Contents

Chapter 1

Background

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

1.1 INTRODUCTION

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

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

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

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

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

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

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

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

as background for students who will later take more advanced software courses. This same approach is also applied to the other topics surveyed in Chapter 7.

1.2 SYSTEM SOFTWARE AND MACHINE ARCHITECTURE

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

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

Because most system software is machine-dependent, we must include real machines and real pieces of software in our study. However, most real computers have certain characteristics that are unusual or even unique. It can be difficult to distinguish between those features of the software that are truly fundamental and those that depend solely on the idiosyncrasies of a particular machine. To avoid this problem, we present the fundamental functions of each piece of software through discussion of a Simplified Instructional Computer (SIC). SIC is a hypothetical computer that has been carefully designed to include the hardware features most often found on real machines, while avoiding unusual or irrelevant complexities. In this way, the central concepts of a piece of system software can be clearly separated from the implementation details associated with a particular machine. This approach provides the

reader with a starting point from which to begin the design of system software for a new or unfamiliar computer.

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

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

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

1.3 THE SIMPLIFIED INSTRUCTIONAL COMPUTER (SIC)

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

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

1.3.1 SIC Machine Architecture

Memory

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

Registers

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

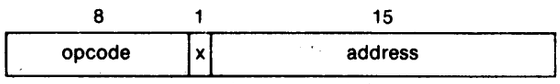
Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched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

Data Formats

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

Instruction Formats

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



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

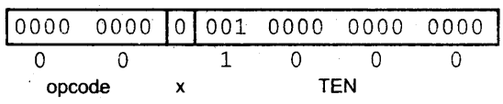
Addressing Modes

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

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

Direct addressing mode

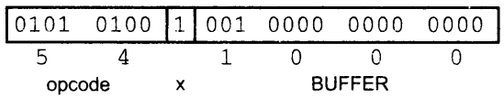
Example. LDA TEN



Effection address (EA) = 1000
Content of the address 1000 is loaded to Accumulator.

Indexed addressing mode

Example. STCH BUFFER, X



Effective address (EA) = 1000 + [X]
= 1000 + conent of the index register X
The Accumulator content, the character is loaded to the Effective address.

Instruction Set

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

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

Input and Output

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

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

1.3.2 SIC/XE Machine Architecture

Memory

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

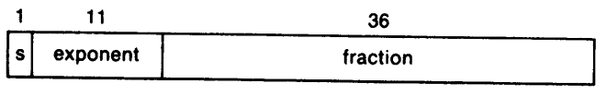
Registers

The following additional registers are provided by SIC/XE:

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

Data Formats

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



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

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

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

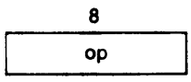
Instruction Formats

The larger memory available on SIC/XE means that an address will (in general) no longer fit into a 15-bit field; thus the instruction format used on the standard version of SIC is no longer suitable. There are two possible options—either use some form of relative addressing, or extend the address field to 20 bits. Both of these options are included in SIC/XE (Formats 3 and 4 in the following

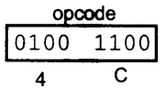
description). In addition, SIC/XE provides some instructions that do not reference memory at all. Formats 1 and 2 in the following description are used for such instructions.

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

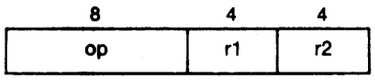
Format 1 (1 byte):



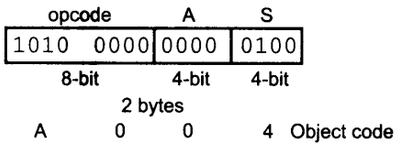
Example. RSUB (Return to subroutine)



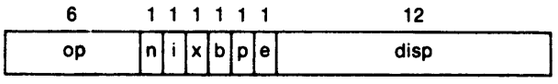
Format 2 (2 bytes):



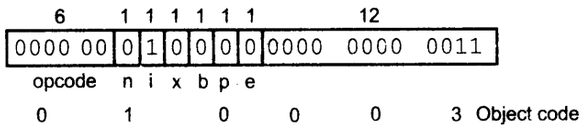
Example. COMPR A, S (Compare the contents of registers A & S)



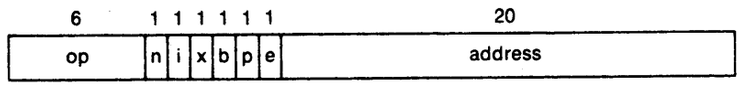
Format 3 (3 bytes):



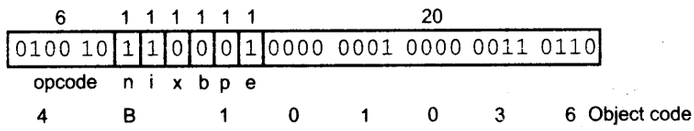
Example. LDA #3 (Load 3 to Accumlator A)



Format 4 (4 bytes):



Example. +JSUB RDREC (Jump to the address, 1036)

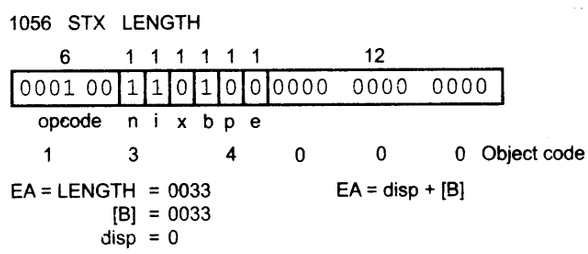


Addressing Modes

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

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

For *base relative* addressing, the displacement field *disp* in a Format 3 instruction is interpreted as a 12-bit unsigned integer.



