A Simple Computer: Organization and Programming

The purpose of this chapter is to introduce the terminology and basic functions of a simple but complete computer, mainly from a programmer’s (user’s) point of view. We call the simple hypothetical computer ASC (A Simple Computer). Although ASC appears very primitive in comparison with any commercially available machine, its organization reflects the basic structure of the most complex modern computer. The instruction set is limited but complete enough to write powerful programs. Assembly language programming and understanding of assembly process are a must for a system designer. We will not outline the trade-offs involved in selecting the architectural features of a machine in this chapter. Subsequent chapters of this book, however, deal with such tradeoffs. The detailed hardware design of ASC is provided in Chapter 6. Chapters 7 through 15 examine selected architectural attributes of commercially available computer systems.

5.1 A SIMPLE COMPUTER

Figure 5.1 shows the hardware components of ASC. We will assume that ASC is a 16-bit machine; hence the unit of data manipulated by and transferred between various registers of the machine is 16 bits long. Recall that a register is a storage device capable of holding certain number of bits. Each bit corresponds to a flip-flop. The data bits written (loaded) into the register remain in the register until new data are loaded, as long as the power is on. The data in a register can be read and transferred to other registers. The read operation does not change the content of the register.

Figure 5.2 shows the model of a random-access memory (RAM) used as the main memory of ASC. Other types of memory systems are described in Chapter 9. In a RAM, any addressable location in the memory can be accessed in a random manner. That is, the process of reading from and writing into a location in a RAM is the same and consumes an equal amount of time, no matter where the location is.
Figure 5.1 ASC hardware components.

Figure 5.2 Random access (read/write) memory.
The two types of RAM available are read/write memory (RWM) and read-only memory (ROM).

The most common type of main memory is the RWM, whose model is shown in Figure 5.2. In an RWM, each memory register or memory location has an “address” associated with it. Data are input into (written into) and output from (read from) a memory location by accessing the location using its “address.” The memory address register (MAR) stores such an address. With \( n \) bit in the MAR, \( 2^n \) locations can be addressed, and they are numbered from 0 through \( 2^n - 1 \).

Transfer of data in and out of memory is usually in terms of a set of bits known as a memory word. Each of the \( 2^n \) words in the memory has \( m \) bit. Thus, this is a \( (2^n \times m) \)-bit memory. This is a common notation used to describe RAMs. In general, an \( (N \times M) \) unit memory contains \( N \) words of \( M \) units each. A “unit” is either a bit, a byte (8 bit), or a word of certain number of bits. A memory buffer register (MBR) is used to store the data to be written into or read from a memory word. To read the memory, the address of the memory word to be read from is provided in MAR and the read signal is set to 1. A copy of the contents of the addressed memory word is then brought by the memory logic into the MBR. The content of the memory word is thus not altered by a read operation. To write a word into the memory, the data to be written are placed in MBR by external logic; the address of the location into which the data are to be written is placed in MAR; and the write signal is set to 1. The memory logic then transfers the MBR content into the addressed memory location. The content of the memory word is thus altered during a write operation.

A memory word is defined as the most often accessed unit of data. The typical word sizes used in memory organizations of commercially available machines are 6, 16, 32, 36, and 64 bit. In addition to addressing a memory word, it is possible to address a portion of it (e.g., half-word, quarter-word) or a multiple of it (e.g., double word, quad word), depending on the memory organization. In a “byte-addressable,” memory, for example, an address is associated with each byte (8 bit per byte) in the memory, and a memory word consists of one or more bytes.

The literature routinely uses the acronym RAM to mean RWM. We follow this popular practice and use RWM only when the context requires us to be more specific. We have included MAR and MBR as components of the memory system in this model. In practice, these registers may not be located in the memory subsystem, but other registers in the system may serve the functions of these registers.

ROM is also a RAM, except that data can only be read from it. Data are usually written into a ROM either by the memory manufacturer or by the user in an off-line mode; that is, by special devices that can write (burn) the data pattern into the ROM. A ROM is also used as main memory and contains data and programs that are not usually altered in real time during the system operation. Chapter 9 provides further description of ROM and other memory systems.

With a 16-bit address we can address \( 2^{16} \) (i.e., \( 2^6 \times 2^{10} = 64 \times 1024 \)) \( 64K \) memory words, where \( K = 2^{10} \) or 1024. For ASC we will assume a memory with \( 64K \), 16-bit words. A 16-bit long MAR is thus required. The MBR is also 16-bit long. MAR stores the address of a memory location to be accessed, and MBR receives the data from the memory word during a memory-read operation and
retains the data to be written into a memory word during a memory-write operation. These two registers are not normally accessible by the programmer.

ASC is a stored-program machine. That is, programs are stored in the memory. During the execution of the program, each instruction from the stored program is first fetched from the memory into the control unit and then the operations called for by the instruction are performed (i.e., the instruction is executed). Two special registers are used for performing fetch–execute operations: a program counter (PC) and an instruction register (IR). The PC contains the address of the instruction to be fetched from the memory and is usually incremented by the control unit to point to the next instruction address at the end of an instruction fetch. The instruction is fetched into IR. The circuitry connected to IR decodes the instruction and generates appropriate control signals to perform the operations called for by the instruction. PC and IR are both 16-bit long in ASC.

There is a 16-bit accumulator register (ACC) used in all arithmetic and logic operations. As the name implies, it accumulates the result of arithmetic and logic operations.

There are three index registers (INDEX 1, 2, and 3) in ASC that are used in manipulation of addresses. We will discuss the function of these registers later in this chapter.

There is a 5-bit processor status register (PSR) whose bits represent carry (C), negative (N), zero (Z), overflow (V), and interrupt enable (I) conditions. If an operation in the arithmetic/logic unit results in a carry from the most significant bit of the accumulator, then the carry bit is set. Negative and zero bits indicate the status of the accumulator after each operation that involves the accumulator. The interrupt-enable flag indicates that the processor can accept an interrupt. Interrupts are discussed in Chapter 7. The overflow bit is provided to complete the PSR illustration but is not used further in this chapter.

A console is needed to permit operator interaction with the machine. ASC console permits the operator to examine and change the contents of memory locations and initialize the program counter. Power ON/OFF and START/STOP controls are also on the console. The console has a set of 16 switches through which a 16-bit data word can be entered into ASC memory. There are 16 lights (monitors) that can display 16-bit data from either a memory location or a specified register. To execute a program, the operator first loads the programs and data into the memory, then sets the PC contents to the address of the first instruction in the program and STARTs the machine. The concept of a console is probably old-fashioned, since most of the modern machines are designed to use one of the I/O devices (such as a terminal) as the system console. A console is necessary during the debugging phase of the computer design process. We have included a console to simplify the discussion of program loading and execution concepts.

During the execution of the program, additional data input (or output) is done through an input (or output) device. For simplicity, we assume that there is one input device that can transfer a 16-bit data word into the ACC and one output device that can transfer the 16-bit content of the ACC to an output medium. It could very well be that the keyboard of a terminal is the input device and its display is the
output device. Note that the data in the ACC are not altered due to the output, but an input operation replaces the original ACC content with the new data.

5.1.1 Data Format

ASC memory is an array of up to $64K$ 16-bit words. Each of these 16-bit words will be either an instruction or a 16-bit unit of data. The exact interpretation depends on the context in which the machine accesses a particular memory word. The programmer should be aware (at least at the assembly and machine-language programming levels) of the data and program segments in the memory and should make certain that a data word is not accessed during a phase in which the processor is accessing an instruction and vice versa.

