 140 and 155 were written using the relative operands *–3 and *–14. This sort of relative addressing in a source statement may be acceptable for short jumps such as "JEQ *–3." However, for longer jumps spanning several instructions, such notation is very inconvenient, error-prone, and difficult to read. Many macro processors avoid these problems by allowing the creation of special types of labels within macro instructions.

Figure 4.7 illustrates one technique for generating unique labels within a macro expansion. A definition of the RDBUFF macro is shown in Fig. 4.7(a). Labels used within the macro body begin with the special character $. Figure 4.7(b) shows a macro invocation statement and the resulting macro expansion. Each symbol beginning with $ has been modified by replacing $ with $AA. More generally, the character $ will be replaced by $xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded. For the first macro expansion in a program, xx will have the value AA. For succeeding macro expansions, xx will be set to AB, AC, etc. (If only alphabetic and numeric characters are allowed in xx, such a two-character counter provides for as many as 1296 macro expansions in a single program.) This results in the generation of unique labels for each expansion of a macro instruction. For further examples, see Figs. 4.8 and 4.10.

The SIC assembler language allows the use of the character $ in symbols; however, programmers are instructed not to use this character in their source programs. This avoids any possibility of conflict between programmer-generated symbols and those created by the macro processor.

4.2.3 Conditional Macro Expansion

In all of our previous examples of macro instructions, each invocation of a particular macro was expanded into the same sequence of statements. These statements could be varied by the substitution of parameters, but the form of the statements, and the order in which they appeared, were unchanged. A simple macro facility such as this can be a useful tool. Most macro processors, however, can also modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation. Such a capability adds greatly to the power and flexibility of a macro language. In this section we present a typical set of conditional macro expansion statements. Other examples are found in the macro processor descriptions in Section 4.4.

The term *conditional assembly* is commonly used to describe features such as those discussed in this section. However, there are applications of macro processors that are not related to assemblers or assembler language programming. For this reason, we prefer to use the term *conditional macro expansion.*

The use of one type of conditional macro expansion statement is illustrated in Fig. 4.8. Figure 4.8(a) shows a definition of a macro RDBUFF, the logic and functions of which are similar to those previously discussed. However, this definition of RDBUFF has two additional parameters: &EOR, which specifies a

hexadecimal character code that marks the end of a record, and &MAXLTH, which specifies the maximum length record that can be read. (As we shall see, it is possible for either or both of these parameters to be omitted in an invocation of RDBUFF.)

```
RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096          SET MAXIMUM RECORD LENGTH
$LOOP    TD       =X'&INDEV'     TEST INPUT DEVICE
         JEQ      $LOOP          LOOP UNTIL READY
         RD       =X'&INDEV'     READ CHARACTER INTO REG A
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $EXIT          EXIT LOOP IF EOR
         STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP            HAS BEEN REACHED
$EXIT    STX      &RECLTH        SAVE RECORD LENGTH
         MEND
```

(a)

```
         RDBUFF   F1,BUFFER,LENGTH

         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096          SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F1'         TEST INPUT DEVICE
         JEQ      $AALOOP        LOOP UNTIL READY
         RD       =X'F1'         READ CHARACTER INTO REG A
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $AAEXIT        EXIT LOOP IF EOR
         STCH     BUFFER,X       STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $AALOOP          HAS BEEN REACHED
$AAEXIT  STX      LENGTH         SAVE RECORD LENGTH
```

(b)

Figure 4.7 Generation of unique labels within macro expansion.


```
RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
         IF       (&EOR NE '')
&EORCK   SET      1
         ENDIF
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'       SET EOR CHARACTER
         RMO      A,S
         ENDIF
         IF       (&MAXLTH EQ '')
        +LDT      #4096          SET MAX LENGTH = 4096
         ELSE
        +LDT      #&MAXLTH       SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD       =X'&INDEV'     TEST INPUT DEVICE
         JEQ      $LOOP          LOOP UNTIL READY
         RD       =X'&INDEV'     READ CHARACTER INTO REG A
         IF       (&EORCK EQ 1)
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $EXIT          EXIT LOOP IF EOR
         ENDIF
         STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP            HAS BEEN REACHED
$EXIT    STX      &RECLTH        SAVE RECORD LENGTH
         MEND
```

(a)

```
         RDBUFF   F3,BUF,RECL,04,2048

         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         LDCH     =X'04'         SET EOR CHARACTER
         RMO      A,S
        +LDT      #2048          SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F3'         TEST INPUT DEVICE
         JEQ      $AALOOP        LOOP UNTIL READY
         RD       =X'F3'         READ CHARACTER INTO REG A
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $AAEXIT        EXIT LOOP IF EOR
         STCH     BUF,X          STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $AALOOP          HAS BEEN REACHED
$AAEXIT  STX      RECL           SAVE RECORD LENGTH
```

(b)

Figure 4.8 Use of macro-time conditional statements.

```
        .       RDBUFF   0E,BUFFER,LENGTH,,80

                         .
30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
47             +LDT      #80            SET MAXIMUM RECORD LENGTH
50      $ABLOOP TD       =X'0E'         TEST INPUT DEVICE
55              JEQ      $ABLOOP        LOOP UNTIL READY
60              RD       =X'0E'         READ CHARACTER INTO REG A
75              STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
87              JLT      $ABLOOP          HAS BEEN REACHED
90      $ABEXIT STX      LENGTH         SAVE RECORD LENGTH
```

(c)

```
        .       RDBUFF   F1,BUFF,RLENG,04

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
40              LDCH     =X'04'         SET EOR CHARACTER
42              RMO      A,S
45             +LDT      #4096          SET MAX LENGTH = 4096
50      $ACLOOP TD       =X'F1'         TEST INPUT DEVICE
55              JEQ      $ACLOOP        LOOP UNTIL READY
60              RD       =X'F1'         READ CHARACTER INTO REG A
65              COMPR    A,S            TEST FOR END OF RECORD
70              JEQ      $ACEXIT        EXIT LOOP IF EOR
75              STCH     BUFF,X         STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $ACLOOP          HAS BEEN REACHED
90      $ACEXIT STX      RLENG          SAVE RECORD LENGTH
```

(d)

Figure 4.8 *(cont'd)*

The statements on lines 44 through 48 of this definition illustrate a simple macro-time conditional structure. The IF statement evaluates a Boolean expression that is its operand. If the value of this expression is TRUE, the statements following the IF are generated until an ELSE is encountered. Otherwise, these statements are skipped, and the statements following the ELSE are generated. The ENDIF statement terminates the conditional expression that was begun by the IF statement. (As usual, the ELSE clause can be omitted entirely.) Thus if the parameter &MAXLTH is equal to the null string

(that is, if the corresponding argument was omitted in the macro invocation statement), the statement on line 45 is generated. Otherwise, the statement on line 47 is generated.

A similar structure appears on lines 26 through 28. In this case, however, the statement controlled by the IF is not a line to be generated into the macro expansion. Instead, it is another macro processor directive (SET). This SET statement assigns the value 1 to &EORCK. The symbol &EORCK is a *macro-time variable* (also often called a *set symbol*), which can be used to store working values during the macro expansion. Any symbol that begins with the character & and that is not a macro instruction parameter is assumed to be a macro-time variable. All such variables are initialized to a value of 0. Thus if there is an argument corresponding to &EOR (that is, if &EOR is not null), the variable &EORCK is set to 1. Otherwise, it retains its default value of 0. The value of this macro-time variable is used in the conditional structures on lines 38 through 43 and 63 through 73.

