Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

6

MICROCOMPUTER ARCHITECTURE, PROGRAMMING, AND SYSTEM DESIGN CONCEPTS

This chapter describes the fundamental material needed to understand the basic characteristics of microprocessors. It includes topics such as typical microcomputer architecture, timing signals, internal microprocessor structure, and status flags. The architectural features are then compared to the Intel 8086 architecture. Topics such as microcomputer programming languages and system design concepts are also described.

6.1 Basic Blocks of a Microcomputer

A microcomputer has three basic blocks: a central processing unit (CPU), a memory unit, and an input/output unit. The CPU executes all the instructions and performs arithmetic and logic operations on data. The CPU of the microcomputer is called the "microprocessor." The microprocessor is typically a single VLSI (Very Large-Scale Integration) chip that contains all the registers, control unit, and arithmetic/ logic circuits of the microcomputer.

A memory unit stores both data and instructions. The memory section typically contains ROM and RAM chips. The ROM can only be read and is nonvolatile, that is, it retains its contents when the power is turned off. A ROM is typically used to store instructions and data that do not change. For example, it might store a table of codes for outputting data to a display external to the microcomputer for turning on a digit from 0 to 9.

One can read from and write into a RAM. The RAM is volatile; that is, it does not retain its contents when the power is turned off. A RAM is used to store programs and data that are temporary and might change during the course of executing a program. An I/O (Input/Output) unit transfers data between the microcomputer and the external devices via I/O ports (registers). The transfer involves data, status, and control signals.

In a single-chip microcomputer, these three elements are on one chip, whereas with a single-chip microprocessor, separate chips for memory and I/O are required. Microcontrollers evolved from single-chip microcomputers. The microcontrollers are typically used for dedicated applications such as automotive systems, home appliances, and home entertainment systems. Typical microcontrollers, therefore, include on-chip timers and A/D (analog to digital) and D/A (digital to analog) converters. Two popular

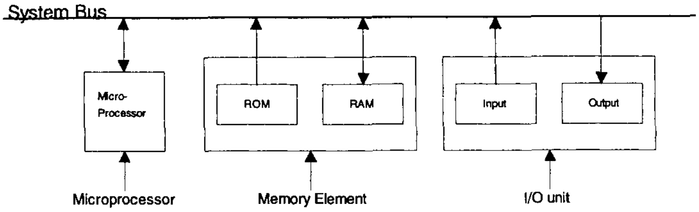


FIGURE 6.1 Basic blocks of a microcomputer

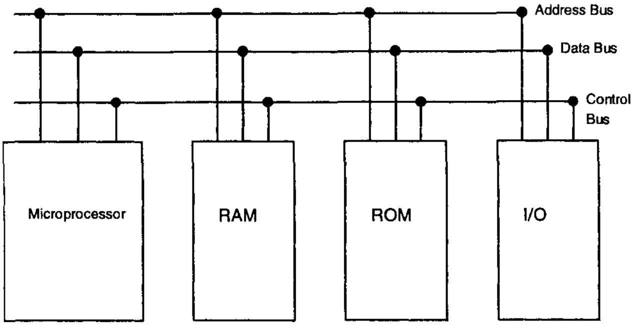


FIGURE 6.2 Simplified version of a typical microcomputer

microcontrollers are the Intel 8751 (8 bit)/8096 (16 bit) and the Motorola HC11 (8 bit)/ HC16 (16 bit). The 16-bit microcontrollers include more on-chip ROM, RAM, and I/O than the 8-bit microcontrollers. Figure 6.1 shows the basic blocks of a microcomputer. The System bus (comprised of several wires) connects these blocks.

6.2 Typical Microcomputer Architecture

In this section, we describe the microcomputer architecture in more detail. The various microcomputers available today are basically the same in principle. The main variations are in the number of data and address bits and in the types of control signals they use.

To understand the basic principles of microcomputer architecture, it is necessary to investigate a typical microcomputer in detail. Once such a clear understanding is obtained, it will be easier to work with any specific microcomputer. Figure 6.2 illustrates the most simplified version of a typical microcomputer. The figure shows the basic blocks of a microcomputer system. The various buses that connect these blocks are also shown. Although this figure looks very simple, it includes all the main elements of a typical microcomputer system.

6.2.1 The Microcomputer Bus

The microcomputer's system bus contains three buses, which carry all the address, data, and control information involved in program execution. These buses connect the microprocessor (CPU) to each of the ROM, RAM, and I/O chips so that information transfer between the microprocessor and any of the other elements can take place.

In the microcomputer, typical information transfers are carried out with respect to the memory or I/O. When a memory or an I/O chip receives data from the microprocessor , it is called a WRITE operation, and data is written into a selected memory location or an I/O port (register). When a memory or an I/O chip sends data to the microprocessor, it is called a READ operation, and data is read from a selected memory location or an I/O port.

In the *address bus*, information transfer takes place only in one direction, from the microprocessor to the memory or I/O elements. Therefore, this is called a "unidirectional bus." This bus is typically 20 to 32 bits long. The size of the address bus determines the total number of memory addresses available in which programs can be executed by the microprocessor. The address bus is specified by the total number of address pins on the microprocessor chip. This also determines the direct addressing capability or the size of the main memory of the microprocessor. The microprocessor can only execute the programs located in the main memory. For example, a microprocessor with 20 address pins can generate $2^{20} = 1{,}048{,}576$ (one megabyte) different possible addresses (combinations of 1's and 0's) on the address bus. The microprocessor includes addresses from 0 to 1,048,575 (00000_{16} through $FFFFF_{16}$). A memory location can be represented by each one of these addresses. For example, an 8-bit data item can be stored at address 00200_{16}.

When a microprocessor such as the 8086 wants to transfer information between itself and a certain memory location, it generates the 20-bit address from an internal register on its 20 address pins A_0–A_{19}, which then appears on the address bus. These 20 address bits are decoded to determine the desired memory location. The decoding process normally requires hardware (decoders) not shown in Figure 6.2.

In the *data bus*, data can flow in both directions, that is, to or from the microprocessor. Therefore, this is a bidirectional bus. In some microprocessors, the data pins are used to send other information such as address bits in addition to data. This means that the data pins are time-shared or multiplexed. The Intel 8086 microprocessor is an example where the 20 bits of the address are multiplexed with the 16-bit data bus and four status lines.

The *control bus* consists of a number of signals that are used to synchronize the operation of the individual microcomputer elements. The microprocessor sends some of these control signals to the other elements to indicate the type of operation being performed. Each microcomputer has a unique set of control signals. However, there are some control signals that are common to most microprocessors. We describe some of these control signals later in this section.

6.2.2 Clock Signals

The system clock signals are contained in the control bus. These signals generate the appropriate clock periods during which instruction executions are carried out by the microprocessor. The clock signals vary from one microprocessor to another. Some microprocessors have an internal clock generator circuit to generate a clock signal. These microprocessors require an external crystal or an RC network to be connected at the appropriate microprocessor pins for setting the operating frequency. For example, the Intel 80186 (16-bit microprocessor) does not require an external clock generator circuit. However, most microprocessors do not have the internal clock generator circuit and require an external chip or circuit to generate the clock signal. Figure 6.3 shows a typical clock signal.

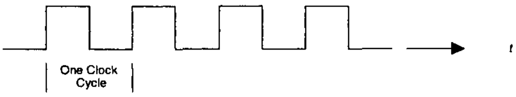


FIGURE 6.3 A typical clock signal

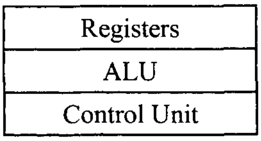


FIGURE 6.4 A microprocessor chip with the main functional elements

6.3 The Single-Chip Microprocessor

As mentioned before, the microprocessor is the CPU of the microcomputer. Therefore, the power of the microcomputer is determined by the capabilities of the microprocessor. Its clock frequency determines the speed of the microcomputer. The number of data and address pins on the microprocessor chip make up the microcomputer's word size and maximum memory size. The microcomputer's I/O and interfacing capabilities are determined by the control pins on the microprocessor chip.