Only fixed-point (integer) arithmetic is allowed on ASC. Figure 5.3a shows the data format: the most significant bit is the sign bit followed by 15 magnitude bits. Since ASC uses 2s complement representation, the sign and magnitude bits are

![Data format diagram](image)

Figure 5.3a shows the data format: the most significant bit is the sign bit followed by 15 magnitude bits. Since ASC uses 2s complement representation, the sign and magnitude bits are

![Instruction format diagram](image)

Figure 5.3 ASC data and instruction formats.
treated alike in all computations. Note also the four-digit hexadecimal notation used to represent the 16-bit data word. We use this notation to denote a 16-bit quantity, irrespective of whether it is data or an instruction.

### 5.1.2 Instruction Format

Each instruction in an ASC program occupies a 16-bit word. An instruction word has four fields, as shown in Figure 5.3b. Bits 15 through 11 of the instruction word are used for the operation code (opcode). The opcode is a unique bit pattern that encodes a primitive operation the computer can perform. Thus, ASC can have a total of \(2^5 = 32\) instructions. We use an instruction set with only 16 instructions for simplicity in this book. The opcodes for these 16 instructions occupy bits 15 through 12, and bit 11 is set to 0. If the instruction set were to be expanded beyond the current set of 16, the opcodes for the new instructions would have a 1 in bit 11. Bit 10 of the instruction word is the indirect flag. This bit will be set to 1 if indirect addressing is used; otherwise it is set to 0. Bits 9 and 8 of the instruction word select one of the three index registers when indexed addressing is called for or if the instruction manipulates an index register:

<table>
<thead>
<tr>
<th>Bit 9</th>
<th>Bit 8</th>
<th>Index Register Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Bits 7 through 0 are used to represent the memory address in those instructions that refer to memory. If the instruction does not refer to memory, the indirect, index, and memory address fields are not used; the opcode field represents the complete instruction.

With only 8 bits in the address representation, ASC can directly address only \(2^8 = 256\) memory locations. That means the program and data must always be in the first 256 locations of the memory. Indexed and indirect addressing modes are used to extend the addressing range to 64K. Thus ASC has direct, indirect, and indexed addressing modes. When both indirect and indexed addressing mode fields are used, the addressing mode can be interpreted either as indexed-indirect (preindexed indirect) or as indirect-indexed (postindexed indirect). We assume that ASC allows only the indexed-indirect mode. We will describe the addressing modes further after the description of the instruction set that follows.

### 5.1.3 Instruction Set

Table 5.1 lists the complete instruction set of ASC. Column 2 shows the most significant four bits of the opcode in hexadecimal form. The fifth bit being 0 is not shown. Each opcode is also identified by a symbolic name, or mnemonic, shown in column 1.

© 2007 by Taylor & Francis Group, LLC.
We add one more construct to our HDL (hardware description language) described in Chapter 4. The memory is designed as M. A memory read operation is shown as

$$\text{MBR} \leftarrow \text{M[MAR]}.$$  

and a memory write operation is shown as

$$\text{M[MAR]} \leftarrow \text{MBR}.$$  

The operand within the [ ] can be

1. A register; the content of the register is a memory address.
2. A symbolic address; the symbolic address will eventually be associated with an absolute address.
3. An absolute address.

Thus,

$$\text{ACC} \leftarrow \text{M[27]},$$
$$\text{M[28]} \leftarrow \text{ACC},$$
$$\text{IR} \leftarrow \text{M[Z1]}, \text{ and}$$
$$\text{M[Z1]} \leftarrow \text{ACC}. \text{ (Z1 is a symbolic address.)}$$
are all valid data transfers. Further,

\[ \text{ACC} \leftarrow \text{Z1}. \]

implies that the absolute address value corresponding to Z1 is transferred to ACC. Thus,

\[ \text{Z1} \leftarrow \text{ACC}. \]

is not valid.

The ASC instruction set consists of the following three classes of instructions:

1. Zero address (TCA, HLT, SHL, and SHR).
2. One address (LDA, STA, ADD, BRU, BIP, BIN, LDX, STX, TIX, and TDX).
3. Input/output (RWD, WWD).

A description of instructions and their representation follows. In this description, hexadecimal numbers are distinguished from decimal numbers with a preceding ‘‘#H.’’

Zero-Address Instructions. In this class of instructions, the opcode represents the complete instruction. The operand (if needed) is implied to be in the ACC. The address field, the index flag and the indirect flag are not used.

| Opcode | - | -- | -------- |

A description of each instruction follows:

**HLT Stop Halt**

The HLT instruction indicates the logical end of a program and hence stops the machine from fetching the next instruction (if any).

**TCA ACC \leftarrow ACC’ + one 2s complement accumulator**

TCA complements each bit of the ACC to produce the 1s complement and then a 1 is added to produce the 2s complement. The 2s complement of the ACC is stored back into the ACC.

**SHL ACC_{15-1} \leftarrow ACC_{14-0} \quad \text{Shift left} \quad ACC_0 \leftarrow 0**

The SHL instruction shifts the contents of the ACC 1 bit to the left and fills a 0 into the least significant bit of the ACC.

**SHR ACC_{14-0} \leftarrow ACC_{15-1} \quad \text{Shift right} \quad ACC_{15} \leftarrow ACC_{15}**
The SHR instruction shifts the contents of the ACC 1 bit to the right and the most significant bit of the ACC remains unchanged. The contents of the last significant bit position are lost.

One-Address Instructions. These instructions use all 16 bit of an instruction word. In the following, MEM is a symbolic address of an arbitrary memory location. An absolute address is the physical address of a memory location, expressed as a numeric quantity. A symbolic address is mapped to an absolute address when an assembly language problem is translated into machine language. The description assumes a direct addressing mode in which MEM is the effective address (the address of the operand). The 8-bit address is usually modified by the indirect and index operations to generate the effective address of a memory operand for each of these instructions.

 Opcode X XX Xxxxxxxxx

The description of one-address instructions follows:

LDA MEM  ACC ← M[MEM]. Load accumulator

LDA loads the ACC with the contents of the memory location (MEM) specified. Contents of MEM are not changed, but the contents of the ACC before the execution of this instruction are replaced by the contents of MEM.

STA MEM  M[MEM] ← ACC. Store accumulator

STA stores the contents of the ACC into the specified memory location. ACC contents are not altered.

ADD MEM  ACC ← M[MEM]. Add

ADD adds the contents of the memory location specified to the contents of the ACC. Memory contents are not altered.

BRU MEM  PC ← MEM. Branch unconditional

BRU transfers the program control to the address MEM. That is, the next instruction to be executed is at MEM.