In the previous example the value of the macro-time variable &EORCK was used to store the result of the comparison involving &EOR (line 26). The IF statements that use this value (lines 38 and 63) could, of course, simply have repeated the original test. However, the use of a macro-time variable makes it clear that the same logical condition is involved in both IF statements. Examining the value of the variable may also be faster than repeating the original test, especially if the test involves a complicated Boolean expression rather than just a single comparison.

Figure 4.8(b–d) shows the expansion of three different macro invocation statements that illustrate the operation of the IF statements in Fig. 4.8(a). You should carefully work through these examples to be sure you understand how the given macro expansion was obtained from the macro definition and the macro invocation statement.

The implementation of the conditional macro expansion features just described is relatively simple. The macro processor must maintain a symbol table that contains the values of all macro-time variables used. Entries in this table are made or modified when SET statements are processed. The table is used to look up the current value of a macro-time variable whenever it is required.

When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated. If the value of this expression is TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement. If an ELSE is found, the macro processor then skips lines in DEFTAB until the next ENDIF. Upon reaching the ENDIF, it resumes expanding the macro in the usual way. If the value of the specified Boolean expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion.

The implementation outlined above does not allow for nested IF structures. You are encouraged to think about how this technique could be modified to handle such nested structures (see Exercise 4.2.10).

It is extremely important to understand that the testing of Boolean expressions in IF statements occurs at the time macros are *expanded*. By the time the program is assembled, all such decisions have been made. There is only one sequence of source statements [for example, the statements in Fig. 4.8(c)], and the conditional macro expansion directives have been removed. Thus macro-time IF statements correspond to options that might have been selected by the programmer in writing the source code. They are fundamentally different from statements such as COMPR (or IF statements in a high-level programming language), which test data values during program *execution*. The same applies to the assignment of values to macro-time variables, and to the other conditional macro expansion directives we discuss.

The macro-time IF-ELSE-ENDIF structure provides a mechanism for either generating (once) or skipping selected statements in the macro body. A different type of conditional macro expansion statement is illustrated in Fig. 4.9. Figure 4.9(a) shows another definition of RDBUFF. The purpose and function of the macro are the same as before. With this definition, however, the programmer can specify a list of end-of-record characters. In the macro invocation statement in Fig. 4.9(b), for example, there is a list (00,03,04) corresponding to the parameter &EOR. Any one of these characters is to be interpreted as marking the end of a record. To simplify the macro definition, the parameter &MAXLTH has been deleted; the maximum record length will always be 4096.

The definition in Fig. 4.9(a) uses a macro-time looping statement WHILE. The WHILE statement specifies that the following lines, until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. As before, the testing of this condition, and the looping, are done while the macro is being expanded. The conditions to be tested involve macro-time variables and arguments, not run-time data values.

The use of the WHILE-ENDW structure is illustrated on lines 63 through 73 of Fig. 4.9(a). The macro-time variable &EORCT has previously been set (line 27) to the value %NITEMS(&EOR). %NITEMS is a macro processor function that returns as its value the number of members in an argument list. For example, if the argument corresponding to &EOR is (00,03,04), then %NITEMS(&EOR) has the value 3.

The macro-time variable &CTR is used to count the number of times the lines following the WHILE statement have been generated. The value of &CTR is initialized to 1 (line 63), and incremented by 1 each time through the loop (line 71). The WHILE statement itself specifies that the macro-time loop will continue to be executed as long as the value of &CTR is less than or equal to the value of &EORCT. This means that the statements on lines 65 and 70


```
25   RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR
27   &EORCT   SET     %NITEMS(&EOR)
30            CLEAR   X                 CLEAR LOOP COUNTER
35            CLEAR   A
45           +LDT     #4096             SET MAX LENGTH = 4096
50   $LOOP    TD      =X'&INDEV'        TEST INPUT DEVICE
55            JEQ     $LOOP             LOOP UNTIL READY
60            RD      =X'&INDEV'        READ CHARACTER INTO REG A
63   &CTR     SET     1
64            WHILE   (&CTR LE &EORCT)
65            COMP    =X'0000&EOR[&CTR]'
70            JEQ     $EXIT
71   &CTR     SET     &CTR+1
73            ENDW
75            STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
80            TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
85            JLT     $LOOP               HAS BEEN REACHED
90   $EXIT    STX     &RECLTH           SAVE RECORD LENGTH
100           MEND
```

(a)

```
.             RDBUFF  F2,BUFFER,LENGTH,(00,03,04)

30            CLEAR   X                 CLEAR LOOP COUNTER
35            CLEAR   A
45           +LDT     #4096             SET MAX LENGTH = 4096
50   $AALOOP  TD      =X'F2'            TEST INPUT DEVICE
55            JEQ     $AALOOP           LOOP UNTIL READY
60            RD      =X'F2'            READ CHARACTER INTO REG A
65            COMP    =X'000000'
70            JEQ     $AAEXIT
65            COMP    =X'000003'
70            JEQ     $AAEXIT
65            COMP    =X'000004'
70            JEQ     $AAEXIT
75            STCH    BUFFER,X          STORE CHARACTER IN BUFFER
80            TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
85            JLT     $AALOOP             HAS BEEN REACHED
90   $AAEXIT  STX     LENGTH            SAVE RECORD LENGTH
```

(b)

Figure 4.9 Use of macro-time looping statements.

will be generated once for each member of the list corresponding to the parameter &EOR. The value of &CTR is used as a subscript to select the proper member of the list for each iteration of the loop. Thus on the first iteration the expression &EOR[&CTR] on line 65 has the value 00; on the second iteration it has the value 03, and so on.

Figure 4.9(b) shows the expansion of a macro invocation statement using the definition in Fig. 4.9(a). You should examine this example carefully to be sure you understand how the WHILE statements are handled.

The implementation of a macro-time looping statement such as WHILE is also relatively simple. When a WHILE statement is encountered during macro expansion, the specified Boolean expression is evaluated. If the value of this expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ENDW statement, and then resumes normal macro expansion. If the value of the Boolean expression is TRUE, the macro processor continues to process lines from DEFTAB in the usual way until the next ENDW statement. When the ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value of this expression as previously described.

This method of implementation does not allow for nested WHILE structures. You are encouraged to think about how such nested structures might be supported (see Exercise 4.2.14).

4.2.4 Keyword Macro Parameters

All the macro instruction definitions we have seen thus far used *positional parameters*. That is, parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement. With positional parameters, the programmer must be careful to specify the arguments in the proper order. If an argument is to be omitted, the macro invocation statement must contain a null argument (two consecutive commas) to maintain the correct argument positions. [See, for example, the macro invocation statement in Fig. 4.8(c).]

Positional parameters are quite suitable for most macro instructions. However, if a macro has a large number of parameters, and only a few of these are given values in a typical invocation, a different form of parameter specification is more useful. (Such a macro may occur in a situation in which a large and complex sequence of statements—perhaps even an entire operating system—is to be generated from a macro invocation. In such cases, most of the parameters may have acceptable default values; the macro invocation specifies only the changes from the default set of values.)