The logic inside the microprocessor chip can be divided into three main areas: the register section, the control unit, and the arithmetic and logic unit (ALU). A microprocessor chip with these three sections is shown in Figure 6.4. We now describe these sections.

6.3.1 Register Section

The number, size, and types of registers vary from one microprocessor to another. However, the various registers in all microprocessors carry out similar operations. The register structures of microprocessors play a major role in designing the microprocessor architectures. Also, the register structures for a specific microprocessor determine how convenient and easy it is to program this microprocessor.

We first describe the most basic types of microprocessor registers, their functions, and how they are used. We then consider the other common types of registers.

Basic Microprocessor Registers

There are four basic microprocessor registers: instruction register, program counter, memory address register, and accumulator.

- **Instruction Register (IR).** The instruction register stores instructions. The contents of an instruction register are always decoded by the microprocessor as an instruction. After fetching an instruction code from memory, the microprocessor stores it in the instruction register. The instruction is decoded internally by the microprocessor, which then performs the required operation. The word size of the microprocessor determines the size of the instruction register. For example, a 16-bit microprocessor has a 16-bit instruction register.
- **Program Counter (PC).** The program counter contains the address of the instruction or operation code (op-code). The program counter normally contains the address of the next instruction to be executed. Note the following features of the program counter:
  1. Upon activating the microprocessor's RESET input, the address of the first instruction to be executed is loaded into the program counter.
  2. To execute an instruction, the microprocessor typically places the contents of the program counter on the address bus and reads ("fetches") the contents of this address, that is, instruction, from memory. The program counter contents are automatically incremented by the microprocessor's internal logic. The microprocessor thus executes a program sequentially, unless the program contains an instruction such as a `JUMP` instruction, which changes the sequence.
  3. The size of the program counter is determined by the size of the address bus.

4. Many instructions, such as JUMP and conditional JUMP, change the contents of the program counter from its normal sequential address value. The program counter is loaded with the address specified in these instructions.

- **Memory Address Register (MAR).** The memory address register contains the address of data. The microprocessor uses the address, which is stored in the memory address register, as a direct pointer to memory. The contents of the address consists of the actual data that is being transferred.
- **Accumulator (A).** For an 8-bit microprocessor, the accumulator is typically an 8-bit register. It is used to store the result after most ALU operations. These microprocessors have instructions to shift or rotate the accumulator 1 bit to the right or left through the carry flag. The accumulator is typically used for inputting a byte into the accumulator from an external device or outputting a byte to an external device from the accumulator. Some microprocessors, such as the Motorola 6809, have more than one accumulator. In these microprocessors, the accumulator to be used by the instruction is specified in the op-code.

Depending on the register section, the microprocessor can be classified either as an accumulator-based or a general-purpose register-based machine. In an accumulator-based microprocessor such as the Intel 8085 and Motorola 6809, the data is assumed to be held in a register called the "accumulator." All arithmetic and logic operations are performed using this register as one of the data sources. The result after the operation is stored in the accumulator. Eight-bit microprocessors are usually accumulator based.

The general-purpose register-based microprocessor is usually popular with 16-, 32-, and 64-bit microprocessors, such as the Intel 8086/80386/80486/Pentium and the Motorola 68000 /68020 /68030 /68040 /PowerPC. The term "general-purpose" comes from the fact that these registers can hold data, memory addresses, or the results of arithmetic or logic operations. The number, size, and types of registers vary from one microprocessor to another.

Most registers are general-purpose whereas some, such as the program counter (PC), are provided for dedicated functions. The PC normally contains the address of the next instruction to be executed. As metioned before, upon activating the microprocessor chi p's RESET input pin, the PC is normally initialized with the address of the first instruction. For example, the 80486, upon hardware reset, reads the first instruction from the 32-bit hex address FFFFFFF0. To execute the instruction, the microprocessor normally places the PC contents on the address bus and reads (fetches) the first instruction from external memory. The program counter contents are then automatically incremented by the ALU. The microcomputer thus usually executes a program sequentially unless it encounters a jump or branch instruction. As mentioned earlier, the size of the PC varies from one microprocessor to another depending on the address size. For example, the 68000 has a 24-bit PC, whereas the 68040 contains a 32-bit PC. Note that in general-purpose register-based microprocessors, the four basic registers typically include a PC, an MAR, an IR, and a data register.

Use of the Basic Microprocessor Registers

To provide a clear understanding of how the basic microprocessor registers are used, a binary addition program will be considered. The program logic will be explained by showing how each instruction changes the contents of the four registers. Assume that all numbers are in hex. Suppose that the contents of the memory location 2010 are to be added with the contents of 2012. Assume that [NNNN] represents the contents of the memory location NNNN. Now, suppose that [2010] = 0002 and [2012] = 0005. The steps involved in accomplishing this addition can be summarized as follows:

1. Load the memory address register (MAR) with the address of the first data item to be added, that is, load 2010 into MAR.
2. Move the contents of this address to a data register, D0; that is, move first data into D0.
3. Increment the MAR by 2 to hold 2012, the address of the second data item to be added.
4. Add the contents of this memory location to the data that was moved to the data register, D0 in step 2, and store the result in the 16-bit data register, D0. The above addition program will be written using 68000 instructions. Note that the 68000 uses 24-bit addresses; 24-bit addresses such as 002000_{16} will be represented as 2000_{16} (16-bit number) in the following.

The following steps will be used to achieve this addition for the 68000:

1. Load the contents of the next 16-bit memory word into the memory address register, A1. Note that register A1 can be considered as MAR in the 68000.
2. Read the 16-bit contents of the memory location addressed by MAR into data register, D0.
3. Increment MAR by 2 to hold 2012, the address of the second data to be added.
4. Add the current contents of data register, D0 to the contents of the memory location whose address is in MAR and store the 16-bit result in D0.

The following steps for the Motorola 68000 will be used to achieve the above addition:

3279_{16}	Load the contents of the next 16-bit memory word into the memory address register, A1.
3010_{16}	Read the 16-bit contents of the memory location addressed by MAR into data register, D0.
5249_{16}	Increment MAR by 2.
$D051_{16}$	Add the current contents of data register, D0, to the contents of the memory location whose address is in MAR and store the 16-bit result in D0.

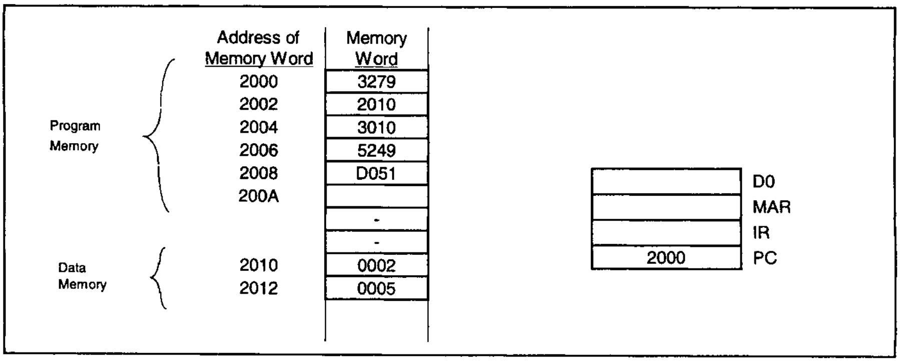
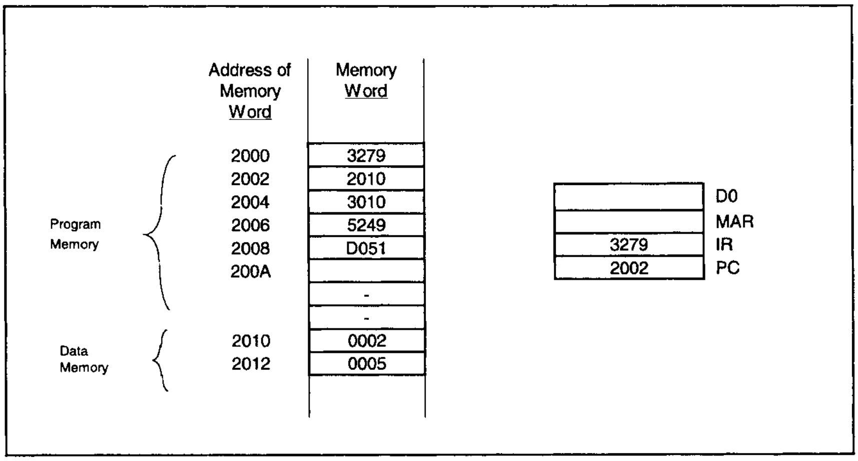


FIGURE 6.5 Microprocessor addition program with initial register and memory

The complete program in hexadecimal, starting at location 2000_{16} (arbitrarily chosen) is given in Figure 6.5. Note that each memory address stores 16 bits. Hence, memory addresses are shown in increments of 2. Assume that the microcomputer can be instructed that the starting address of the program is 2000_{16}. This means that the program counter can be initialized to contain 2000_{16}, the address of the first instruction to be executed. Note that the contents of the other three registers are not known at this point. The microprocessor loads the contents of memory location addressed by the program counter into IR. Thus, the first instruction, 3279_{16}, stored in address 2000_{16} is transferred into IR.