BIP MEM  IF ACC > 0 THEN PC ← MEM. Branch if ACC is positive

The BIP instruction tests the N and Z bits of PSR. If both of them are 0, then the program execution resumes at the address (MEM) specified; if not, execution continues with the next instruction in sequence. Since the PC must contain the
address of the instruction to be executed next, the branching operation corresponds to transferring the address into PC.

\[
\text{BIN MEM IF ACC} < 0 \text{ THEN PC} \leftarrow \text{MEM. Branch if accumulator negative}
\]

The BIN instruction tests the N bit of PSR; if it is 1, program execution resumes at the address specified; if not, the execution continues with the next instruction in sequence.

\[
\text{LDX MEM, INDEX INDEX} \leftarrow \text{M[MEM]. Load index register}
\]

The LDX loads the index register (specified by INDEX) with the contents of memory location specified. In the assembly language instruction format, INDEX will be 1, 2, or 3.

\[
\text{STX MEM, INDEX M[MEM]} \leftarrow \text{INDEX. Score index register}
\]

The STX stores a copy of the contents of the index register specified by the index flag into the memory location specified by the address. The index register contents remain unchanged.

\[
\text{TIX MEM, INDEX INDEX} \leftarrow \text{INDEX + 1 Test index}
\]

\[
\text{IF INDEX = 0 THEN PC} \leftarrow \text{MEM. increment}
\]

TIX increments the index register content by 1. Next, it tests the index register content; if it is 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

\[
\text{TDX MEM, INDEX INDEX} \leftarrow \text{INDEX} - 1 \text{ Test index}
\]

\[
\text{IF INDEX} \neq 0 \text{ THEN PC} \leftarrow \text{MEM. increment}
\]

TDX decrements the index register content by 1. Next it tests the index register content; if it is not equal to 0, the program execution resumes at the address specified; otherwise, execution continues with the next sequential instruction.

It is important to note that LDX, STX, TDX, and TIX instructions “refer” to an index register as one of the operands. Indexed mode of addressing is thus not possible with these instructions since the index field is used for the index register reference. Only direct and indirect modes of addressing can be used. For example:

| LDA Z, 3 adds the contents of index register 3 to Z to compute the effective address EA. Then the contents of memory location EA are loaded into the ACC. Index register is not altered | LDX Z, 3 loads the index register 3 from the contents of memory location Z |

© 2007 by Taylor & Francis Group, LLC.
**Input/Output Instructions.** Since ASC has one input and one output device, the address, index, and indirect fields in the instruction word are not used. Thus, these are also zero-address instructions.

**RWD**  \( \text{ACC} \leftarrow \text{Input data. Read a word} \)

RWD instruction reads a 16-bit word from the input device into the ACC. The contents of the ACC before RWD are thus lost.

**WWD**  \( \text{Output} \leftarrow \text{ACC. Write a word} \)

WWD instruction writes a 16-bit word from the ACC onto the output device. ACC contents remain unaltered.

### 5.1.4 Addressing Modes

Addressing modes allowed by a machine are influenced by the programming languages and corresponding data structures that the machine uses. ASC instruction format allows the most common addressing modes. Various other modes are used in machines commercially available. They are described in Chapter 8.

ASC addressing modes are described here with reference to the load accumulator (LDA) instruction. Here, \( Z \) is assumed to be the symbolic address of the memory location 10. For each mode, the assembly language format is shown first, followed by the instruction format encoded in binary (i.e., the machine language). The effective address calculation and the effect of the instruction are also illustrated. Note that the effective address is the address of the memory word where the operand is located.

**Direct Addressing.**  
Instruction format: LDA \( Z \)

\[
\begin{array}{c|c|c|c}
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

Effective address: \( Z \) or \#HA  
Effect: \( \text{ACC} \leftarrow \text{M}[Z] \).

The effect of this instruction is illustrated in Figure 5.4a. We will use hexadecimal notation to represent all data and addresses in the following diagrams.  
Note: contents of register and memory are shown in hexadecimal.

**Indexed Addressing.**  
Instruction format: LDA \( Z, 2 \)

\[
\begin{array}{c|c|c|c}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

Effective address: \( Z \) + index register 2  
Effect: \( \text{ACC} \leftarrow \text{M}[Z + \text{index register} \ 2] \).

The number in the operand field after the comma denotes the index register used. Assuming that index register 2 contains 3, Figure 5.4b illustrates the effect...
of this instruction. The numbers in circles show the sequence of operations. Contents of index register 2 are added to $Z$ to derive the effective address $Z + 3$. Contents of location $Z + 3$ are then loaded into the accumulator, as shown in Figure 5.4b.

![Figure 5.4](image)

**Figure 5.4** Addressing modes.

*Note: Contents of register and memory are shown in hexadecimal.*

(continued)
Note that the address field of the instruction refers to Z and the contents of the index register specify an offset from Z. Contents of an index register can be varied by using LDX, TIX, AND TDX instructions, thereby accessing various memory consecutive memory locations dynamically, by changing the contents of the index register. Further, since index registers are 16 bits wide, the effective address can be 16 bits long, thereby extending the memory addressing range to 64K from the range of 256 locations possible with 8 address bits.

The most common use of indexed addressing mode is in referencing the elements of an array. The address field in the instruction points to the first element. Subsequent elements are referenced by incrementing the index register.

**Indirect Addressing.**

Instruction format: LDA*Z

```
00010 1 00 0000 1010
```
Effective address: \( M[Z] \)
Effect: \[ \begin{align*}
\text{MAR} &\leftarrow M[Z]. \\
\text{ACC} &\leftarrow M[\text{MAR}]. \\
\text{i.e., } \text{ACC} &\leftarrow M[M[Z]]. 
\end{align*} \]

The asterisk next to the mnemonic denotes the indirect addressing mode. In this mode, the address field points to a location where the address of the operand can be found (see Figure 5.4c).

Since a memory word is 16 bits long, the indirect addressing mode can also be used to extend the addressing range to 64K. Further, by simply changing the contents of location \( Z \) in the above illustration, we can refer to various memory addresses using the same instruction. This feature is useful, for example, in creating a multiple jump instruction in which contents of \( Z \) are dynamically changed to refer to the appropriate address to jump to. The most common use of indirect addressing is in referencing data elements through pointers. A pointer contains the address of the data to be accessed. The data access takes place through indirect addressing, using the pointer as the operand. When data are moved to other locations, it is sufficient to change the pointer value accordingly, in order to access the data from the new location.

If both indirect and index flags are used, there are two possible modes of effective address computation, depending on whether indirecting or indexing is performed first. They are illustrated here.

**Indexed-Indirect Addressing (Preindexed-Indirect).**
Instruction format: \( \text{LDA}^{*}Z, 2 \)

| 00010 | 1 | 10 | 0000 1010 |

Effective address: \( M[Z + \text{index register } 2] \)
Effect: \( \text{ACC} \leftarrow M[M[Z + \text{index register } 2]]. \)

Indexing is done first, followed by indirect to compute the effective address whose contents are loaded into the accumulator, as shown in Figure 5.4d.

**Indirect-Indexed Addressing (Postindexed-Indirect).**
Instruction format: \( \text{LDA}^{*}Z, 2 \)

| 00010 | 1 | 10 | 00001010 |

Effective address: \( M[Z] + \text{index register } 2 \)
Effect: \( \text{ACC} \leftarrow M[M[Z] + \text{index register } 2]. \)

Indirect is performed first, followed by indexing to compute the effective address whose contents are loaded into the accumulator, as shown in Figure 5.4e.

Note that the instruction formats in the above two modes are identical. ASC cannot distinguish between these two modes. The indirect flag must be expanded to 2 bits if both of these modes are to be allowed. Instead, we assume that ASC always performs preindexed-indirect, and postindexed-indirect is not supported.

The above addressing modes are applicable to all single-address instructions. The only exceptions are the index-reference instructions (LDX, STX, TIX, and TDIX) in which indexing is not permitted.
Consider an array of pointers located in consecutive locations in the memory. The preindexed-indirect addressing mode is useful in accessing the data elements since we can first index to a particular pointer in the array and indirect on that pointer to access the data element. On the other hand, the postindexed-indirect mode is useful in setting up pointers to an array since we can access the first element of the array by indirecting on the pointer and access subsequent elements of the array by indexing over the pointer value.