For example, suppose that a certain macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only the third and ninth parameters are to be specified. If positional parameters were used, the macro invocation statement might look like

```
GENER   ,,DIRECT,,,,,,3.
```

Using a different form of parameter specification, called *keyword parameters*, each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. If the third parameter in the previous example is named &TYPE and the ninth parameter is named &CHANNEL, the macro invocation statement would be

```
GENER   TYPE=DIRECT,CHANNEL=3.
```

This statement is obviously much easier to read, and much less error-prone, than the positional version.

Figure 4.10(a) shows a version of the RDBUFF macro definition using keyword parameters. Except for the method of specification, the parameters are the same as those in Fig. 4.8(a). In the macro prototype, each parameter name is followed by an equal sign, which identifies a keyword parameter. After the equal sign, a default value is specified for some of the parameters. The parameter is assumed to have this default value if its name does not appear in the macro invocation statement. Thus the default value for the parameter &INDEV is F1. There is no default value for the parameter &BUFADR.

Default values can simplify the macro definition in many cases. For example, the macro definitions in Figs. 4.10(a) and 4.8(a) both provide for setting the maximum record length to 4096 unless a different value is specified by the user. The default value established in Fig. 4.10(a) takes care of this automatically. In Fig. 4.8(a), an IF-ELSE-ENDIF structure is required to accomplish the same thing.

The other parts of Fig. 4.10 contain examples of the expansion of keyword macro invocation statements. In Fig. 4.10(b), all the default values are accepted. In Fig. 4.10(c), the value of &INDEV is specified as F3, and the value of &EOR is specified as null. These values override the corresponding defaults. Note that the arguments may appear in any order in the macro invocation statement. You may want to work through these macro expansions for yourself, concentrating on how the default values are handled.

4.3 MACRO PROCESSOR DESIGN OPTIONS

In this section we discuss some major design options for a macro processor. The algorithm presented in Fig. 4.5 does not work properly if a macro

```
25   RDBUFF   MACRO    &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
26            IF       (&EOR NE '')
27   &EORCK   SET      1
28            ENDIF
30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
38            IF       (&EORCK EQ 1)
40            LDCH    =X'&EOR'        SET EOR CHARACTER
42            RMO      A,S
43            ENDIF
47           +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
50   $LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
55            JEQ      $LOOP          LOOP UNTIL READY
60            RD      =X'&INDEV'      READ CHARACTER INTO REG A
63            IF       (&EORCK EQ 1)
65            COMPR    A,S            TEST FOR END OF RECORD
70            JEQ      $EXIT          EXIT LOOP IF EOR
73            ENDIF
75            STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
80            TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85            JLT      $LOOP            HAS BEEN REACHED
90   $EXIT    STX      &RECLTH        SAVE RECORD LENGTH
95            MEND
```

(a)

```
.             RDBUFF   BUFADR=BUFFER,RECLTH=LENGTH

30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
40            LDCH    =X'04'          SET EOR CHARACTER
42            RMO      A,S
47           +LDT     #4096           SET MAXIMUM RECORD LENGTH
50   $AALOOP  TD      =X'F1'          TEST INPUT DEVICE
55            JEQ      $AALOOP        LOOP UNTIL READY
60            RD      =X'F1'          READ CHARACTER INTO REG A
65            COMPR    A,S            TEST FOR END OF RECORD
70            JEQ      $AAEXIT        EXIT LOOP IF EOR
75            STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80            TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85            JLT      $AALOOP          HAS BEEN REACHED
90   $AAEXIT  STX      LENGTH         SAVE RECORD LENGTH
```

(b)

Figure 4.10 Use of keyword parameters in macro instructions.


```
.             RDBUFF   RECLTH=LENGTH,BUFADR=BUFFER,EOR=,INDEV=F3

30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
47           +LDT     #4096           SET MAXIMUM RECORD LENGTH
50   $ABLOOP  TD      =X'F3'          TEST INPUT DEVICE
55            JEQ      $ABLOOP        LOOP UNTIL READY
60            RD      =X'F3'          READ CHARACTER INTO REG A
75            STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80            TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85            JLT      $ABLOOP          HAS BEEN REACHED
90   $ABEXIT  STX      LENGTH         SAVE RECORD LENGTH
```

(c)

Figure 4.10 *(cont'd)*

invocation statement appears within the body of a macro instruction. However, it is often desirable to allow macros to be used in this way. Section 4.3.1 examines the problems created by such macro invocation statements, and suggests some possibilities for the solution of these problems.

Although the most common use of macro instructions is in connection with assembler language programming, there are other possibilities. Section 4.3.2 discusses general-purpose macro processors that are not tied to any particular language. An example of such a macro processor can be found in Section 4.4.3. Section 4.3.3 examines the other side of this issue: the integration of a macro processor with a particular assembler or compiler. We discuss the possibilities for cooperation between the macro processor and the language translator, and briefly indicate some of the potential benefits and problems of such integration.

4.3.1 Recursive Macro Expansion

In Fig. 4.3 we presented an example of the *definition* of one macro instruction by another. We have not, however, dealt with the *invocation* of one macro by another. Figure 4.11 shows an example of such a use of macros. The definition of RDBUFF in Fig. 4.11(a) is essentially the same as the one in Fig. 4.1. The order of the parameters has been changed to make the point of the example clearer. In this case, however, we have assumed that a related macro instruction (RDCHAR) already exists. The purpose of RDCHAR is to read one character from a specified device into register A, taking care of the necessary

test-and-wait loop. The definition of this macro appears in Fig. 4.11(b). It is convenient to use a macro like RDCHAR in the definition of RDBUFF so that the programmer who is defining RDBUFF need not worry about the details of device access and control. (RDCHAR might be written at a different time, or even by a different programmer.) The advantages of using RDCHAR in this way would be even greater on a more complex machine, where the code to read a single character might be longer and more complicated than our simple three-line version.

```
10   RDBUFF   MACRO    &BUFADR,&RECLTH,&INDEV
15   .
20   .        MACRO TO READ RECORD INTO BUFFER
25   .
30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
40            CLEAR    S
45           +LDT      #4096          SET MAXIMUM RECORD LENGTH
50   $LOOP    RDCHAR   &INDEV         READ CHARACTER INTO REG A
65            COMPR    A,S            TEST FOR END OF RECORD
70            JEQ      $EXIT          EXIT LOOP IF EOR
75            STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
80            TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85            JLT      $LOOP            HAS BEEN REACHED
90   $EXIT    STX      &RECLTH        SAVE RECORD LENGTH
95            MEND
```

(a)

```
 5   RDCHAR   MACRO    &IN
10   .
15   .        MACRO TO READ CHARACTER INTO REGISTER A
20   .
25            TD       =X'&IN'        TEST INPUT DEVICE
30            JEQ      *-3            LOOP UNTIL READY
35            RD       =X'&IN'        READ CHARACTER
40            MEND
```

(b)

```
              RDBUFF   BUFFER,LENGTH,F1
```

(c)

Figure 4.11 Example of nested macro invocation.

Unfortunately, the macro processor design we have discussed previously cannot handle such invocations of macros within macros. For example, suppose that the algorithm of Fig. 4.5 were applied to the macro invocation statement in Fig. 4.11(c). The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows:

Parameter	Value
1	BUFFER
2	LENGTH
3	F1
4	(unused)
.	.

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until line 50, which contains a statement invoking RDCHAR. At that point, PROCESSLINE would call EXPAND again. This time, ARGTAB would look like

Parameter	Value
1	F1
2	(unused)
.	.