The program counter contents are then incremented by 2 by the microprocessor's ALU to hold 2002_{16}. The register contents that result along with the program are shown in Figure 6.6.

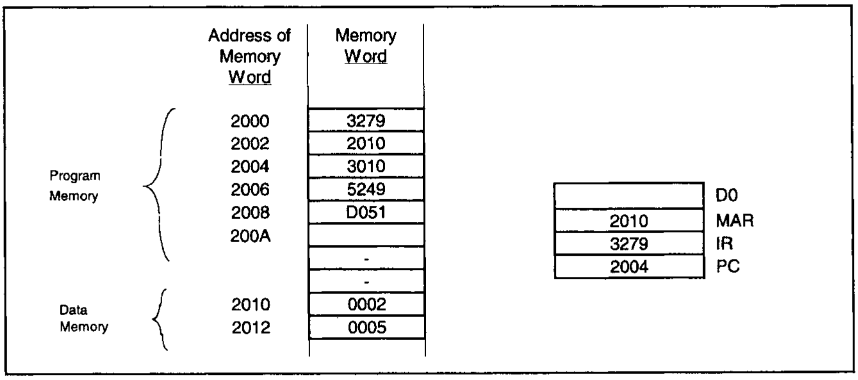
The binary code 3279_{16} in the IR is executed by the microprocessor. The microprocessor then takes appropriate actions. Note that the instruction, 3279_{16}, loads the contents of the next memory location addressed by the PC into the MAR. Thus, 2010_{16} is loaded into the MAR. The contents of the PC are then incremented by 2 to hold 2004_{16}. This is shown in Figure 6.7.



	Address of Memory Word	Memory Word
Program Memory	2000	3279
	2002	2010
	2004	3010
	2006	5249
	2008	D051
	200A	
		-
		-
Data Memory	2010	0002
	2012	0005

	D0
	MAR
3279	IR
2002	PC

FIGURE 6.6 Microprocessor addition program (modified during execution)



	Address of Memory Word	Memory Word
Program Memory	2000	3279
	2002	2010
	2004	3010
	2006	5249
	2008	D051
	200A	
		-
		-
Data Memory	2010	0002
	2012	0005

	D0
2010	MAR
3279	IR
2004	PC

FIGURE 6.7 Microprocessor addition program (modified during execution)

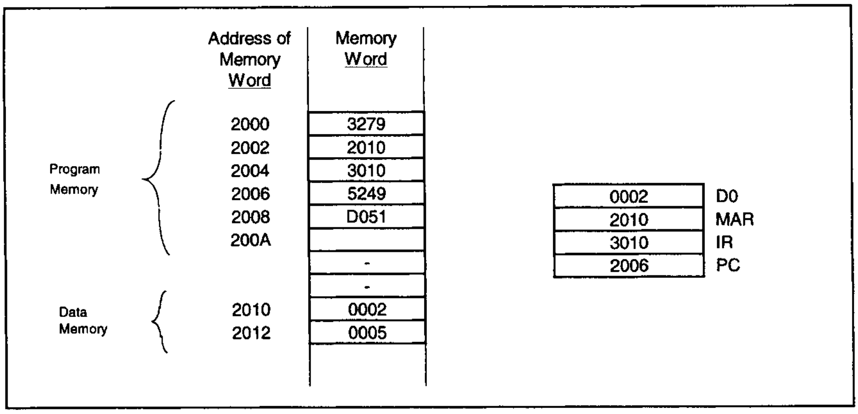


FIGURE 6.8 Microprocessor addition program (modified during execution)

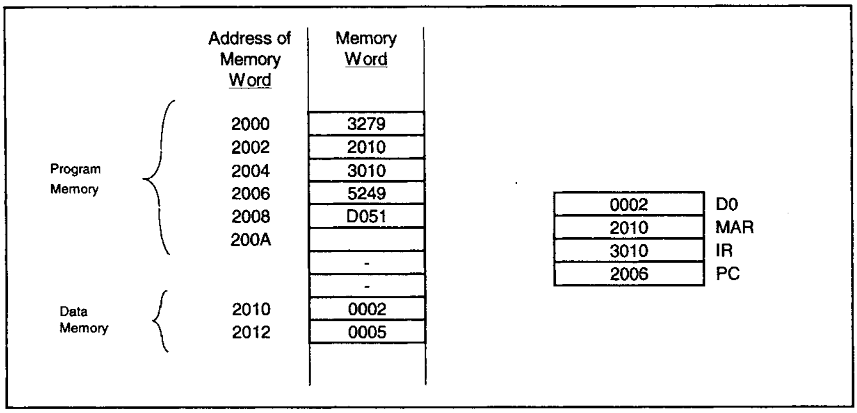


FIGURE 6.9 Microprocessor addition program (modified during execution)

Next, the microprocessor loads the contents of the memory location addressed by the PC into the IR; thus, 3010_{16} is loaded into the IR. The PC contents are then incremented by 2 to hold 2006_{16}. This is shown in Figure 6.8. In response to the instruction 3010_{16}, the contents of the memory location addressed by the MAR are loaded into the data register, D0; thus, 0002_{16} is moved to register D0. The contents of the PC are not incremented this time. This is because 0002_{16} is not immediate data. Figure 6.9 shows the details. Next the microprocessor loads 5249_{16} to IR and then increments PC to contain 2008_{16} as shown in Figure 6.10.

In response to the instruction 5249_{16} in the IR, the microprocessor increments the MAR by 2 to contain 2012_{16} as shown in Figure 6.11. Next, the instruction $D051_{16}$ in location 2008_{16} is loaded into the IR, and the PC is then incremented by 2 to hold $200A_{16}$ as shown in Figure 6.12. Finally, in response to instruction $D051_{16}$, the microprocessor adds the contents of the memory location addressed by MAR (address 2012_{16}) with the contents of register D0 and stores the result in D0. Thus, 0002_{16} is added with 0005_{16}, and the 16-bit result 0007_{16} is stored in D0 as shown in Figure 6.13. This completes the execution of the binary addition program.