### 5.1.5 Other Addressing Modes

Chapter 8 describes several other addressing modes that are employed in practice. For example, it is convenient sometimes in programming to include data as part of the instruction. *Immediate addressing mode* is used in such cases. Immediate addressing implies that the data are part of the instruction itself. This mode is not allowed in ASC, but the instruction set can be extended to include instructions such as load immediate (LDI), add immediate (ADI), etc. In such instructions, the opcode field will contain a 5-bit opcode, and the remaining 11 bits will contain the data. For instance,

- LDI 10 would imply loading 10 into ACC, and
- ADI 20 would imply adding 20 to the ACC.

Since ASC does not permit this addressing mode, the ASC assembler is designed to accept the so-called *literal addressing mode*, which simulates the immediate addressing mode on ASC. Refer to the following section for further details.

### 5.1.6 Addressing Limitations

As discussed earlier, ASC instruction format restricts the direct-addressing range to the first 256 locations in the memory. Thus, if the program and data can fit into locations 0 through 255, no programming difficulties are encountered. If this is not possible, the following programming alternatives can be used:

1. The program resides in the first 256 locations, and the data resides in higher-addressed locations in the memory. In this case, all instruction addresses can be represented by the 8-bit address field. Since data references require an address field longer than 8 bit, all data references are handled using indexed and indirect addressing modes. For example, the data location 300 can be loaded into the ACC by either of the following instructions:
   a. LDA 0,2 assuming that index register 2 contains 300.
   b. LDA* 0 assuming that location 0 in the memory contains 300.
2. Data reside in the first 256 locations, and the program resides beyond location 255. In this case, all data reference instructions (such as LDA, STA, etc.) can use direct, indirect, and/or indexed modes, but all other memory reference instructions (such as BRU, BIP, BIN) must use indexed and/or indirect modes.
3. If the program and data both reside beyond location 255, all memory reference instructions must be indirect and/or indexed.
Recall that the index reference instructions can use only the direct and indirect modes of addressing.

5.1.7 Machine Language Programming

It is possible to write a program for ASC using absolute addresses (actual physical memory addresses) and opcodes only, since the instruction set and instruction and data formats are now known. Such programs are called machine language programs. They need not be further translated for the hardware to interpret them since they are already in binary form. Programming at this level is tedious, however. A program in ASC machine language to add two numbers and store the sum in a third location is shown below:

```
0001 0000 0000 1000
0011 0000 0000 1001
0001 0000 0000 1000
```

Can you decode these instructions and determine what the program is doing?

Modern-day computers are seldom programmed at this level. All programs must be at this level, however, before execution of the program can begin. Translators (assemblers and compilers) are used in converting programs written in assembly and high-level languages into this machine language. We will discuss a hypothetical assembler for ASC assembly language in the next section.

5.2 ASC ASSEMBLER

An assembler that translates ASC assembly language program into machine language programs is available. We will provide details of the language as accepted by this assembler and outline the assembly process in this section.

An assembly language program consists of a sequence of statements (instructions) coded in mnemonics and symbolic addresses. Each statement consists of four fields: label, operation (mnemonic), operand, and comments, as shown in Figure 5.5.

```
<table>
<thead>
<tr>
<th>Label</th>
<th>Mnemonic</th>
<th>Operand</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consists of alphabetic and numeric characters</td>
<td>Comprises three-character standard symbolic opcodes</td>
<td>Consists of absolute and symbolic addresses</td>
<td>Starts with a &quot;.&quot; Assembler ignores it</td>
</tr>
<tr>
<td>First character must be alphabetic</td>
<td>An &quot;*&quot; as fourth character signifies indirect addressing</td>
<td>Index register designations following a &quot;*&quot;</td>
<td></td>
</tr>
<tr>
<td>An &quot;*&quot; as first character denotes that complete statement is a comment</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

1Optional field

Figure 5.5 Assembly language statement format.

Note: A space (partition) is required between label and mnemonic fields and between mnemonic and operand fields.

© 2007 by Taylor & Francis Group, LLC.
The label is a symbolic name denoting the memory location where the instruction itself is located. It is not necessary to provide a label for each statement. Only those statements that are referenced from elsewhere in the program need labels. When provided, the label is a set of alphabetic and numeric characters, the first of which must be an alphabetic character. The mnemonic field contains the instructions mnemonic. A "*" following the instruction mnemonic denotes indirect addressing. The operand field consists of symbolic addresses, absolute addresses, and index register designations. Typical operands are shown in Figure 5.6.

The comments fields start with a ".". This optional field consists only of comments by the programmer. It does not affect the instruction in any way and is ignored by the assembler. An "*" as the first character in the label field designates that the complete statement is a comment.

Each instruction in an assembly language program can be classified as either an executable instruction or an assembler directive (or a pseudoinstruction). Each of the 16 instructions in ASC instruction set is an executable instruction. The assembler generates a machine language instruction corresponding to each such instruction in the program. A pseudoinstruction is a directive to the assembler. This instruction is used to control the assembly process, to reserve memory locations, and to establish constants required by the program. The pseudo-instructions when assembled do not generate machine language instructions and as such are not executable. Care must be taken by the assembly language programmer to partition the program such that an assembler directive is not in the execution sequence. A description of ASC pseudo-instructions follows:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Decimal 25 (by default)</td>
<td>7</td>
</tr>
<tr>
<td>#H25</td>
<td>Hexadecimal 25</td>
<td>A</td>
</tr>
<tr>
<td>#O25</td>
<td>Octal 25</td>
<td>8</td>
</tr>
<tr>
<td>#B1001</td>
<td>Binary 1001</td>
<td>9</td>
</tr>
<tr>
<td>Z</td>
<td>Symbolic address Z</td>
<td>7</td>
</tr>
<tr>
<td>Z, 1</td>
<td>Z indexed with index register 1</td>
<td>A</td>
</tr>
<tr>
<td>Z + 4</td>
<td>Four locations after Z</td>
<td>B</td>
</tr>
<tr>
<td>Z + 4, 1</td>
<td>Address Z + 4 indexed with register 1</td>
<td>C</td>
</tr>
<tr>
<td>Z - 4</td>
<td>Address Z - 4 (four locations before Z)</td>
<td>D</td>
</tr>
<tr>
<td>Z - P</td>
<td>Z and P are symbolic addresses;</td>
<td>E</td>
</tr>
<tr>
<td>Z - P</td>
<td>yields an absolute address;</td>
<td></td>
</tr>
<tr>
<td>Z must be at a higher physical address than P for Z - P value to be positive.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The comments | Description | Memory |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7Z - 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z - 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z - 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z + 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z + 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7Z + 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.6 ASC operands.

The function of ORG directive is to provide the assembler the memory address where the next instruction is to be located. ORG is usually the first instruction in the assembly language program.

© 2007 by Taylor & Francis Group, LLC.
program. Then the operand field of ORG provides the starting address (i.e., the address where the first instruction in the program is located). If the first instruction in the program is not ORG, the assembler defaults to a starting address of 0. There can be more than one ORG directive in a program.

END Address Physical end

END indicates the physical end of the program and is the last statement in a program. The operand field of the END normally contains the label of the first executable statement in the program.

EQU Equate

EQU provides a means of giving multiple symbolic names to memory locations, as shown by the following example:

Example 5.1

A EQU B  A is another name for B. (B must already be defined.)
A EQU B + 5  A is the name of location B + 5.
A EQU 10  A is the name of the absolute address 10.

BSS Block storage starting

BSS is used to reserve blocks of storage locations for intermediate or final results

Example 5.2

Z BSS 5  Reserves five locations, the first of which is named Z.

The operand field always designates the number of locations to be reserved. Contents of these reserved locations are not defined.

BSS Block storage constants

BSC provides a means of storing constants in memory locations in addition to reserving those locations. The operand field consists of one or more operands (separate by a ‘,’). Each operand requires one memory word.
Example 5.3

Z   BSC 5  Reserves one location named Z containing a 5.
P   BSC 5, −6, 7  Reserves three locations: P containing 5, P + 1 containing −6 and P + 2 containing 7.

*Literal Addressing Mode.* It is convenient for the programmer to be able to define constants (data) as a part of the instruction. This feature also makes an assembly language program more readable. Literal addressing mode enables this. A literal is a constant preceded by an “¼.” For example,

\[ \text{LDA} \ ¼ \ 2 \]

implies loading a constant 2 (decimal) into the accumulator, and

\[ \text{ADD} \ ¼ \ #\text{H}10 \]

implies adding a #H10 to the accumulator. ASC assembler recognizes such literals, reserves an available memory location for the constant in the address field, and substitutes the address of the memory location into the instruction.

We will now provide some assembly language programs as examples.

Example 5.4

Figure 5.7 shows a program to add three numbers located at A, B, and C and save the result at D. The program is ORiGined to location 10. Note how the HLT

