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. Section 3.5.3 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).

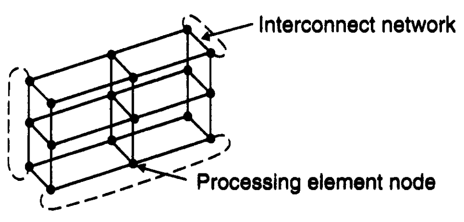


Figure 1.8 Overall T3E architecture.

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 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 SIC instructions to swap the values of ALPHA and BETA.
4. 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).
5. 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.
6. 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.
7. Write a sequence of instructions for SIC to clear a 20-byte string to all blanks.
8. 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.
9. 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.
10. 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.

11. Suppose that ALPHA is an array of 100 words. Write a sequence of instruction for SIC/XE to arrange the 100 words in ascending order and store result in an array BETA of 100 words.
12. Suppose that ALPHA and BETA are the two arrays of 100 words. Another array of GAMMA elements are obtained by multiplying the corresponding ALPHA element by 4 and adding the corresponding BETA elements.
13. Suppose that ALPHA is an array of 100 words. Write a sequence of instructions for SIC/XE to find the maximum element in the array and store results in MAX.
14. 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.
15. 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.
16. 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.
17. 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

```
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.

buffer and subroutine WRREC to write the record from the buffer to the output 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     0101  0100
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.

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.

```
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.

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 (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

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.

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.

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
```

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

```
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.

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 0001A. 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.

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.