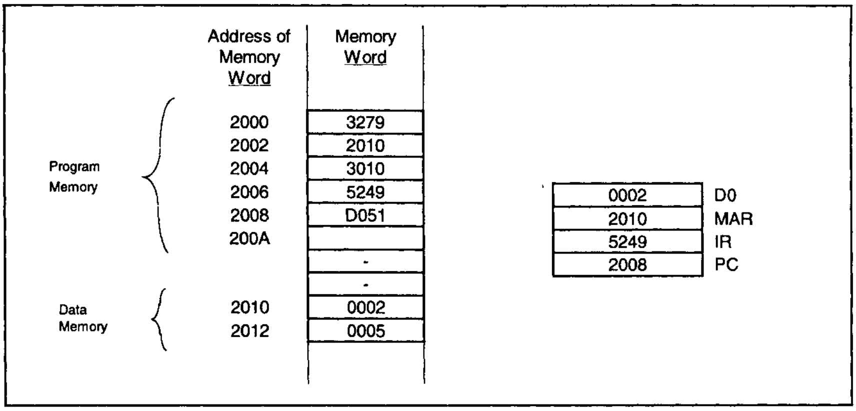


FIGURE 6.10 Microprocessor addition program (modified during execution)

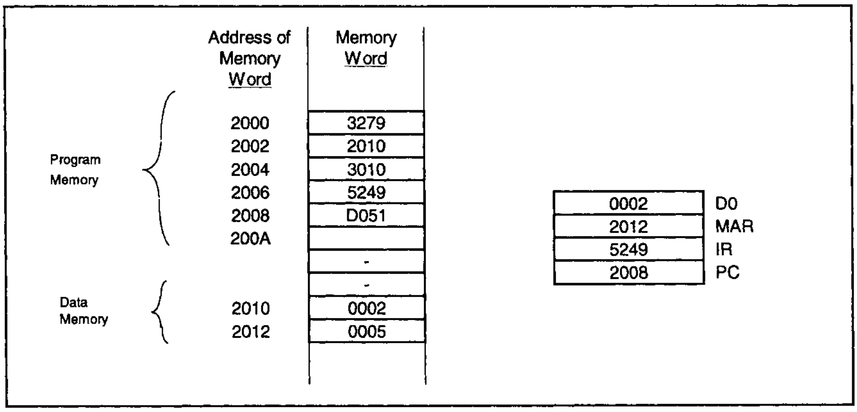


FIGURE 6.11 Microprocessor addition program (modified during execution)

Other Microprocessor Registers

- **General-Purpose Registers**

The 16-, 32-, and 64-bit microprocessors are register oriented. They have a number of general-purpose registers for storing temporary data or for carrying out data transfers between various registers. The use of general-purpose registers speeds up the execution of a program because the microprocessor does not have to read data from external memory via the data bus if data is stored in one of its general-purpose registers. These registers are typically 16 to 32 bits. The number of general-purpose registers will vary from one microprocessor to another. Some of the typical functions performed by instructions associated with the general-purpose registers are given here. We will use [REG] to indicate the contents of the general-purpose register and [M] to indicate the contents of a memory location.

1. Move [REG] to or from memory: [M] ← [REG] or [REG] ← [M].
2. Move the contents of one register to another: [REG1] ← [REG2].
3. Increment or decrement [REG] by 1: [REG] ← [REG] + 1 or [REG] ← [REG] - 1.
4. Load 16-bit data into a register [REG] : [REG] ← 16-bit data.

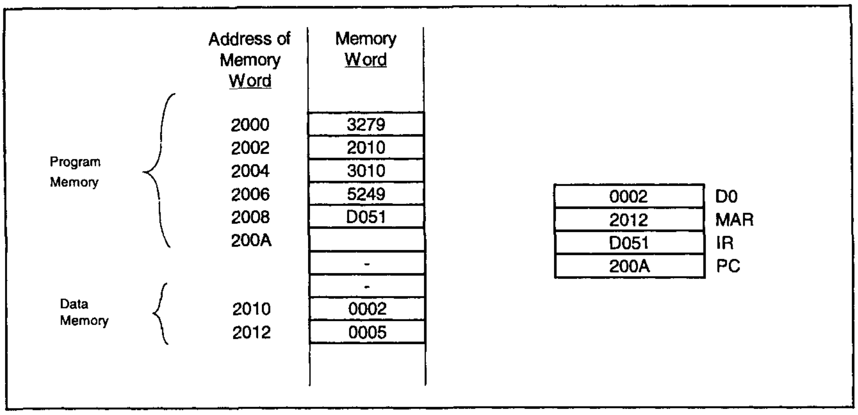


FIGURE 6.12 Microprocessor addition program (modified during execution)

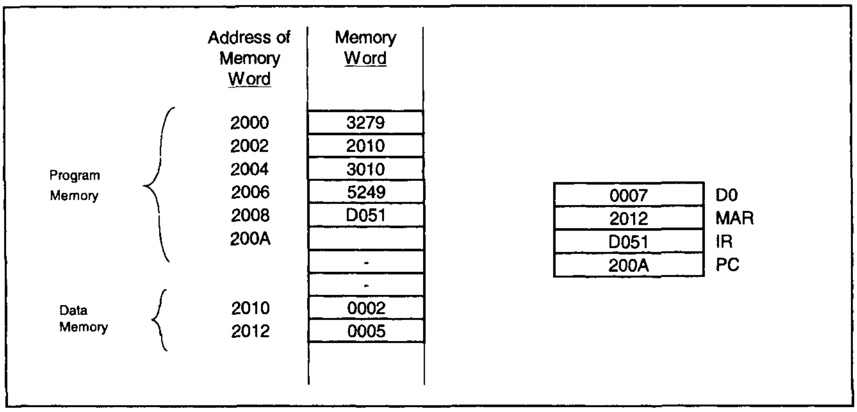


FIGURE 6.13 Microprocessor addition program (modified during execution)

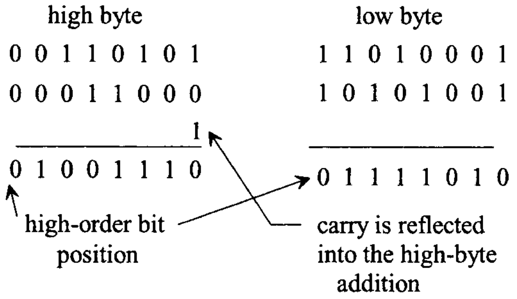
- **Index Register**

  An *index register* is typically used as a counter in address modification for an instruction, or for general storage functions. The index register is particularly useful with instructions that access tables or arrays of data. In this operation the index register is used to modify the address portion of the instruction. Thus, the appropriate data in a table can be accessed. This is called "indexed addressing." This addressing mode is normally available to the programmers of microprocessors. The effective address for an instruction using the indexed addressing mode is determined by adding the address portion of the instruction to the contents of the index register. Index registers are typically 16 or 32 bits long. In a typical 16- or 32-bit microprocessor, general-purpose registers can be used as index registers.

- **Status Register**

  The *status register,* also known as the "processor status word register" or the "condition code register," contains individual bits, with each bit having special significance. The bits in the status register are called "flags." The status of a specific microprocessor operation is indicated by each flag, which is set or reset by the microprocessor's internal logic to indicate the status of certain microprocessor operations such as arithmetic and logic operations. The status flags are also used in conditional `JUMP` instructions. We will describe some of the common flags in the following.

The *carry flag* is used to reflect whether or not the result generated by an arithmetic operation is greater than the microprocessor's word size. As an example, the addition of two 8-bit numbers might produce a carry. This carry is generated out of the eighth position, which results in setting the carry flag. However, the carry flag will be zero if no carry is generated from the addition. As mentioned before, in multibyte arithmetic, any carry out of the low-byte addition must be added to the high-byte addition to obtain the correct result. This can illustrated by the following example:



While performing BCD arithmetic with microprocessors, the carry out of the low nibble (4 bits) has a special significance. Because a BCD digit is represented by 4 bits, any carry out of the low 4 bits must be propagated into the high 4 bits for BCD arithmetic. This carry flag is known as the *auxiliary carry flag* and is set to 1 if the carry out of the low 4 bits is 1, otherwise it is 0.

A *zero flag* is used to show whether the result of an operation is zero. It is set to 1 if the result is zero, and it is reset to 0 if the result is nonzero. A *parity flag* is set to 1 to indicate whether the result of the last operation contains either an even number of 1's (even parity) or an odd number of 1's (odd parity), depending on the microprocessor. The type of parity flag used (even or odd) is determined by the microprocessor's internal structure and is not selectable. The sign flag (also sometimes called the negative flag) is used to indicate whether the result of the last operation is positive or negative. If the most significant bit of the last operation is 1, then this flag is set to 1 to indicate that the result is negative. This flag is reset to 0 if the most significant bit of the result is zero, that is, if the result is positive.