```
* Program to add three numbers

ORG   10
BEGIN LDA A
      ADD B
      ADD C
      STA D
      HLT

A   BSC 5
B   BSC 7
C   BSC −3
D   BSS 1

END BEG
```
statement separates the program logic from the data block A, B, and C are defined using BSC statements, and one location is reserved for D using a BSS.

**Example 5.5**

Figure 5.8 shows a program to accumulate five numbers stored starting at location X in memory and store the result at Z. Here, index register 1 is first set to 4 so as to point to the last number in the block (i.e., at X + 4). The ACC is set to 0. TDX is used to access numbers one at a time from the last to first and to terminate the loop after all the numbers are accumulated.

```
* A program to accumulate the values of
* five numbers located at x

ORG 0
START
  LDX =4,1  .Zero accumulator
  LDA =0
LOOP
  ADD X,1  .ADD loop
  TDX LOOP,1
  ADD X  .ADD the remaining value
  STA Z  .Store the result in Z
HLT

X BSC 5,35,26,−7,4
Z BSS 1
END START
```

*Figure 5.8* Program to accumulate a block of numbers.

**Example 5.6**

Division can be treated as the repeated subtraction of the divisor from the divided until a zero or negative result is obtained. The quotient is equal to the maximum number of times the subtraction can be performed without yielding a negative result. *Figure 5.9* shows the division routine.

The generation of the object code from the assembly language programs is described in the following section.

**5.2.1 Assembly Process**

The major functions of the assembler program are (1) to generate an address for each symbolic name in the program and (2) to generate the binary equivalent of each assembly instruction. The assembly process is usually carried out in two scans over the source program. Each of these scans is called a *pass* and the assembler is called a *two-pass assembler*. The first pass is used for allocating a memory location for each symbolic name used in the program; during the second pass, references to these symbolic names are resolved. If the restriction is made that each symbol must be defined before it can be referenced, one pass will suffice.
Details of the ASC two-pass assembler are given in Figures 5.10 and 5.11. The assembler uses a counter known as location counter (LC) to keep track of the memory locations used. If the first instruction in the program is ORG, the operand field of ORG defines the initial value of LC; otherwise, LC is set to 0. LC is incremented appropriately during the assembly process. Content of LC at any time is the address of the next available memory location.

The assembler performs the following tasks during the first pass:

1. Enters labels into a symbol table along with the LC value as the address of the label
2. Validates mnemonics
3. Interprets pseudo-instructions completely
4. Manages the location counter
The major activities during the second pass are

1. Evaluation of the operand field
2. Insertion of opcode, address, and address modifiers into the instruction format
3. Resolution of literal addressing

---

**Figure 5.10** Assembler pass 1. (continued)
The assembler uses an opcode table to extract opcode information. The opcode table is a table storing each mnemonic, the corresponding opcode, and any other attribute of the instruction useful for the assembly process. The symbol table created by the assembler consists of two entries for each symbolic name: the symbol itself and the address in which the symbol will be located. We will illustrate the assembly process in Example 5.7.

**Example 5.7**

Consider the program shown in Figure 5.12a. The symbol table is initially empty. Location counter starts at the default value of 0. The first instruction is ORG. Its operand field is evaluated and the value (0) is entered into LC. The label field of the next instruction is BEGIN. BEGIN is entered into symbol table and assigned the address of 0. The mnemonic field has LDX, which is a valid mnemonic. Since this instruction takes up one memory word, LC is incremented by 1. The operand field of LDX instruction is not evaluated during the first pass. This process of scanning the label and mnemonic fields, entering labels (if any) into the symbol table, validating mnemonic, and incrementing LC continues until END instruction is reached. Pseudo-instructions are completely evaluated during this pass. Location counter values are shown in Figure 5.12a along with symbol table entries at the end of pass 1 in Figure 5.12b. By the end of the first pass, the location counter will have advanced to E, since BSS 4 takes up four locations (A through D).
During the second pass, machine instructions are generated using the source program and the symbol table. The operand fields are evaluated during this pass and instruction format fields are appropriately filled for each instruction. Starting with LC = 0, the label field of instruction at 0 is ignored and the opcode (11000) is substituted for the mnemonic LDX. Since this is a one-address instruction, the operand field is evaluated. There is no "*" next to the mnemonic, and hence

![Flowchart Image](image-url)

**Figure 5.11** Assembler pass 2.

(continued)
the indirect flag is set to 0. The absolute address of C is obtained from the symbol table and entered into the address field of the instruction, and the index flag is set to 01. This process continues for each instruction until END is reached. The object code is shown in Figure 5.9c in binary and hexadecimal formats. The symbol table shown in Figure 5.9d has one more entry corresponding to the literal = 0. Note that the instruction LDX = 0,2 has been assembled as LDX #HE, 2 with the location #HE containing a 0. Contents of the words reserved in response to BSS are not defined, and unused bits of HLT instruction words are assumed to be 0s.

5.3 PROGRAM LOADING

The object code must be loaded into the machine memory before it can be executed. ASC console can be used to load programs and data into the memory. Loading
through the console is tedious and time consuming, however, especially when programs are large. In such cases, a small program that reads the object code statements from the input device and stores them in appropriate memory locations is first written, assembled, and loaded (using the console) into machine memory. This loader program is then used to load the object code or data into the memory.

Figure 5.13 shows a loader program for ASC. Instead of loading this program each time through the console, it can be stored in a ROM that forms part of the 64K memory space of ASC. Then loading can be initiated by setting the PC to the beginning address of the loader (using the console) and starting the machine. Note that the loader occupies locations 224 through 255. Hence, care must be taken to make sure that other programs do not overwrite this space.

Note also that the assembler itself must be in the binary object code form and loaded into ASC memory before it can be used to assemble other programs. That means the assembler must be translated from the source language to ASC binary code either manually or by implementing the assembler on some other machine. It can then be loaded into ASC either by using the loader or by retaining it in a ROM portion of the memory for further use.
5.4 SUBROUTINES

A subroutine (function, method, procedure, or subprogram) is a portion of code within a larger program, which performs a specific task and is independent of the remaining code. The syntax of many programming languages includes support for creating self-contained subroutines.

A subroutine consists of instructions for performing some task, chunked together and given a name. “Chunking” allows us to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go through to perform that task, we just need to remember the name of the subroutine. Whenever we want our program to perform the task, we just “call” the subroutine. Subroutines are a major tool for handling the complexity. They reduce the redundancy in a program, enable reuse of code across multiple programs, allow us to decompose complex problems into simpler pieces, and improve readability of a program.

Typical components of a subroutine are a body of code to be executed when the subroutine is called, parameters that are passed to the subroutine from the point where it is called, and one or more values that are returned to the point where the call occurs.

Some programming languages, like Pascal and FORTRAN, distinguish between functions, which return values, and subroutines or procedures, which do not. Other languages, like C and LISP, do not make this distinction, and treat those terms as synonymous. The name method is commonly used in connection with object-oriented programming, specifically for subroutines that are part of objects.

Calling a subroutine means jumping to the first instruction in the subroutine, using a JMP instruction. The execution of the subroutine will end with a jump back to the same point in the program from which the subroutine was called, so that the program can pick up where it left off before calling the subroutine. This is known as returning from the subroutine. If the subroutine is to be reusable in a meaningful

<table>
<thead>
<tr>
<th>Label</th>
<th>Mnemonic</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG</td>
<td>244</td>
<td></td>
<td>Read starting address</td>
</tr>
<tr>
<td>RWD</td>
<td></td>
<td></td>
<td>.Read starting address</td>
</tr>
<tr>
<td>STA</td>
<td>SAVE</td>
<td></td>
<td>.Starting address in index register 2</td>
</tr>
<tr>
<td>LDX</td>
<td>SAVE,2</td>
<td></td>
<td>.Input number of statements N</td>
</tr>
<tr>
<td>RWD</td>
<td></td>
<td></td>
<td>.SAVE contains N</td>
</tr>
<tr>
<td>STA</td>
<td>SAVE</td>
<td></td>
<td>.Load (N) into index register 1</td>
</tr>
<tr>
<td>LDX</td>
<td>SAVE,1</td>
<td></td>
<td>.Input an object code statement</td>
</tr>
<tr>
<td>LOOP</td>
<td>RWD</td>
<td>0,2</td>
<td>.Store the object code statement</td>
</tr>
<tr>
<td>LL</td>
<td>TIX</td>
<td>LL,2</td>
<td>.Increment index register 2</td>
</tr>
<tr>
<td>TDX</td>
<td>LOOP,1</td>
<td></td>
<td>.Decrement N by 1 and loop</td>
</tr>
<tr>
<td>SAVE</td>
<td>BSS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>START</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.13 Loader for ASC.
sense, it must be possible to call the subroutine from many different places in a program. If this is the case, how does the computer know what point in the program to return to when the subroutine ends? The answer is that the return point has to be recorded somewhere before the subroutine is called. The address in memory to which the computer is supposed to return after the subroutine ends is called the **return address**. Before jumping to the start of the subroutine, the program must store the return address in a place where the subroutine can find it. When the subroutine has finished performing its assigned task, it ends with a jump back to the return address.

Let us expand ASC instruction set to include two instructions to handle subroutines: jump to subroutine (JSR) and return from subroutine (RET). The following program illustrates the subroutine call and return operations.