The expansion of RDCHAR would also proceed normally. At the end of this expansion, however, a problem would appear. When the end of the definition of RDCHAR was recognized, EXPANDING would be set to FALSE. Thus the macro processor would "forget" that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the values in ARGTAB were overwritten with the arguments from the invocation of RDCHAR.

The cause of these difficulties is the recursive call of the procedure EXPAND. When the RDBUFF macro invocation is encountered, EXPAND is called. Later, it calls PROCESSLINE for line 50, which results in another call to EXPAND before a return is made from the original call. A similar problem would occur with PROCESSLINE since this procedure too would be called recursively. For example, there might be confusion about whether the return from PROCESSLINE should be made to the main (outermost) loop of the macro processor logic or to the loop within EXPAND.

These problems are not difficult to solve if the macro processor is being written in a programming language (such as Pascal or C) that allows recursive calls. The compiler would be sure that previous values of any variables declared within a procedure were saved when that procedure was called recursively. It would also take care of other details involving return from the procedure. (In Chapter 5 we consider in detail how such recursive calls are handled by a compiler.)

If a programming language that supports recursion is not available, the programmer must take care of handling such items as return addresses and values of local variables. In such a case, PROCESSLINE and EXPAND would probably not be procedures at all. Instead, the same logic would be incorporated into a looping structure, with data values being saved on a stack. The concepts involved are the same as those that we discuss when we consider recursion in Chapter 5. An example of such an implementation can be found in Donovan (1972).

4.3.2 General-Purpose Macro Processors

The most common use of macro processors is as an aid to assembler language programming. Often such macro processors are combined with, or closely related to, the assembler. Macro processors have also been developed for some high-level programming languages. (One example of this kind of macro processor is discussed in Section 4.4.2.) These special-purpose macro processors are similar in general function and approach; however, the details differ from language to language. In this section we discuss general-purpose macro processors. These are not dependent on any particular programming language, but can be used with a variety of different languages.

The advantages of such a general-purpose approach to macro processing are obvious. The programmer does not need to learn about a different macro facility for each compiler or assembler language, so much of the time and expense involved in training are eliminated. The costs involved in producing a general-purpose macro processor are somewhat greater than those for developing a language-specific processor. However, this expense does not need to be repeated for each language; the result is a substantial overall saving in software development cost. Similar savings in software maintenance effort should also be realized. Over a period of years, these maintenance costs may be even more significant than the original cost for software development.

In spite of the advantages noted, there are still relatively few general-purpose macro processors. One of the reasons for this situation is the large number of details that must be dealt with in a real programming language. A special-purpose macro processor can have these details built into its logic and

structure. A general-purpose facility, on the other hand, must provide some way for a user to define the specific set of rules to be followed.

In a typical programming language, there are several situations in which normal macro parameter substitution should not occur. For example, comments should usually be ignored by a macro processor (at least in scanning for parameters). However, each programming language has its own methods for identifying comments. Some languages (such as Pascal and C) use special characters to mark the start and end of a comment. Others (such as Ada) use a special character to mark only the start of a comment; the comment is automatically terminated at the end of the source line. Some languages (such as FORTRAN) use a special symbol to flag an entire line as a comment. In most assembler languages, any characters on a line following the end of the instruction operand field are automatically taken as comments. Sometimes comments are recognized partly by their position in the source line.

Another difference between programming languages is related to their facilities for grouping together terms, expressions, or statements. A general-purpose macro processor may need to take these groupings into account in scanning the source statements. Some languages use keywords such as **begin** and **end** for grouping statements. Others use special characters such as { and }. Many languages use parentheses for grouping terms and expressions. However, the rules for doing this may vary from one language to another.

A more general problem involves the *tokens* of the programming language—for example, identifiers, constants, operators, and keywords. Languages differ substantially in their restrictions on the length of identifiers and the rules for the formation of constants. Sometimes the rules for such tokens are different in certain parts of the program (for example, within a FORMAT statement in FORTRAN or a DATA DIVISION in COBOL). In some languages, there are multiple-character operators such as ** in FORTRAN and := in Pascal. Problems may arise if these are treated by a macro processor as two separate characters rather than as a single operator. Even the arrangement of the source statements in the input file may create difficulties. The macro processor must be concerned with whether or not blanks are significant, with the way statements are continued from one line to another, and with special statement formatting conventions such as those found in FORTRAN and COBOL.

Another potential problem with general-purpose macro processors involves the syntax used for macro definitions and macro invocation statements. With most special-purpose macro processors, macro invocations are very similar in form to statements in the source programming language. (For example, the invocation of RDBUFF in Fig. 4.1 has the same form as a SIC assembler language statement.) This similarity of form tends to make the source program easier to write and read. However, it is difficult to achieve with a

general-purpose macro processor that is to be used with programming languages having different basic statement forms.

In Section 4.4.3 we briefly describe one example of a general-purpose macro processor. Other discussions of general-purpose macro processors and macro processors for high-level languages can be found in Cole (1981), Kernighan and Plauger (1976), Brown (1974), and Campbell-Kelley (1973).

4.3.3 Macro Processing within Language Translators

The macro processors that we have discussed so far might be called *preprocessors*. That is, they process macro definitions and expand macro invocations, producing an expanded version of the source program. This expanded program is then used as input to an assembler or compiler. In this section we discuss an alternative: combining the macro processing functions with the language translator itself.

The simplest method of achieving this sort of combination is a *line-by-line* macro processor. Using this approach, the macro processor reads the source program statements and performs all of its functions as previously described. However, the output lines are passed to the language translator as they are generated (one at a time), instead of being written to an expanded source file. Thus the macro processor operates as a sort of input routine for the assembler or compiler.

This line-by-line approach has several advantages. It avoids making an extra pass over the source program (writing and then reading the expanded source file), so it can be more efficient than using a macro preprocessor. Some of the data structures required by the macro processor and the language translator can be combined. For example, OPTAB in an assembler and NAMTAB in the macro processor could be implemented in the same table. In addition, many utility subroutines and functions can be used by both the language translator and the macro processor. These include such operations as scanning input lines, searching tables, and converting numeric values from external to internal representations. A line-by-line macro processor also makes it easier to give diagnostic messages that are related to the source statement containing the error (i.e., the macro invocation statement). With a macro preprocessor, such an error might be detected only in relation to some statement in the macro expansion. The programmer would then need to backtrack to discover the original source of trouble.

Although a line-by-line macro processor may use some of the same utility routines as the language translator, the functions of macro processing and program translation are still relatively independent. The main form of communication between the two functions is the passing of source statements from one

to the other. It is possible to have even closer cooperation between the macro processor and the assembler or compiler. Such a scheme can be thought of as a language translator with an *integrated* macro processor.

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator. The actual degree of integration varies considerably from one system to another. At a relatively simple level of cooperation, the macro processor may use the results of such translator operations as scanning for symbols, constants, etc. Such operations must be performed by the assembler or compiler in any case; the macro processor can simply use the results without being involved in such details as multiple-character operators, continuation lines, and the rules for token formation. This is particularly useful when the rules for such details vary from one part of the program to another (for example, within FORMAT statements and character string constants in FORTRAN).