As mentioned before, the *overflow flag* arises from the representation of the sign flag by the most significant bit of a word in signed binary operation. The overflow flag is set to 1 if the result of an arithmetic operation is too big for the microprocessor's maximum word size, otherwise it is reset to 0. Let C_f be the final carry out of the most significant bit (sign bit) and C_p be the previous carry. It was shown in Chapter 2 that the overflow flag is the exclusive OR of the carries C_p and C_f.

$$\text{Overflow} = C_p \oplus C_f$$

- **Stack Pointer Register**

The *stack* consists of a number of RAM locations set aside for reading data from or writing data into these locations and is typically used by subroutines (a subroutine is a program that performs operations frequently needed by the main or calling program). The address of the stack is contained in a register called the "stack pointer." Two instructions, PUSH and POP, are usually available with the stack. The PUSH operation

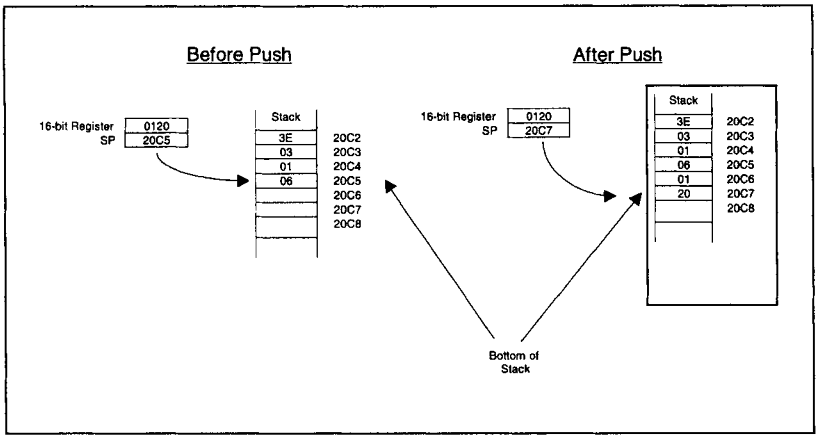


FIGURE 6.14 PUSH operation when accessing stack from bottom

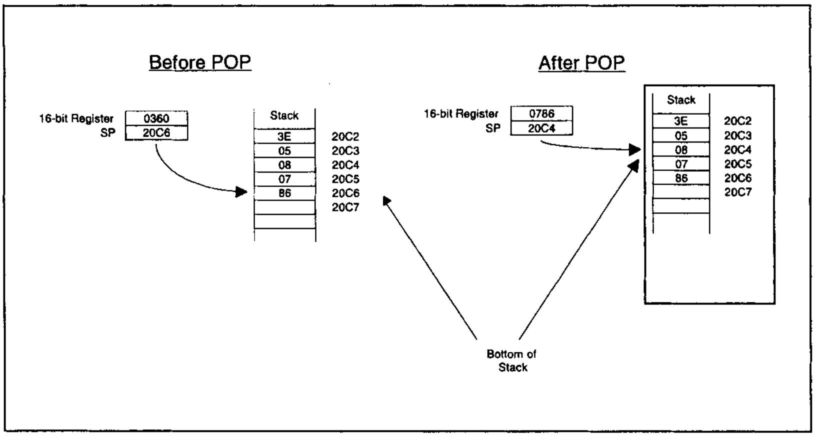


FIGURE 6.15 POP operation when accessing stack from bottom

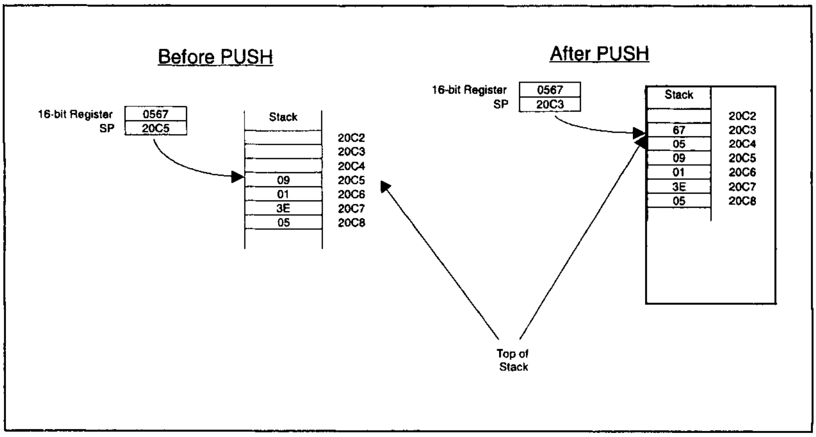


FIGURE 6.16 PUSH operation when accessing stack from top

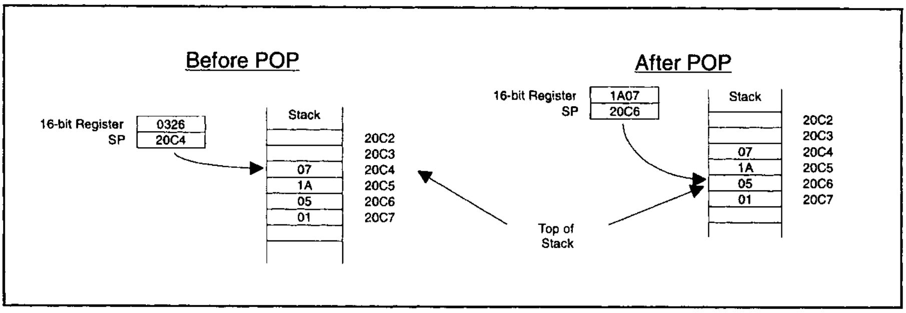


FIGURE 6.17 POP operation when accessing stack from top

is defined as writing to the top or bottom of the stack, whereas the POP operation means reading from the top or bottom of the stack. Some microprocessors access the stack from the top; the others access via the bottom. When the stack is accessed from the bottom, the stack pointer is incremented after a PUSH and decremented after a POP operation. On the other hand, when the stack is accessed from the top, the stack pointer is decremented after a PUSH and incremented after a POP. Microprocessors typically use 16- or 32-bit registers for performing the PUSH or POP operations. The incrementing or decrementing of the stack pointer depends on whether the operation is PUSH or POP and also whether the stack is accessed from the top or the bottom.

We now illustrate the stack operations in more detail. We use 16-bit registers in Figures 6.14 and 6.15. In Figure 6.14, the stack pointer is incremented by 2 (since 16-bit register) to address location 20C7 after the PUSH. Now consider the POP operation of Figure 6.15. Note that after the POP, the stack pointer is decremented by 2. [20C5] and [20C6] are assumed to be empty conceptually after the POP operation. Finally, consider the PUSH operation of Figure 6.16. The stack is accessed from the top. Note that the stack pointer is decremented by 2 after a PUSH. Next, consider the POP (Figure 6.17). [20C4] and [20C5] are assumed to be empty after the POP.

Note that the stack is a LIFO (Last In First Out) memory.

Example 6.1

Determine the carry (*C*), sign (*S*), zero (*Z*), overflow (*V*), and parity (*P*) flags for the following operation: 0110_2 plus 1010_2.

Assume the parity bit = 1 for ODD parity in the result; otherwise the parity bit = 0. Also, assume that the numbers are signed. Draw a logic diagram for implementing the flags in a 5-bit register using D flip-flops; use P = bit 0, V = bit 1, Z = bit 2, S = bit 3, and C = bit 4. Note that Verilog and VHDL descriptions along with simulation results of this status register are provided in Appendices I and J respectively.