The content of the address 0033 is loaded to the index register X.

For *program-counter relative* addressing, this field is interpreted as a 12-bit signed integer, with negative values represented in 2's complement notation.

```
0000  STL  RETADR
      6     1 1 1 1 1 1          12
   0001 01 1 1 0 0 1 0 0000  0010  1101
     opcode  n i x b p e
     1   7       2        0      2      D  Object code
EA = RETADR = 0030                EA = (PC) + disp
          PC = 0003 (address of the next instruction)
        disp = 002D
```

Linkage register contains the content of RETADR 0030.

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

```
LDA  LENGTH
      6     1 1 1 1 1 1          12
   0000 00 1 1 0 0 0 0 0000  0011  0011
     opcode  n i x b p e
     0       3       0      0      3      3
```

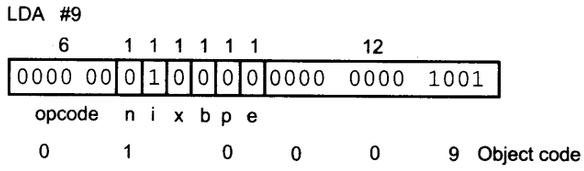
Accumulator contains the content of LENGTH 0033.

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

```
STCH  BUFFER, X
      6     1 1 1 1 1 1          12
   0101 01 1 1 1 1 0 0 0000  0000  0011
     opcode  n i x b p e
     5       7     C        0      0      3
BUFFER  = 0036
[B]     = 0033
disp    = 3
```

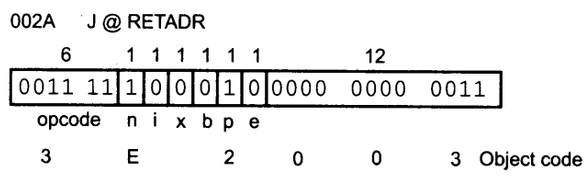
Accumulator A contains the content of BUFFER 0036.

Bits *i* and *n* in Formats 3 and 4 are used to specify how the target address is used. If bit $i = 1$ and $n = 0$, the target address itself is used as the operand value; no memory reference is performed. This is called *immediate* addressing.



Accumulator contains 9.

If bit $i = 0$ and $n = 1$, the word at the location given by the target address is fetched; the *value* contained in this word is then taken as the *address* of the operand value. This is called *indirect* addressing.



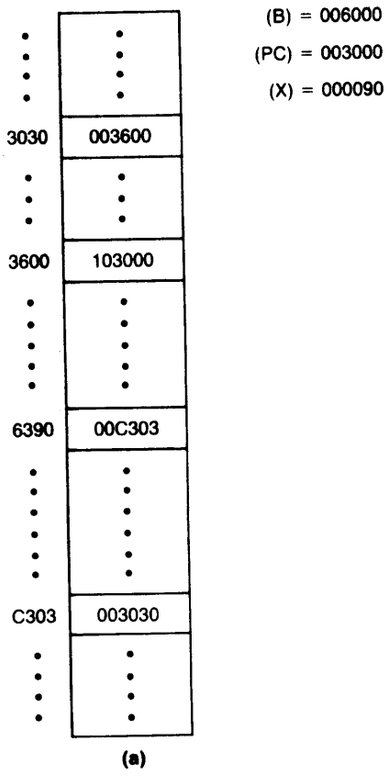
Jump to content of the address 0030 RETADR.

If bits i and n are both 0 or both 1, the target address is taken as the location of the operand; we will refer to this as *simple* addressing. Indexing cannot be used with immediate or indirect addressing modes.

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

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

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



(a)

Machine instruction										
Hex	Binary									
	op	n	i	x	b	p	e	disp/address	Target address	Value loaded into register A
032600	000000	1	1	0	0	1	0	0110 0000 0000	3600	103000
03C300	000000	1	1	1	1	0	0	0011 0000 0000	6390	00C303
022030	000000	1	0	0	0	1	0	0000 0011 0000	3030	103000
010030	000000	0	1	0	0	0	0	0000 0011 0000	30	000030
003600	000000	0	0	0	0	1	1	0110 0000 0000	3600	103000
0310C303	000000	1	1	0	0	0	1	0000 1100 0011 0000 0011	C303	003030

(b)

Figure 1.1 Examples of SIC/XE instructions and addressing modes.

also shown. You should carefully examine these examples, being sure you understand the different addressing modes illustrated.

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

Instruction Set

SIC/XE provides all of the instructions that are available on the standard version. In addition, there are instructions to load and store the new registers (LDB, STB, etc.) and to perform floating-point arithmetic operations (ADDF, SUBF, MULF, DIVF). There are also instructions that take their operands from registers. Besides the RMO (register move) instruction, these include register-to-register arithmetic operations (ADDR, SUBR, MULR, DIVR). A special *supervisor call* instruction (SVC) is provided. Executing this instruction generates an interrupt that can be used for communication with the operating system. (Supervisor calls and interrupts are discussed in Chapter 6.)

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

Input and Output

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

1.3.3 SIC Programming Examples

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

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