The sort of token scan just mentioned is conceptually quite simple. However, many real programming languages have certain characteristics that create unpleasant difficulties. One classic example is the FORTRAN statement

```
DO 100 I = 1,20
```

This is a DO statement: DO is recognized as a keyword, 100 as a statement number, I as a variable name, etc. However, blanks are not significant in FORTRAN statements (except within character string constants). Thus the similar statement

```
DO 100 I = 1
```

has a quite different meaning. This is an assignment statement that gives the value 1 to the variable DO100I. Thus the proper interpretation of the characters DO, 100, etc., cannot be decided until the rest of the statement is examined. Such interpretations would be very important if, for example, a macro involved substituting for the variable name I. A FORTRAN compiler must be able to recognize and handle situations such as this. However, it would be very difficult for an ordinary macro processor (not integrated with a compiler) to do so. Such a macro processor would be concerned only with character strings, not with the interpretation of source statements.

With an even closer degree of cooperation, an integrated macro processor can support macro instructions that depend upon the context in which they occur. For example, a macro could specify a substitution to be applied only to variables or constants of a certain type, or only to variables appearing as loop indices in DO statements. The expansion of a macro could also depend upon a variety of characteristics of its arguments.

There are, of course, disadvantages to integrated and line-by-line macro processors. They must be specially designed and written to work with a particular implementation of an assembler or compiler (not just with a particular programming language). The costs of macro processor development must therefore be added to the cost of the language translator, which results in a more expensive piece of software. In addition, the assembler or compiler will be considerably larger and more complex than it would be if a macro preprocessor were used. The size may be a problem if the translator is to run on a computer with limited memory. In any case, the additional complexity will add to the overhead of language translation. (For example, some assemblers with integrated macro processors consume more time per line of source code than do some compilers on the same computing system.) Decisions about what type of macro processor to use should be based on considerations such as the frequency and complexity of macro processing that is anticipated, and other characteristics of the computing environment.

4.4 IMPLEMENTATION EXAMPLES

In this section we briefly present three examples of actual macro processors. As before, we do not attempt to cover all the characteristics of each system. Instead, we focus on the more interesting or unusual features. The first example is a macro processor that is integrated with the MASM assembler (see also Section 2.5.1). The second is a macro facility that is part of the ANSI C preprocessor. The third example is a general-purpose macro processor that is not associated with any particular programming language.

4.4.1 MASM Macro Processor

This section describes some of the macro processing features of the Microsoft MASM assembler. Further information about MASM can be found in Barkakati (1992).

The macro processor of MASM is integrated with Pass 1 of the assembler. It supports all of the main macro processor functions that we have discussed, including the definition and invocation of macro instructions within macros. Macros may be redefined during a program, without causing an error. The new definition of the macro simply replaces the first one. However, this practice can be very confusing to a person reading the program—it should probably be avoided.

One of the main differences between the MASM macro processor and the one we discussed for SIC lies in the nature of the conditional macro expansion

statements. MASM calls these *conditional assembly* statements. Although the main use of these statements is in connection with macro instructions, they can also appear outside of macros.

Figure 4.12 illustrates some of the MASM macro and conditional assembly statements. The macro instruction defined in Fig. 4.12(a) computes the absolute difference between the values of its first two parameters. These parameters may be either words or doublewords. If they are doublewords, the third parameter has the value E and the calculation uses the doubleword register EAX. If the first two parameters are words, the third parameter is omitted. In that case, the calculation uses the word-length portion of the register, which is designated by AX.

The MACRO header on line 1 gives the name of the macro and its parameters. Notice that macro parameters in MASM need not begin with & or any other special character. The end of the macro is marked by the ENDM on line 15.

Line 2 declares that EXIT is a local label. When the macro is expanded, each local label is replaced by a unique name. MASM generates these unique names in the form *??n*, where *n* is a hexadecimal number in the range 0000 to FFFF. See the macro expansions in Fig. 4.12(b) and (c) for an example of this.

The IFNB on line 3 evaluates to "true" if its operand is not blank. If the parameter SIZE is not blank (that is, if it is present in the macro invocation), lines 4 through 8 are processed. Otherwise, these lines are skipped. Lines 4 through 8 contain a nested conditional statement. The IFDIF on line 4 is true if the string represented by SIZE is different from the string E. In that case, lines 5 through 7 are processed. Line 5 generates a comment that will appear on the assembly listing. The .ERR on line 6 signals to MASM that an error has been detected, and the EXITM directs MASM to terminate the expansion of the macro. Figure 4.12(d) illustrates the result.

The & on line 10 is a concatenation operator, used to combine the value of the parameter SIZE with the string AX. If SIZE has the value E, the result is EAX. If SIZE is blank, the result is simply AX. [Compare the macro expansions in Fig. 4.12(b) and (c).]

Notice the difference between the comments on lines 9 and 10. The comment on line 9, which begins with ;; , is a *macro comment*. It serves only as documentation for the macro definition; it is ignored when the macro is expended. The comment on line 10, which begins with ; , is an ordinary assembler language comment. It is included as part of the macro expansion.

Figure 4.13(a) illustrates one of the iteration statements available in MASM. The IRP on line 2 sets the macro-time variable S to a sequence of values specified in < ... >. The statements between the IRP and the matching ENDM on line 4 are generated once for each such value of S. Figure 4.13(b) shows an example of the resulting macro expansion.

```
ABSDIF   MACRO    OP1,OP2,SIZE
         LOCAL    EXIT
         IFNB     <SIZE>        ;; IF SIZE IS NOT BLANK
         IFDIF    <SIZE>,<E>    ;;   THEN IT MUST BE E
         ; ERROR -- SIZE MUST BE E OR BLANK
         .ERR
         EXITM
         ENDIF                  ;; END OF IFDIF
         ENDIF                  ;; END OF IFNB
         MOV      SIZE&AX,OP1   ; COMPUTE ABSOLUTE DIFFERENCE
         SUB      SIZE&AX,OP2   ;; SUBTRACT OP2 FROM OP1
         JNS      EXIT          ;; EXIT IF RESULT GE 0
         NEG      SIZE&AX       ;;   OTHERWISE CHANGE SIGN
EXIT:
         ENDM
```

(a)

```
ABSDIF   J,K
```

↓

```
         MOV      AX,J          ; COMPUTE ABSOLUTE DIFFERENCE
         SUB      AX,K
         JNS      ??0000
         NEG      AX
??0000:
```

(b)

```
ABSDIF   M,N,E
```

↓

```
         MOV      EAX,M         ; COMPUTE ABSOLUTE DIFFERENCE
         SUB      EAX,N
         JNS      ??0001
         NEG      EAX
??0001:
```

(c)

```
ABSDIF   P,Q,X
```

↓

```
; ERROR -- SIZE MUST BE E OR BLANK
```

(d)

Figure 4.12 Examples of MASM macro and conditional statements.


```
NODE     MACRO    NAME
         IRP      S,<'LEFT','DATA','RIGHT'>
NAME&S   DW       0
         ENDM                    ;; END OF IRP
         ENDM                    ;; END OF MACRO
```

(a)

```
         NODE     X
```

↓

```
XLEFT    DW       0
XDATA    DW       0
XRIGHT   DW       0
```

(b)

Figure 4.13 Example of MASM iteration statement.

4.4.2 ANSI C Macro Language

This section describes some of the macro processing features of the ANSI C programming language. Section 5.5.1 discusses the structure of a typical compiler and preprocessor that implement these features. Further information can be found in Schildt (1990), as well as in many C language reference books.