Solution

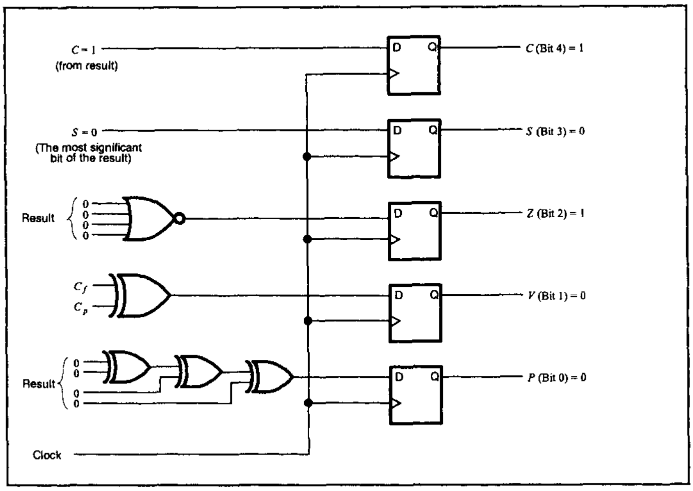
1 1 0 ← Intermediate Carries
0 1 1 0
+ 1 0 1 0

Result = 0 0 0 0

$C_f = C = 1$ ←
$S = 0$ ←
$C_p = 1$ ←

$Z = 1$ since result = 0
$P = 0$ since even parity
$V = C_f \oplus C_p = 1 \oplus 1 = 0$

The flag register can be implemented from the 4-bit result as follows:



6.3.2 Control Unit

The main purpose of the control unit is to read and decode instructions from the program memory. To execute an instruction, the control unit steps through the appropriate blocks of the ALU based on the op-codes contained in the instruction register. The op-codes define the operations to be performed by the control unit in order to execute an instruction. The control unit interprets the contents of the instruction register and then responds to the instruction by generating a sequence of enable signals. These signals activate the appropriate ALU logic blocks to perform the required operation.

The control unit generates the *control signals*, which are output to the other microcomputer elements via the control bus. The control unit also takes appropriate actions in response to the control signals on the control bus provided by the other microcomputer elements.

The control signals vary from one microprocessor to another. For each specific microprocessor, these signals are described in detail in the manufacturer's manual. It is impossible to describe all the control signals for various manufacturers. However, we cover some of the common ones in the following discussion.

- **RESET.** This input is common to all microprocessors. When this input pin is driven to HIGH or LOW (depending on the microprocessor), the program counter is loaded with a predefined address specified by the manufacturer. For example, in the 80486, upon hardware reset, the program counter is loaded with $FFFFFFF0_{16}$. This means that the instruction stored at memory location $FFFFFFF0_{16}$ is executed first. In some other microprocessors, such as the Motorola 68000, the program counter is not loaded directly by activating the RESET input. In this case, the program counter is loaded indirectly from two locations (such as 000004 and 000006) predefined by the manufacturer. This means that these two locations contain the address of the first instruction to be executed.
- **READ/WRITE ($R/\overline{W}$).** This output line is common to all microprocessors. The status of this line tells the other microcomputer elements whether the microprocessor

is performing a READ or a WRITE operation. A HIGH signal on this line indicates a READ operation and a LOW indicates a WRITE operation. Some microprocessors have separate READ and WRITE pins.

- $\overline{\textbf{READY}}$. This is an input to the microprocessor. Slow devices (memory and I/O) use this signal to gain extra time to transfer data to or receive data from a microprocessor. The $\overline{\text{READY}}$ signal is usually an active low signal, that is, LOW means that the microprocessor is ready. Therefore, when the microprocessor selects a slow device, the device places a LOW on the $\overline{\text{READY}}$ pin. The microprocessor responds by suspending all its internal operations and enters a WAIT state. When the device is ready to send or receive data, it removes the $\overline{\text{READY}}$ signal. The microprocessor comes out of the WAIT state and performs the appropriate operation.
- **Interrupt Request (INT or IRQ).** The external I/O devices can interrupt the microprocessor via this input pin on the microprocessor chip. When this signal is activated by the external devices, the microprocessor jumps to a special program, called the "interrupt service routine." This program is normally written by the user for performing tasks that the interrupting device wants the microprocessor to do. After completing this program, the microprocessor returns to the main program it was executing when the interrupt occurred.

6.3.3 Arithmetic and Logic Unit (ALU)

The ALU performs all the data manipulations, such as arithmetic and logic operations, inside the microprocessor. The size of the ALU conforms to the word length of the microcomputer. This means that a 32-bit microprocessor will have a 32-bit ALU. Typically, the ALU performs the following functions:

1. Binary addition and logic operations
2. Finding the ones complement of data
3. Shifting or rotating the contents of a general-purpose register 1 bit to the left or right through carry

6.3.4 Functional Representations of a Simple and a Typical Microprocessor

Figure 6.18 shows the functional block diagram of a simple microprocessor. Note that the

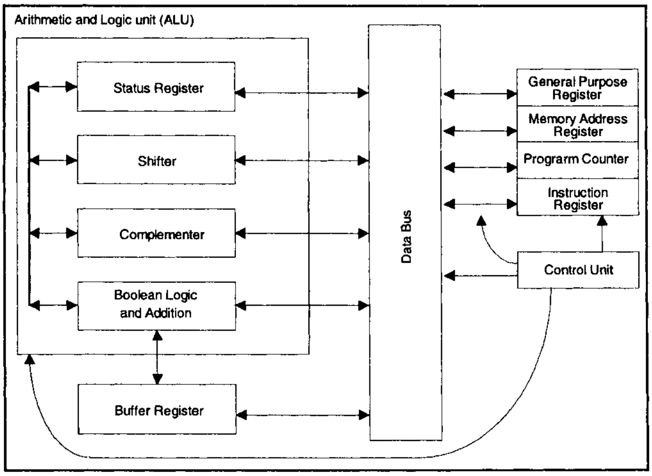


FIGURE 6.18 Functional representation of a simple microprocessor

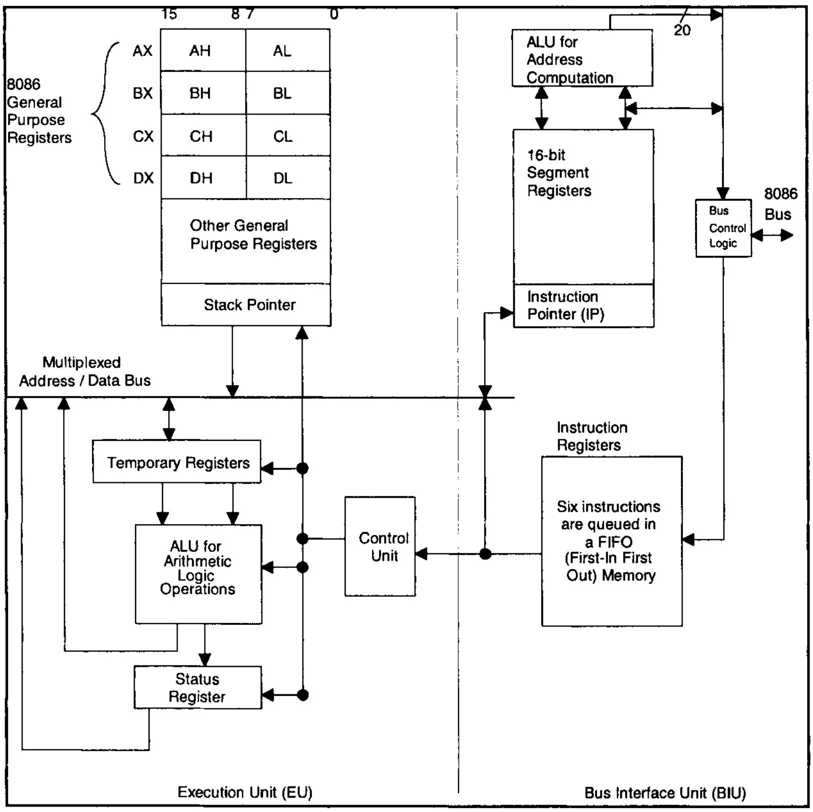


FIGURE 6.19 Simplified block diagram of the 8086

data bus shown is internal to the microprocessor chip and should not be confused with the system bus. The system bus is external to the microprocessor and is used to connect all the necessary chips to form a microcomputer. The buffer register in Figure 6.18 stores any data read from memory for further processing by the ALU. All other blocks of Figure 6.18 have been discussed earlier. Figure 6.19 shows the simplified block diagram of a realistic microprocessor, the Intel 8086.