```assembly
ORG 0
BEGIN
LDA X
JSR SUB1
A
WWD
HLT
X
BSC 23
SUB1
STA Z
ADD Z
RET
END BEGIN
```

The main program calls the subroutine SUB1 through JSR. The program control transfers to SUB1, and the two instructions in the subroutine are executed followed by RET. The effect of executing RET is to return control to the return address A in the main program. The main program executes WWD followed by HLT.

To enable this operation, the JSR instruction has to store the return address (i.e., the address of the instruction following JSR) somewhere, before jumping to the subroutine. When RET is executed, the return address is retrieved and the program transfers to the instruction following the call.

The return address can be stored in a dedicated register or a dedicated memory location. If the subroutine calls another subroutine (i.e., nested call), the return address corresponding to the first call will be lost and hence the program cannot return properly. We typically push the return address on to a stack (see Appendix B for details on stack implementation) during the call and it is popped from the stack during return. This allows subroutine calls to be nested.

The return address is not the only item of information that the program has to send to the subroutine. If the task of the subroutine is to multiply a number by seven, the main program has to tell the subroutine which number to multiply by seven. This information is said to be a parameter of the subroutine. Similarly, the subroutine has to get its answer, the result of multiplying the parameter value by seven back to the main program. This answer is called the return value of the subroutine. In the program above, the parameter value is in the accumulator before calling the subroutine. The subroutine knows to look for it there. Before it jumps back to the main program, the subroutine puts its return value in the accumulator. The main
program knows to look for it there. Passing parameter values and return values back and forth in a register, such as the accumulator, is a very simple and efficient method of communication between a subroutine and the rest of a program. In ASC, the ACC and the three index registers can be used for parameter passing. A more common way of passing parameters is for the main program to push them onto the stack just prior to calling the subroutine, and the subroutine to retrieve them from the stack, operate on them, and return the results back on the stack, for the main program.

5.5 MACROS

A macro is a group of instructions which are codified only once and can be used as many times as necessary in a program. Unlike subroutines, when a macro is called from a program, the macro call is replaced by the group of instructions that constitute the macro body. Thus, each call to the macro results in the substitution of the macro body. We can also pass parameters to a macro. When a macro call is executed, each parameter is substituted by the name or value specified at the time of the call.

An example of macro definition is shown here:

```
MACRO ADD2
BIP POS
ADD = 1
POS ADD = 1
ENDM
```

MACRO is an assembler directive indicating the macro definition. ADD2 is the macro name. ENDM signifies the end of macro definition. The instructions in the macro body add 1 to the accumulator if it is positive; otherwise, they add 2 to the accumulator.

To call the macro we simply use ADD2 as an instruction in the program. The assembler will replace ADD2 with the three instructions corresponding to the macro body every time ADD2 is used in the program. Usually this substitution (macro expansion) is done prior to the first pass of the assembler.

A macro to accumulate two values A and B and store the result in C is defined below. A, B, C are parameters to the macro.

```
MACRO ABC (A, B, C)
LDA A
ADD B
STA C
ENDM
```

A call to this macro would be