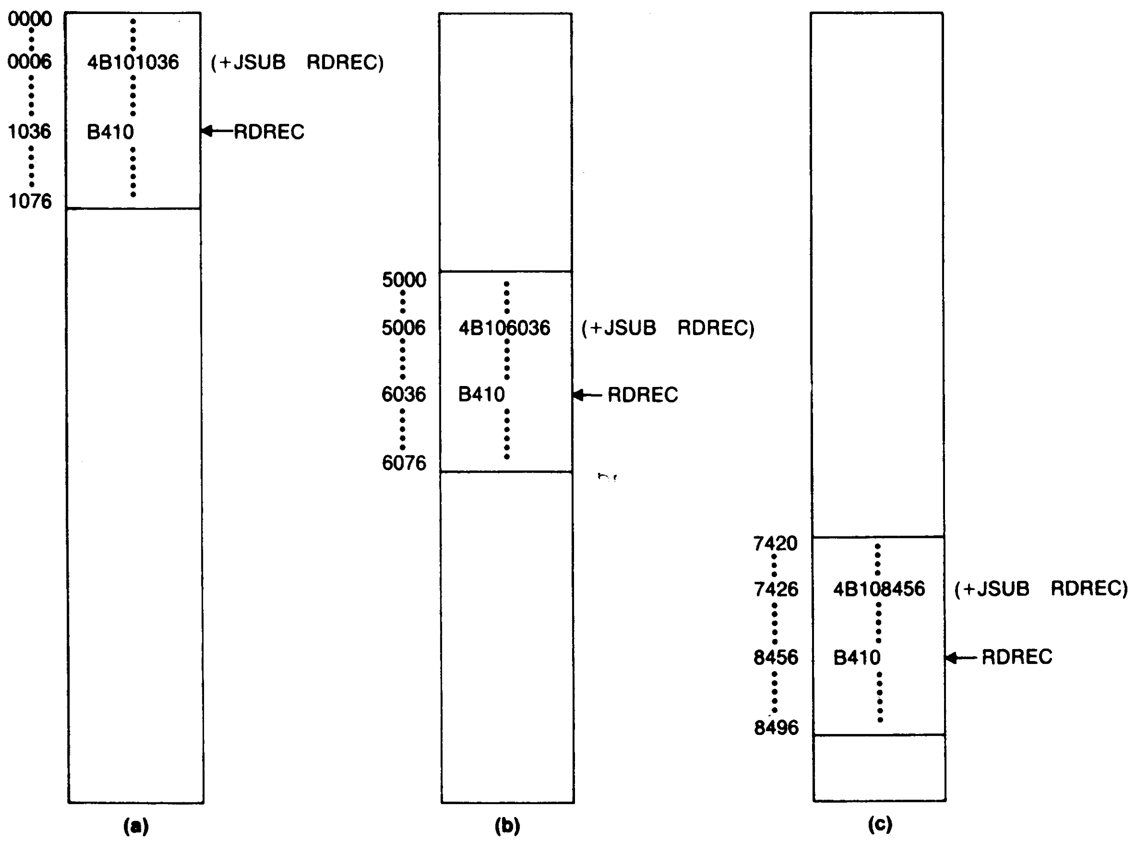


Figure 2.7 Examples of program relocation.

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

```
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.

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.

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
```

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

Line **Source statement**

```
COPY     START   0              COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR         SAVE RETURN ADDRESS
         LDB     #LENGTH        ESTABLISH BASE REGISTER
         BASE    LENGTH
CLOOP   +JSUB    RDREC          READ INPUT RECORD
         LDA     LENGTH         TEST FOR EOF (LENGTH = 0)
         COMP    #0
         JEQ     ENDFIL         EXIT IF EOF FOUND
        +JSUB    WRREC          WRITE OUTPUT RECORD
         J       CLOOP          LOOP
ENDFIL   LDA     =C'EOF'        INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     #3             SET LENGTH = 3
         STA     LENGTH
        +JSUB    WRREC          WRITE EOF
         J       @RETADR        RETURN TO CALLER
         LTORG
RETADR   RESW    1
LENGTH   RESW    1              LENGTH OF RECORD
BUFFER   RESB    4096           4096-BYTE BUFFER AREA
BUFEND   EQU     *
MAXLEN   EQU     BUFEND-BUFFER  MAXIMUM RECORD LENGTH
.
.        SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC    CLEAR   X              CLEAR LOOP COUNTER
         CLEAR   A              CLEAR A TO ZERO
         CLEAR   S              CLEAR S TO ZERO
        +LDT     #MAXLEN
RLOOP    TD      INPUT          TEST INPUT DEVICE
         JEQ     RLOOP          LOOP UNTIL READY
         RD      INPUT          READ CHARACTER INTO REGISTER A
         COMPR   A,S            TEST FOR END OF RECORD (X'00')
         JEQ     EXIT           EXIT LOOP IF EOR
         STCH    BUFFER,X       STORE CHARACTER IN BUFFER
         TIXR    T              LOOP UNLESS MAX LENGTH
         JLT     RLOOP            HAS BEEN REACHED
EXIT     STX     LENGTH         SAVE RECORD LENGTH
         RSUB                   RETURN TO CALLER
INPUT    BYTE    X'F1'          CODE FOR INPUT DEVICE
.
.        SUBROUTINE TO WRITE RECORD FROM BUFFER
.
WRREC    CLEAR   X              CLEAR LOOP COUNTER
         LDT     LENGTH
WLOOP    TD      =X'05'         TEST OUTPUT DEVICE
         JEQ     WLOOP          LOOP UNTIL READY
         LDCH    BUFFER,X       GET CHARACTER FROM BUFFER
         WD      =X'05'         WRITE CHARACTER
         TIXR    T              LOOP UNTIL ALL CHARACTERS
         JLT     WLOOP            HAVE BEEN WRITTEN
         RSUB                   RETURN TO CALLER
         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.

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
```

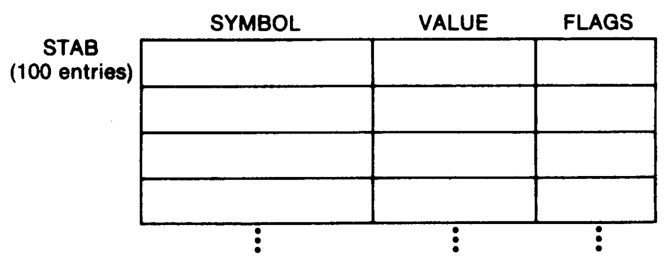
the programmer can establish and use names that reflect the logical function of the registers in the program.

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 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

```
Line     Source statement

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

Figure 2.11 Example of a program with multiple program blocks.