The 8086 microprocessor is internally divided into two functional units: the bus interface unit (BIU) and the execution unit (EU). The BIU interfaces the 8086 to external memory and I/O chips. The BIU and EU function independently. The BIU reads (fetches) instructions and writes or reads data to or from memory and I/O ports. The EU executes instructions that have already been fetched by the BIU. The BIU contains segment registers, the instruction pointer (IP), the instruction queue registers, and the address generation/bus control circuitry.

The 8086 uses segmented memory. This means that the 8086's 1 MB main memory is divided into 16 segments of 64 KB each. Within a particular segment, the instruction pointer (IP) works as a program counter (PC). Both the IP and the segment registers are 16 bits wide. The 20-bit address is generated in the BIU by using the contents of a 16-bit IP and a 16-bit segment register. The ALU in the BIU is used for this purpose. Memory segmentation is useful in a time-shared system when several users share a microprocessor. Segmentation makes it easy to switch from one user program to another by changing the contents of a segment register.

The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and I/O. The BIU's instruction register consist of a first-in–first-out (FIFO) memory in which up to six instruction bytes are preread (prefetched) from external memory ahead of time to speed up instruction execution. The control unit in the EU translates the instructions based on the contents of the instruction registers in the BIU.

The EU contains several 16-bit general-purpose registers. Some of them are AX, BX, CX, and DX. Each of these registers can be used either as an 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) or as a 16-bit register (AX, BX, CX, DX). Register BX can also be used to hold the address in a segment. The EU also contain a 16-bit status register. The ALU in the EU performs all arithmetic and logic operations. The 8086 is covered in detail in Chapter 9.

6.3.5 Microprogramming the Control Unit (A Simplified Explanation)

In this section, we discuss how the op-codes are interpreted by the microprocessor. Most microprocessors have an internal memory, called the "control memory" (ROM). This memory is used to store a number of codes, called the "microinstructions." These microinstructions are combined together to design instructions. Each instruction in the instruction register initiates execution of a set of microinstructions in the control unit to perform the operation required by the instruction. The microprocessor manufacturers define the microinstructions by programming the control memory (ROM) and thus, design the instruction set of the microprocessor. This type of programming is known as "microprogramming." Note that the control units of most 16-, 32-, and 64-bit microprocessors are microprogrammed.

For simplicity, we illustrate the concepts of microprogramming using Figure 6.18. Let us consider incrementing the contents of the register. This is basically an addition operation. The control unit will send an enable signal to execute the ALU adder logic.

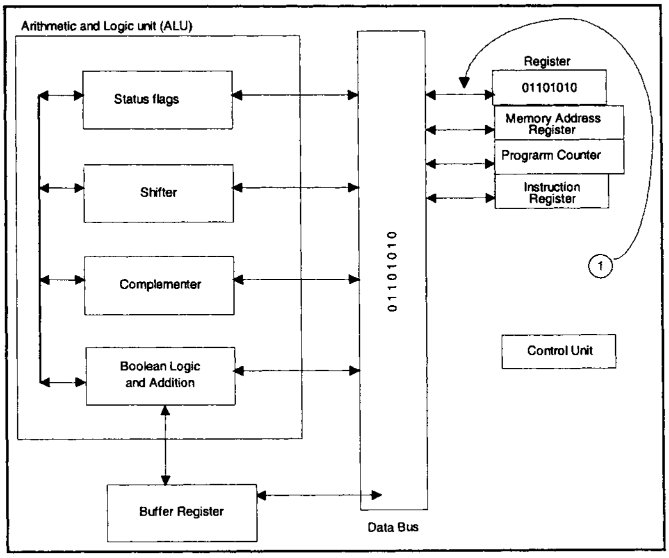


FIGURE 6.20 Transferring register contents to data bus

Incrementing the contents of a register consists of transferring the register contents to the ALU adder and then returning the result to the register. The complete incrementing process is accomplished via the five steps shown in Figures 6.20 through Figure 6.24. In all five steps, the control unit initiates execution of each microinstruction. Figure 6.20 shows the transfer of the register contents to the data bus. Figure 6.21 shows the transfer of the contents of the data bus to the adder in the ALU in order to add 1 to it. Figure 6.22 shows the activation of the adder logic. Figure 6.23 shows the transfer of the result from the adder to the data bus. Finally, Figure 6.24 shows the transfer of the data bus contents to the register.

Microprogramming is typically used by the microprocessor designer to program the logic performed by the control unit. On the other hand, assembly language programming is a popular programming language used by the microprocessor user for programming the microprocessor to perform a desired function. A microprogram is stored in the control unit. An assembly language program is stored in the main memory. The assembly language program is called a macroprogram. A macroinstruction (or simply an instruction) initiates execution of a complete microprogram.

A simplified explanation of microprogramming is provided in this section. This topic will be covered in detail in Chapter 7.