In the ANSI C language, definitions and invocations of macros are handled by a preprocessor. This preprocessor is generally not integrated with the rest of the compiler. Its operation is similar to the macro processor we discussed in Section 4.1. The preprocessor also performs a number of other functions, some of which are discussed in Section 5.5.1.

Here are two simple (and commonly used) examples of ANSI C macro definitions.

```
#define  NULL    0
#define  EOF     (-1)
```

After these definitions appear in the program, every occurrence of NULL will be replaced by 0, and every occurrence of EOF will be replaced by (-1). It is also possible to use macros like this to make limited changes in the syntax of the language. For example, after defining the macro

```
#define  EQ      ==
```

a programmer could write

```
while (I EQ 0)...
```

The macro processor would convert this into

```
while (I == 0)...
```

which is the correct C syntax. This could help avoid the common C error of writing = in place of ==. (However, many people consider such syntactic modifications to be a poor programming practice.)

ANSI C macros can also be defined with parameters. Consider, for example, the macro definition

```
#define  ABSDIFF(X,Y)  ((X) > (Y) ? (X) - (Y) : (Y) - (X))
```

In this case, the macro name is ABSDIFF; the parameters are named X and Y. The body of the macro makes use of a special C language conditional expression. If the condition (X) > (Y) is true, the value of this expression is the first alternative specified, (X) – (Y). If the condition is false, the value of the expression is the second alternative, (Y) – (X).

A macro invocation consists of the name of the macro followed by a parenthesized list of parameters separated by commas. When the macro is expanded, each occurrence of a macro parameter is replaced by the corresponding argument. For example,

```
ABSDIFF(I+1,J-5)
```

would be converted by the macro processor into

```
((I+1) > (J-5) ? (I+1) - (J-5) : (J-5) - (I+1))
```

Notice the similarity between this macro invocation and a function call. Clearly, we could write a function ABSDIFF to perform this same operation. However, the macro is more efficient, because the amount of computation required to compute the absolute difference is quite small—much less than the overhead of calling a function. The macro version can also be used with different types of data. For example, we could invoke the macro as

```
ABSDIFF(I, 3.14159)
```


or

```
ABSDIFF('D', 'A')
```

Because C macros are handled by a preprocessor, it is necessary to be very careful in writing macro definitions with parameters. The macro processor simply makes string substitutions, without considering the syntax of the C language. For example, if we had written the definition of ABSDIFF as

```
#define  ABSDIFF(X,Y)  X > Y ? X - Y : Y - X
```

the macro invocation

```
ABSDIFF(3 + 1,10 - 8)
```

would be expanded into

```
3 + 1 > 10 - 8 ? 3 + 1 - 10 - 8 : 10 - 8 - 3 + 1
```

which would not produce the intended result. (The first alternative in this case has the value –14 instead of 2, as it should be.)

In ANSI C, parameter substitutions are not performed within quoted strings. For example, consider the macro definition

```
#define  DISPLAY(EXPR)  printf("EXPR = %d\n", EXPR)
```

The macro invocation

```
DISPLAY(I*J+1)
```

would be expanded into

```
printf("EXPR = %d\n", I*J+1)
```

(However, some C compilers *would* perform the substitution for EXPR inside the quoted string, possibly with a warning message to the programmer.)

To avoid this problem, ANSI C provides a special "stringizing" operator #. When the name of a macro parameter is preceded by #, argument substitution is performed in the usual way. After the substitution, however, the resulting string is enclosed in quotes. For example, if we define

```
#define  DISPLAY(EXPR)  printf(#EXPR "= %d\n", EXPR)
```

then the invocation

```
DISPLAY(I*J+1)
```

would be expanded into

```
printf("I*J+1" "= %d\n", EXPR)
```

Macros in ANSI C may contain definitions or invocations of other macros. After a macro is expanded, the macro processor rescans the text that has been generated, looking for more macro definitions or invocations. For example, the invocation

```
DISPLAY(ABSDIFF(3,8))
```

would be expanded into

```
printf("ABSDIFF(3,8)" "= %d\n", ABSDIFF(3,8))
```

After rescanning, this would become

```
printf("ABSDIFF(3,8)" "= %d\n",
                ((3) > (8) ? (3) - (8) : (8) - (3)))
```

(Notice that the ABSDIFF within the quoted string is not treated as a macro invocation.) When executed, this statement would produce the output

```
ABSDIFF(3,8) = 5
```

The rescanning process behaves somewhat differently from the macro processing we discussed earlier in this chapter. If the body of a macro contains a token that happens to match the name of the macro, this token is not replaced during rescanning. Thus a macro cannot invoke (or define) itself recursively.

The ANSI C preprocessor also provides conditional compilation statements. These statements can be used to be sure that a macro (or other name) is defined at least once. For example, in the sequence

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1024
#endif
```


the #define will be processed only if BUFFER_SIZE has not already been defined.

Conditionals are also often used to control the inclusion of debugging statements in a program. Consider, for example, the sequence

```
#define DEBUG 1
.
.
.
#if DEBUG == 1
printf(...)     /* debugging output */
#endif
```

In this case, the printf statement will be included in the output from the preprocessor (and therefore compiled into the program). If the first line were changed to

```
#define DEBUG 0
```

the printf would not be included in the program. The same thing could also be accomplished by writing

```
#ifdef DEBUG
printf(...)     /* debugging output */
#endif
```

In this case, the printf would be included if a #define statement for DEBUG appeared in the source program.

4.4.3 The ELENA Macro Processor*

This section describes some of the features of the ELENA general-purpose macro processor. ELENA was developed as a research tool, not as a commercial software product. However, the same design and implementation techniques could be used in developing other general-purpose macro processors. Further information about ELENA can be found in Barcucci and Pelacani (1984).

Macro definitions in ELENA are composed of a header and a body, as with most macro processors. However, the header is not required to have any spe-

*Adapted from "A Software Development System Based on a Macroprocessor" by Elena Barcucci and Gianluca Pelacani, from Software: *Practice and Experience*, Vol. 14, pp. 519-531 (June 1984). ©John Wiley & Sons Ltd.

cial form. It consists of a sequence of keywords and parameter markers (which are identified by the special character %). The only restriction is that at least one of the first two tokens in a macro header must be a keyword, not a parameter marker. A macro invocation is a sequence of tokens that matches the macro header. For example, a macro with the header

```
%1 = %2 + %3
```

could be invoked as

```
ALPHA = BETA + GAMMA
```

and a macro with header

```
ADD %1 TO THE VALUE OF %2
```

could be invoked as

```
ADD 10 TO THE VALUE OF INDEX
```

Figure 4.14 illustrates how ELENA could be used with different languages. Consider the macro header shown in Fig. 4.14(a). If this macro is to be used with the C language, its body might be defined as shown in Fig. 4.14(b). An example of a macro invocation and expansion using this body appears in Fig. 4.14(c).

On the other hand, suppose that the macro is to be used in an x86 assembler language program. In that case, the body might be defined as shown in Fig. 4.14(d). An example of a macro invocation and expansion using this body appears in Fig. 4.14(e). Notice that in this expansion the label &STOR is changed to &STOR0001. The character & identifies &STOR as a local label within the macro definition. The macro processor appends a numeric value to create unique labels each time the macro is expanded.

ELENA also provides macro-time variables and macro-time instructions that can be used to control the macro expansion. Consider the macro header shown in Fig. 4.15(a) and the associated body in Fig. 4.15(b). The .SET statement on the first line of the macro body sets the macro-time variable .LAA to 1. The next line is a statement to be generated as part of the macro expansion. After this line is generated, the following .SET statement adds 1 to the value of .LAA. If this new value is less than or equal to the value of the second parameter, the .IF macro-time instruction causes the macro processor to jump back to the line with the macro-time label .E. Figure 4.15(c) shows an example of a macro invocation and expansion using this body.

The macro-time instructions in ELENA represent a different type of approach to conditional macro expansion. The .IF statement in Fig. 4.15(b) is a macro-time conditional "go to" statement. In the SIC macro language, we would have written this definition using the WHILE-ENDW structure, which is a higher-level macro-time instruction.