```
ABC(X, Y, Z)
```
This call would result in the replacement of A, B, and C with X, Y, and Z during the macro expansion. Note that once the macro is expanded, A, B, and C are not visible in the program. Also, X, Y, and Z must be defined.

Macros allow the programmer to define repetitive code blocks once and use the macro call to expand them into the program. Thus, macros convert into the code inline with the program they are called. The control overhead required by the subroutines (to save and retrieve return address) is thus not needed. The advantage of subroutines is that the subroutine body is not expanded into the calling program, even with multiple calls. Subroutines save instruction memory at the cost of run time overhead. Macros consume instruction memory, but eliminate subroutine call/return overhead.

One of the facilities that the use of macros offers is the creation of libraries, which are groups of macros which can be included in a program from a different file. The creation of these libraries is very simple. We first create a file with all the macro definitions and save it as a text file such as MACROS1. To call these macros it is only necessary to use an instruction such as Include MACROS1 at the beginning of the program.

5.6 LINKERS AND LOADERS

In practice, an executable program is composed of a main program and several subroutines (modules). The modules can either come from a predefined library or developed by the programmer for the particular application. Assemblers (and compilers) allow independent translation of program modules into corresponding machine code. A linker is a program that takes one or more modules generated by assemblers and compilers and assembles them into a single executable program. During the linking process, object files and static libraries are assembled into a new library or an executable program.

The program modules contain machine code and information for the linker. This information comes mainly in the form of two types of symbol definitions:

1. Defined or exported symbols are functions or variables that are present in the module represented by the object, and which should be available for use by other modules.
2. Undefined or imported symbols are functions or variables that are called or referenced by this object, but not internally defined.

The linker’s job is to resolve references to undefined symbols by finding out which other module defines a symbol in question, and replacing placeholders with the symbol’s address.

The linker also takes care of arranging the modules in a program’s address space. This may involve relocating code that assumes a specific base address to another base. Since an assembler (compiler) seldom knows where a module will reside, it often assumes a fixed base location. Relocating machine code may involve retargeting of absolute jumps, loads, and stores. For instance, in case of ASC...
programs the ORG directive defines the beginning address of the program module. If all the modules are assembled with their own origins, the linker has to make sure that they do not overlap in memory when put together into a single executable program. Even after the linking is done, there is no guarantee that the executable will reside at its specified origin when loaded into the machine memory, since other programs may be residing in the memory space at the time of loading. Thus, the program may have to be relocated.

Linkers and loaders perform several related but conceptually separate actions:

**Program Loading.** Copying a program from secondary storage into main memory so that it is ready to run. In some cases loading just involves copying the data from disk to memory, in others it involves allocating storage, setting protection bits, or arranging for virtual memory to map virtual addresses to disk pages.

**Relocation.** As mentioned earlier, relocation is the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses. In many systems, relocation happens more than once. It is quite common for a linker to create a program from multiple subprograms, and create one linked output program that starts at zero, with the various subprograms relocated to locations within the big program. Then, when the program is loaded, the system picks the actual load address, and the linked program is relocated as a whole to the load address.

**Symbol Resolution.** When a program is built from multiple subprograms, the references from one subprogram to another are made using symbols; a main program might use a square root routine called $\text{sqrt}$, and the math library defines $\text{sqrt}$. A linker resolves the symbol by noting the location assigned to $\text{sqrt}$ in the library, and patching the caller’s object code such that the call instruction refers to that location.

Although there is considerable overlap between linking and loading, it is reasonable to define a program that does program loading as a loader, and one that does symbol resolution as a linker. Either can do relocation, and there have been all-in-one linking loaders that do all three functions.

### 5.6.1 Dynamic Linking

Modern operating system environments allow dynamic linking, that is the postponing of the resolving of some undefined symbols until a program is run. That means that the executable still contains undefined symbols, plus a list of modules or libraries that will provide definitions for these. Loading the program will load these modules/libraries as well, and perform a final linking. This approach offers two advantages:

1. Often-used libraries (e.g., the standard system libraries) need to be stored in only one location, not duplicated in every single binary.
2. If an error in a library function is corrected by replacing the library, all programs using it dynamically will immediately benefit from the correction. Programs that included this function by static linking would have to be relinked first.
Dynamic linking means that the data in a library is not copied into a new executable or library at compile time, but remains in a separate file on disk. Only a minimal amount of work is done at compile time by the linker—it only records what libraries the executable needs and the index names or numbers. The majority of the work of linking is done at the time the application is loaded (load time) or during the execution of the program (run time). At the appropriate time, the loader finds the relevant libraries on disk and adds the relevant data from the libraries to the program’s memory space.

Some operating systems can only link in a library at load time, before the program starts executing; others may be able to wait until after the program has started to execute and link in the library just when it is actually referenced (i.e., during run time). The latter is often called “delay loading.” In either case, such a library is called a dynamically linked library.

Dynamic linking was originally developed in the Multics operating system, starting in 1964. It was also a feature of MTS (the Michigan Terminal System), built in the late 1960s. In Microsoft Windows, dynamically linked libraries are called dynamic-link libraries or “DLLs.”

One wrinkle that the loader must handle is that the location in memory of the actual library data cannot be known until after the executable and all dynamically linked libraries have been loaded into memory. This is because the memory locations used depend on which specific dynamic libraries have been loaded. It is not possible to store the absolute location of the data in the executable, not even in the library, since conflicts between different libraries would result: if two of them specified the same or overlapping addresses, it would be impossible to use both in the same program. This might change with increased adoption of 64-bit architectures, which offer enough virtual memory addresses to give every library ever written its own unique address range.

It would theoretically be possible to examine the program at load time and replace all references to data in the libraries with pointers to the appropriate memory locations once all libraries have been loaded, but this method would consume unacceptable amounts of either time or memory. Instead, most dynamic library systems link a symbol table with blank addresses into the program at compile time. All references to code or data in the library pass through this table, the import directory. At load time, the table is modified with the location of the library code/data by the loader/linker. This process is still slow enough to significantly affect the speed of programs that call other programs at a very high rate, such as certain shell scripts.

The library itself contains a table of all the methods within it, known as entry points. Calls into the library “jump through” this table, looking up the location of the code in memory, and then calling it. This introduces overhead in calling into the library, but the delay is usually so small as to be negligible.

Dynamic linkers/loaders vary widely in functionality. Some depend on explicit paths to the libraries being stored in the executable. Any change to the library naming or layout of the file system will cause these systems to fail. More commonly, only the
name of the library (and not the path) is stored in the executable, with the operating system supplying a system to find the library on-disk based on some algorithm.

Most Unix-like systems have a ‘‘search path’’ specifying file system directories in which to look for dynamic libraries. On some systems, the default path is specified in a configuration file; in others, it is hard coded into the dynamic loader. Some executable file formats can specify additional directories in which to search for libraries for a particular program. This can usually be overridden with an environment variable, although it is disabled for setuid and setgid programs, so that a user cannot force such a program to run arbitrary code. Developers of libraries are encouraged to place their dynamic libraries in places in the default search path. On the downside, this can make installation of new libraries problematic, and these ‘‘known’’ locations quickly become home to an increasing number of library files, making management more complex.

Microsoft Windows will check the registry to determine the proper place to find an ActiveX DLL, but for other DLLs it will check the directory that the program was loaded from; the current working directory (only on older versions of Windows); any directories set by calling the SetDllDirectory( ) function; the System32, System, and Windows directories; and finally the directories specified by the PATH environment variable.

One of the biggest disadvantages of dynamic linking is that the executables depend on the separately stored libraries in order to function properly. If the library is deleted, moved, or renamed, or if an incompatible version of the DLL is copied to a place that is earlier in the search, the executable could malfunction or even fail to load; damaging vital library files used by almost any executable in the system will usually render the system completely unusable.

Dynamic loading is a subset of dynamic linking where a dynamically linked library loads and unloads at run time on request. The request to load such a dynamically linked library may be made implicitly at compile time, or explicitly by the application at run time. Implicit requests are made by adding library references, which may include file paths or simply file names, to an object file at compile time by a linker. Explicit requests are made by applications using a run time linker application program interface (API).

Most operating systems that support dynamically linked libraries also support dynamically loading such libraries via a run time linker API. For instance, Microsoft Windows uses the API functions LoadLibrary, LoadLibraryEx, FreeLibrary, and GetProcAddress with Microsoft Dynamic Link Libraries; POSIX-based systems, including most UNIX and UNIX-like systems, use dlopen, dlclose, and dladdr.

5.7 SUMMARY

This chapter is a programmer’s introduction to ASC organization. Details of an assembler for ASC, assembly language programming, and the assembly process have been described. A brief introduction to program loaders and linkers has been
given. Various components of ASC have been assumed to exist, and no justification has been given as to why a component is needed. Architectural trade-offs used in selecting the features of a machine are discussed in subsequent chapters of the book. Chapter 6 provides the detailed hardware design of ASC. Further details on these topics can be found in the References.

REFERENCES


PROBLEMS

5.1 For each of the following memory systems, determine the number of bits needed in MAR and MBR, assuming a word addressable memory:
   a. 64K × 8
   b. 64K × 32
   c. 32K × 16
   d. 32K × 32
   e. 128K × 64

5.2 Answer Problem 5.1, assuming that the memory is byte addressable.

5.3 Determine the content of the MAR of a 16K, word-addressable memory for each of the following:
   a. Word 48
   b. Word 341
   c. Lower half of the memory (i.e., words 0 through 8K – 1)
   d. Upper half of the memory (i.e., words 8K through 16K – 1)
   e. Even memory words (0, 2, 4, etc.)
   f. Any of the 8 words 48 through 55.

5.4 Calculate the effective address for each of the following instructions. (Remember: An “**” as the fourth symbol denotes indirect addressing.)
   STA Z
   STA Z,3
   STA* Z
   STA* Z,3 (Preindex)
   STA* Z,3 (Postindex)
   Assume index register 3 contains 5: Z is memory location #H10 and is the first location of a memory block containing 25, 7, 4, 86, 46, and 77.
5.5 Determine the contents of the ACC after each of the following instructions, assuming that the ACC is set to 25 before each instruction is executed:

SHL
ADD = 5
TCA
ADD = #H3
SHR

5.6 Determine the content of the ACC at the end of each instruction in Problem 5.5, assuming that the ACC is set to 25 to begin with and the effect of each instruction is cumulative; that is, the ACC is not reset to 25 at the beginning of second and subsequent instructions.

5.7 What is the difference (if any) between
   a. A location counter and a program counter
   b. END and HLT
   c. BSS and BSC
   d. ORG and EQU
   e. Executable and pseudo instructions.

5.8 Show the contents (in hexadecimal) of the memory locations reserved by the following program:

<table>
<thead>
<tr>
<th></th>
<th>ORG</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSS</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>BSC</td>
<td>3, 5</td>
</tr>
<tr>
<td></td>
<td>BSS</td>
<td>24</td>
</tr>
<tr>
<td>M</td>
<td>EQU</td>
<td>Z + 1</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.9 What is the effect of inserting ORG 300 instruction after the instruction at Z, in the program of Problem 5.8?

5.10 There are two ways of loading the accumulator with a number:

<table>
<thead>
<tr>
<th></th>
<th>LDA</th>
<th>X</th>
<th>LDI 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>BSC</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

a. Which of the two executes faster? Why?
b. What is the range of the data value that can be loaded by each method?

5.11 NOP (No operation) is a common instruction found in almost all instruction sets. How does the assembler handle this instruction? Why is it needed?

5.12 Another popular addressing mode is PC-relative addressing. An example of such an instruction is BRU + 5, meaning add 5 to the current PC value to determine the target address. Similarly, the target address of BRU - 5 is determined by subtracting 5 from the current PC value.

a. How does the assembler handle this instruction?
b. What is the range of relative jump addresses possible in ASC?
c. What are the advantages and disadvantages of this mode compared to other jump addressing modes?
5.13 Give the symbol table resulting from the assembly of the following program:

```
BEGIN
  ORG 200
  STA A
  BIP TEMP 1
  BIN TEMP 1
  BRU OUT
  LDX B, 1
  LDX = 0, 2
  LDX B, 1
  LDX = 0, 2
  LOOP
    LDA C, 2
    ADD SUM
    STA SUM
    TIX TEMP2, 2
    TIX TEMP2, 2
    TIX TEMP2, 2
    LOOP TIX TEMP2, 2
    BRU LOOP
    HLT
END
```

<table>
<thead>
<tr>
<th>Section</th>
<th>Address</th>
<th>Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>200</td>
<td>ORG</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RWD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>STA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEMP 1</td>
<td>BIP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEMP 1</td>
<td>BIN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUT</td>
<td>BRU</td>
<td></td>
</tr>
<tr>
<td>TEMP1</td>
<td>B, 1</td>
<td>LDX</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>C, 2</td>
<td>LDA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUM</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUM</td>
<td>STA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TEMP2</td>
<td>TIX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUT, 1</td>
<td>TDX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOOP</td>
<td>BRU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HLT</td>
<td>OUT</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>0</td>
<td>BSC</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>BSS</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>BSS</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>BSS</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td>BEGIN</td>
<td>END</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>500</td>
<td>ORG</td>
<td></td>
</tr>
</tbody>
</table>

5.14 The dump of locations 64–77 of ASC memory is shown here. Decode the program segment represented by this object code.

```
1000 0000 0000 0000
0010 0000 1000 0001
0110 0000 0100 0101
0111 0000 0100 0101
0101 0000 0100 1101
1100 0001 1000 0010
1100 0010 1000 0111
0001 0010 1000 0011
0011 0000 1000 0000
0010 0000 1000 0000
1110 0010 0100 1011
1111 0001 0100 1011
0101 0000 0100 1111
0000 0000 0000 0000
```

5.15 Write ASC assembly language programs for the following. Start programs at locations #H0, using ORG 0 Statement.

a. Subtract an integer stored at memory location A from that at B and store the result at C.

b. Read several integers from the input device one at a time and store them in a memory location starting at Z. The input process should stop when the integer read has a value of 0.

c. Change the program in (b) to store only positive integers at a location starting at POS.

d. Modify the program in (b) to store the positive integers starting at POS and the negative integers starting at NEG.
e. Location #H50 contains an address pointing to the first entry in a table of integers. The table is also in the memory and the first entry is the number of entries in the table, excluding itself. Store the maximum and minimum valued integers at memory locations MAX and MIN, respectively.

f. SORT the entries in a table of \( n \) entries in increasing order of magnitude.

g. Multiply integers stored at memory locations A and B and store the result in C. Assume the product is small enough and can be represented in 16 bit. Note that multiplication is the repeated addition of multiplicand to itself multiplier times.

h. Compute the absolute value of each of the 50 integers located at the memory block starting at A and store them at the block starting at B.

i. Read a sequence of numbers and compute their minimum, maximum, and average values. Reading a value of 0 should terminate the reading process. What problems arise in computing the average?

5.16 Assemble each program in Problem 5.15 and list the object code in binary and hexadecimal forms.

5.17 What restrictions are to be imposed on the assembly language if a single-pass assembler is needed?

5.18 Numbers larger than \( 2^{15} - 1 \) are required for certain applications. Two ASC words can be used to store each such number. What changes to the ASC instruction set are needed to enable addition of numbers that are each stored in two consecutive memory words?

5.19 SHR instruction must be enhanced to allow multiple shifts. The address field can be used to represent the shift count; for example,

\[ \text{SHR 5} \]

implies a shift right by 5 bits. Discuss the assembler modifications needed to accommodate multiple-shift SHR. The hardware is capable of performing only one shift at a time.

5.20 Write a subroutine to subtract two numbers in ASC. Use this subroutine in a program that inputs 10 pairs of numbers and outputs the difference of each pair.

5.21 Write a MACRO to perform the subtraction of two numbers. Use it in a program that inputs 10 pairs of numbers and outputs the difference of each pair.

5.22 Compare the solutions in Problems 5.21 and 5.22 in terms of speed and memory requirements.