```
%1 := ABSDIFF(%2,%3)
```

(a)

```
%1 = (%2) > (%3) ? (%2) - (%3) : (%3) - (%2)
```

(b)

```
Z := ABSDIFF(X,Y)
```

↓

```
Z = (X) > (Y) ? (X) - (Y) : (Y) - (X)
```

(c)

```
         MOV     EAX,%2
         SUB     EAX,%3
         JNS     &STOR
         NEG     EAX
&STOR    MOV     EAX,%1
```

(d)

```
Z := ABSDIFF(X,Y)
```

↓

```
         MOV     EAX,X
         SUB     EAX,Y
         JNS     STOR0001
         NEG     EAX
STOR0001 MOV     EAX,Z
```

(e)

Figure 4.14 Examples of ELENA macro definition and invocation.

```
ADD %1 TO THE FIRST %2 ELEMENTS OF V
```

(a)

```
        .SET .LAA = 1
.E      V(.LAA) = V(.LAA) + %1
        .SET .LAA = .LAA + 1
        .IF .LAA LE %2 .JUMP .E
```

(b)

```
ADD 5 TO THE FIRST 3 ELEMENTS OF V
```

↓

```
V(1) = V(1) + 5
V(2) = V(2) + 5
V(3) = V(3) + 5
```

(c)

Figure 4.15 Example of ELENA macro-time instructions.

The ELENA macro processor uses a macro definition table that is similar to the one we discussed for SIC. However, the process of matching a macro invocation with a macro header is more complicated. Notice that there is no single token that constitutes the macro "name." Instead, the macro is identified by the sequence of keywords that appear in its header. Consider, for example, the two macro headers

```
ADD %1 TO %2
ADD %1 TO THE FIRST ELEMENT OF %2
```

Furthermore, it is not even clear from a macro invocation statement which tokens are keywords and which are parameters. A sequence of tokens like

```
DISPLAY TABLE
```


could be an invocation of a macro with header

```
DISPLAY %1
```

(where the parameter specifies what to display). On the other hand, it could also be an invocation of a macro with header

```
%1 TABLE
```

(where the parameter specifies what operation to perform on TABLE). Notice that it is not possible for both DISPLAY and TABLE to be parameters, because at least one of the first two tokens in a macro header is required to be a keyword.

ELENA deals with this problem by constructing an index of all macro headers according to the keywords in the first two tokens of the header. Potential macro invocations are compared against all headers with keywords that match at least one of their first two tokens. For example, the sequence

```
A SUM B,C
```

would be compared against all macro headers in which the first token is A or the second token is SUM.

During this process, it is possible that a sequence of tokens could match more than one macro header. For example, the sequence

```
A = B + 1
```

might match both of the headers

```
%1 = %2 + %3
```

and

```
%1 = %2 + 1
```

In this situation, ELENA selects the header with the fewest parameters (i.e., the second of the two headers just mentioned). If there are two or more matching headers with the same number of parameters, the most recently defined macro is selected.

EXERCISES

Section 4.1

1. Apply the algorithm in Fig. 4.5 to process the source program in Fig. 4.1; the results should be the same as shown in Fig. 4.2.
2. Using the methods outlined in Chapter 8, develop a modular design for a one-pass macro processor.
3. Macro invocation statements are a part of the source program. In many cases, the programmer may not be concerned with the statements in the macro expansion. How could the macro processor and assembler cooperate to list only the macro invocation, and not the expanded version?
4. Suppose we want macro definitions to appear as a part of the assembly listing. How could the macro processor and the assembler accomplish this?
5. In most cases, character strings that occur in comments should not be replaced by macro arguments, even if they happen to match a macro parameter. How could parameter substitution in comments be prevented?
6. How should a programmer decide whether to use a macro or a subroutine to accomplish a given logical function?
7. Suppose that a certain logical task must be performed at 10 different places in an assembler language program. This task could be implemented either as a macro or as a subroutine. Describe a situation where using a macro would take *less* central memory than using a subroutine.
8. Some macros simply expand into instructions that call a subroutine. What are the advantages of this approach, as compared to using either a "pure" macro or a "pure" subroutine?
9. Write an algorithm for a two-pass macro processor in which all macro definitions are processed in the first pass, and all macro invocations are expanded in the second pass. You do not need to allow for macro definitions or invocations within macros.
10. Modify the algorithm in Fig. 4.5 to allow macro definitions to be retrieved from a library if they are not specified by the programmer.

11. Suggest appropriate ways of organizing and accessing the tables DEFTAB and NAMTAB.
12. Suppose that the occurrences of macro parameters in DEFTAB were not replaced by the positional notation *?n*. What changes would be required in the macro processor algorithm of Fig. 4.5?
13. Suppose that we have a two-pass macroassembler—that is, a one-pass macro processor built into Pass 1 of a two-pass assembler. The macro processor uses the general scheme described in Section 4.1.
    a. Suppose that we want the programmer to be able to invoke macros without having to include the macro definitions in the source program. For example, a programmer might simply write a RDBUFF statement in his or her program—the macro processor would automatically retrieve the definition of RDBUFF from a standard macro library. Briefly describe how this feature could be implemented efficiently.
    b. Suppose that macros are *not* allowed to have the same name as machine instructions. For example, if the programmer tries to define a macro named CLEAR, the macroassembler should give an error message (because CLEAR is a SIC/XE machine instruction). Describe how this restriction could be implemented.
    c. Now suppose that we *do* want to allow macros to have the same name as machine instructions. For example, if the programmer defines a macro named CLEAR, then any CLEAR statements in the program being assembled should be expanded as macro invocations. If the programmer does not define a macro named CLEAR, then any CLEAR statements should be assembled as machine instructions. Describe how this could be implemented.
    d. Suppose, as in part (c), that macros are allowed to have the same name as machine instructions. Suppose that we want to have the definitions of such macros retrieved automatically from a macro library, as in part (a). What problems arise? How might you solve these problems?

Section 4.2

1. The macro definitions in Fig. 4.1 contain several statements in which macro parameters are concatenated with other characters (for example, lines 50 and 75). Why was it not necessary to use concatenation operators in these statements?

2. Modify the algorithm in Fig. 4.5 to include the handling of concatenation operators.

3. Modify the algorithm in Fig. 4.5 to include the generation of unique labels within macro expansions.

4. Suppose that we want to allow labels within macro expansions without requiring them to have any special form (such as beginning with $). Each such label would be considered to be defined only within the macro expansion in which it occurs; this would eliminate the problem caused by duplicate labels. How could the macro processor and the assembler work together to allow this?

5. What is the most important difference between the following two sequences of statements?

a.
```
        LDA     ALPHA
        COMP    #0
        JEQ     SKIP
        LDA     #3
        STA     BETA
SKIP    ...
```

b.
```
        IF      (&ALPHA NE 0)
&BETA   SET     3
        ENDIF
```

6. Expand the following macro invocation statements, using the macro definition in Fig. 4.8(a):

a.
```
        RDBUFF  F1,BUFFER,LENGTH,00,1024
```

b.
```
LOOP    RDBUFF  F2,BUFFER,LTH
```

7. Suppose that you have a simple one-pass macro processor like the one described in Section 4.1. Now you want to add a conditional macro expansion statement IFDEF to the macro processor. The following example illustrates the use of IFDEF:

```
IFDEF   ALPHA
.
.
.
ENDIF
```


The statements between IFDEF and ENDIF are to be generated as part of the macro expansion if (and only if) the label ALPHA is defined somewhere in the assembly language program being processed. (It is not necessary that the definition of ALPHA appear before the macro invocation whose expansion contains the IFDEF.) Notice that ALPHA is an ordinary label, not a macro-time variable.

What changes would you have to make to the macro processor in order to implement IFDEF? Explain how your new macro processor would handle the IFDEF... ENDIF conditional statements.

8. Suppose that you have a simple one-pass macro processor like the one described in Section 4.1. Now you want to add a built-in function named %SIZEOF to the macro processor. This function can be applied to macro parameters, and returns the number of bytes occupied by the corresponding argument. Consider, for example, the following program:

```
P8      START   0
MOVE    MACRO   &FROM,&TO
&LENGTH SET     %SIZEOF(&FROM)
        IF      (&LENGTH EQ 1)
        LDCH    &FROM
        STCH    &TO
        ELSE
        LDX     #&LENGTH
        LDS     #FROM
        LDT     #TO
        JSUB    MOVERTN
        ENDIF
        MEND
FIRST   MOVE    A,B
        MOVE    C,D
        RSUB
A       RESB    1
B       RESB    1
C       RESB    500
D       RESB    500
        END
```

In the first invocation of MOVE, %SIZEOF(A) returns 1; in the second, %SIZEOF(C) returns 500. Thus the macro invocations would be expanded as follows:

```
MOVE    A,B          MOVE    C,D
  ↓                    ↓
LDCH    A            LDX     #500
STCH    B            LDS     #C
                     LDT     #D
                     JSUB    MOVERTN
```

What changes would you have to make in the macro processor to implement the function %SIZEOF? Explain how your new macro processor would work when processing the program shown above.

9. Modify the algorithm in Fig. 4.5 to include SET statements and the IF-ELSE-ENDIF structure. You do not need to allow for nested IFs.

10. Modify your answer to Exercise 9 to allow nested IFs.

11. What is the most important difference between the following two control structures?

a.
```
        LDT     #8
        CLEAR   X
LOOP    .
        .
        .
        TIXR    T
        JLT     LOOP
```

b.
```
&CTR    SET     0
        WHILE   (&CTR LT 8)
        .
        .
        .
&CTR    SET     &CTR+1
        ENDW
```

12. Using the definition in Fig. 4.9(a), expand the following macro invocation statements:

```
a.          RDBUFF  F1,BUFFER,LENGTH,(04,12)
b.  LABEL   RDBUFF  F1,BUFFER,LENGTH,00
c.          RDBUFF  F1,BUFFER,LENGTH
```


What value should the function %NITEMS(&EOR) return in the last two cases?

13. Modify your answer to Exercise 9 to include WHILE statements. You do not need to allow for nested WHILEs.

14. Modify your answer to Exercise 13 to allow nested WHILEs.

15. The values of macro-time variables are usually considered to be local to a macro definition. That is, a value assigned to a macro-time variable can be used only within the same macro definition. Sometimes, however, it might be useful to be able to communicate the value of a macro-time variable between two related macros. How could this be accomplished?

16. Modify the algorithm in Fig. 4.5 to include keyword parameters.

17. Some macro processors allow macro instructions in which some of the parameters are keyword parameters and some are positional parameters. How could a macro processor handle such mixed-mode macro instructions?

18. How could default values be specified for positional parameters? What changes in the algorithm of Fig. 4.5 would be necessary to handle such defaults?

19. Refer to the definition of RDBUFF that appears in Fig. 4.8(a). Each of the following macro invocation statements contains an error. Which of these errors would be detected by the macro processor, and which would be detected by the assembler?

```
a.  RDBUFF  F3,BUF,RECL,ZZ
                {illegal value specified for &EOR}
b.  RDBUFF  F3,BUF,RECL,04,2048,01
                {too many arguments}
c.  RDBUFF  F3,,RECL,04
                {no value specified for &BUFADR}
d.  RDBUFF  F3,RECL,BUF
                {arguments specified in wrong order}
```

Section 4.3

1. Suppose that a macro processor with logic similar to that in Fig. 4.5 is to perform recursive macro expansion. The text points out the need to save values of EXPANDING and ARGTAB when making a recursive call to EXPAND. Depending upon how the algorithm is implemented, what other values might it also be necessary to save?
2. How could a recursive macro processor be implemented in assembler language?
3. How could a *nonrecursive* macro processor allow for the invocation of macros within macros? What would be the advantages and disadvantages of such an approach?
4. Select two different high-level programming languages with which you are familiar. What differences between these languages might be significant to a macro processor that is intended for use with the language?
5. Select one high-level language and one (real) assembler language with which you are familiar. What differences between these languages might be significant to a macro processor that is intended for use with the language?
6. Outline an algorithm for combining a line-by-line macro processor with an assembler.
7. List utility functions and routines that might be shared by an assembler and an integrated macro processor.
8. Using the methods outlined in Chapter 8, develop a modular design for a two-pass assembler with an integrated macro processor.

Chapter 5

Compilers

In this chapter we discuss the design and operation of compilers for high-level programming languages. Many textbooks and courses are entirely devoted to compiler construction. We obviously cannot hope to cover the subject thoroughly in a single chapter. Instead, our goal is to give the reader an understanding of how a typical compiler works. We introduce the most important concepts and issues related to compilers, and illustrate them with examples. As each subject is discussed, we give references for those readers who want to explore the topic in more detail.

Section 5.1 presents the basic functions of a simple one-pass compiler. We illustrate the operation of such a compiler by following an example program through the entire translation process. This section contains somewhat more detail than the other parts of the chapter because of the fundamental importance of the material.

Section 5.2 discusses machine-dependent extensions to the basic scheme presented in Section 5.1. These extensions are mainly in the area of object code generation and optimization. Section 5.3 describes some machine-independent extensions to the basic scheme.

Section 5.4 describes some compiler design alternatives. These include multi-pass compilers, interpreters, P-code compilers, and compiler-compilers. Finally, Section 5.5 presents five examples of actual compilers and compiler-writing systems, and relates them to the concepts introduced in previous sections.

5.1 BASIC COMPILER FUNCTIONS

This section introduces the fundamental operations that are necessary in compiling a typical high-level language program. We use as an example the Pascal program in Fig. 5.1; however, the concepts and approaches that we discuss can also be applied to the compilation of programs in other languages.

For the purposes of compiler construction, a high-level programming language is usually described in terms of a *grammar*. This grammar specifies the