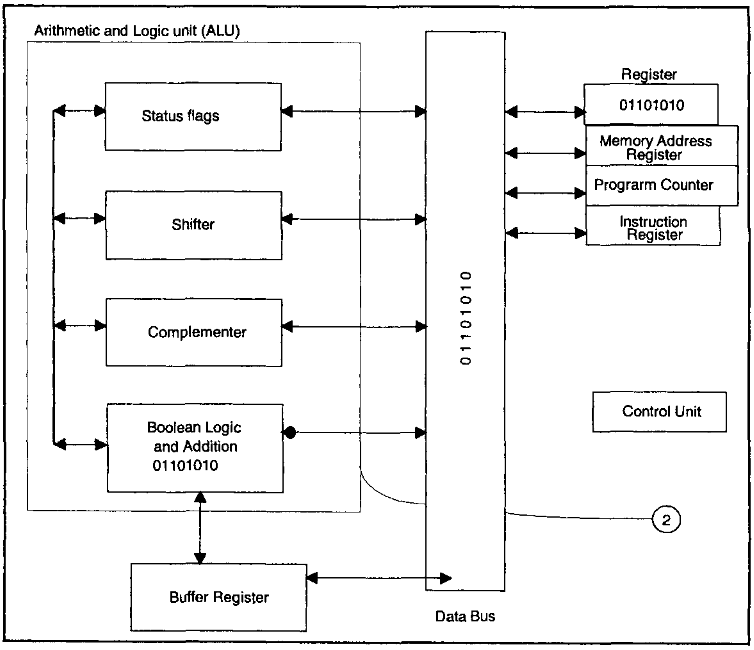


FIGURE 6.21 Transferring data bus contents to the ALU

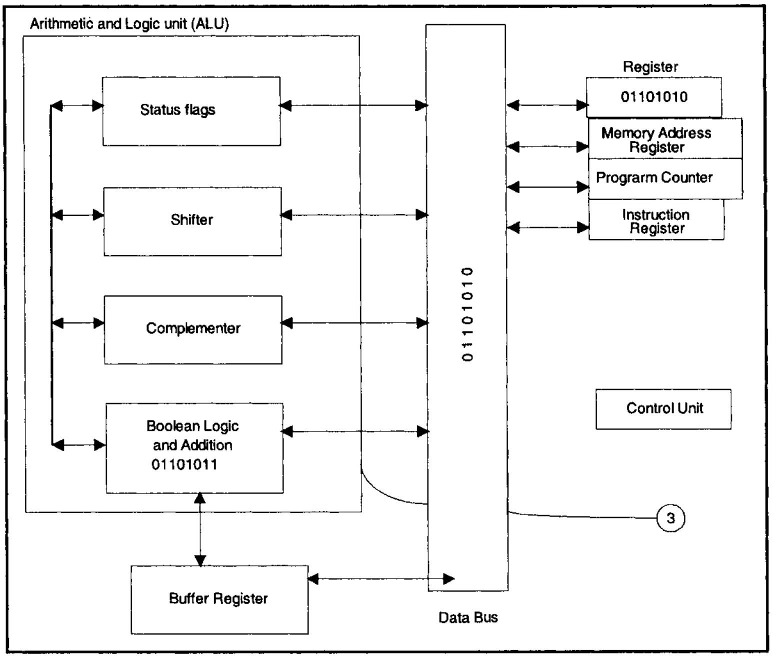


FIGURE 6.22 Activating the ALU logic

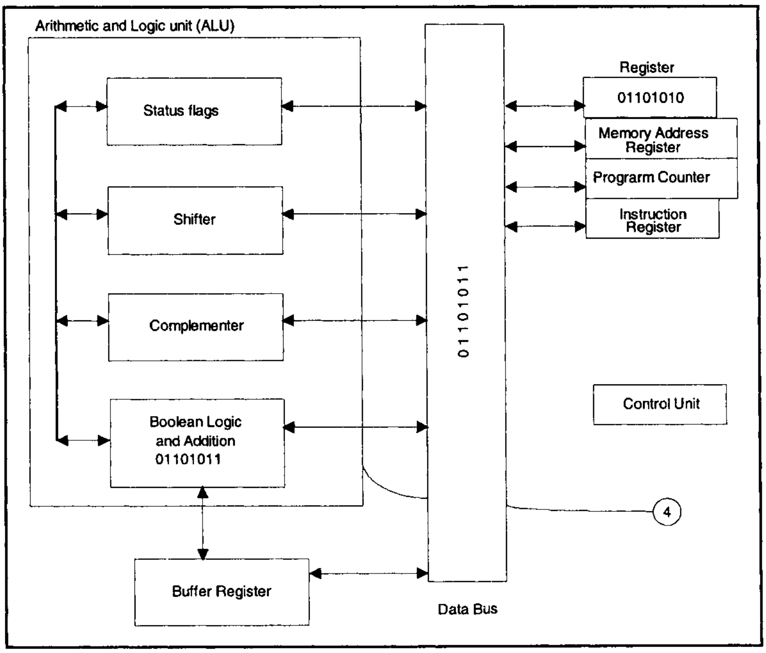


FIGURE 6.23 Transferring the ALU result to the data bus

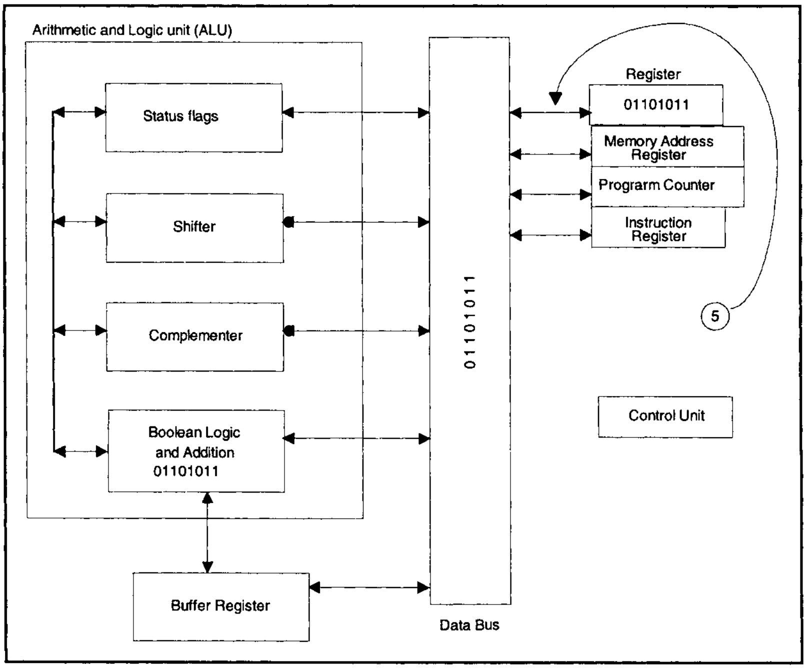


FIGURE 6.24 Transferring the data bus

6.4 The Memory

The main or external memory (or simply the memory) stores both instructions and data. For 8-bit microprocessors, the memory is divided into a number of 8-bit units called "memory words." An 8-bit unit of data is termed a "byte." Therefore, for an 8-bit microprocessor, "memory word" and "memory byte" mean the same thing. For 16-bit microprocessors, a word contains two bytes (16 bits). A memory word is identified in the memory by an address. For example, the 8086 microprocessor uses 20-bit addresses for accessing

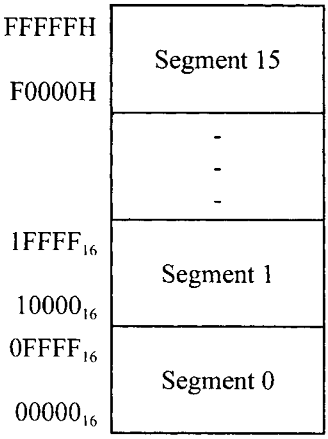


FIGURE 6.25 The main memory of the 8086

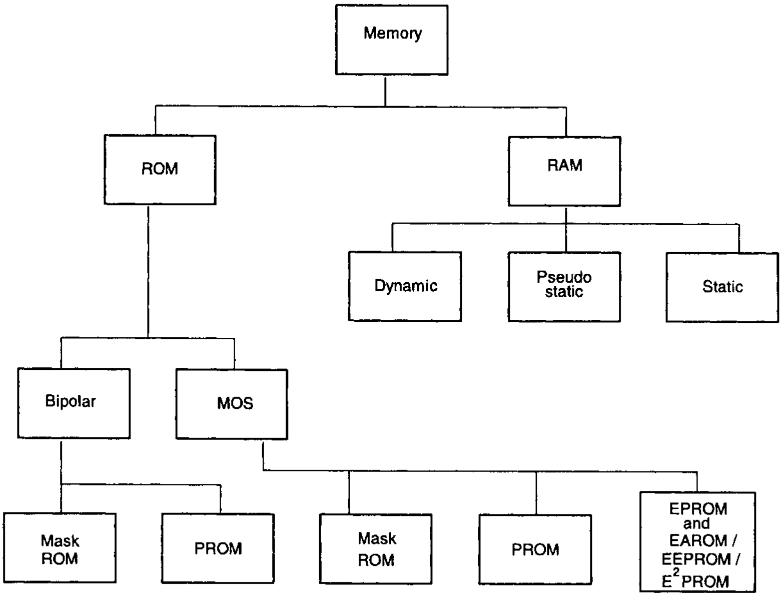


FIGURE 6.26 Summary of available semiconductor memories for microprocessor systems

memory words. This provides a maximum of 2^{20} = 1 MB of memory addresses, ranging from 00000_{16} to $FFFFF_{16}$ in hexadecimal.

As mentioned before, an important characteristic of a memory is whether it is volatile or nonvolatile. The contents of a volatile memory are lost if the power is turned off. On the other hand, a nonvolatile memory retains its contents after power is switched off. Typical examples of nonvolatile memory are ROM and magnetic memory (floppy disk). A RAM is a volatile memory unless backed up by battery.

As mentioned earlier, some microprocessors such as the Intel 8086 divide the memory into segments. For example, the 8086 divides the 1 MB main memory into 16 segments (0 through 15). Each segment contains 64 KB of memory and is addressed by 16 bits. Figure 6.25 shows a typical main memory layout of the 8086. In the figure, the high four bits of an address specify the segment number. As an example, consider address 10005_{16} of segment 1. The high four bits, 0001, of this address define the location is in segment 1 and the low 16 bits, 0005_{16}, specify the particular address in segment 1. The 68000, on the other hand, uses linear or nonsegmented memory. For example, the 68000 uses 24 address pins to directly address 2^{24} = 16 MB of memory with addresses from 000000_{16} to $FFFFFF_{16}$. As mentioned before, memories can be categorized into two main types: read-only memory (ROM) and random-access memory (RAM). As shown in Figure 6.26, ROMs and RAMs are then divided into a number of subcategories, which are discussed next.

6.4.1 Random-Access Memory (RAM)

There are three types of RAM: dynamic RAM, pseudo-static RAM , and static RAM. Dynamic RAM stores data in capacitors, that is, it can hold data for a few milliseconds. Hence, dynamic RAMs are refreshed typically by using external refresh circuitry. Pseudo-static RAMs are dynamic RAMs with internal refresh. Finally, static RAM stores data in flip-flops. Therefore, this memory does not need to be refreshed. RAMs are volatile unless backed up by battery. Dynamic RAMs (DRAMs) are used in applications requiring large memory. DRAMs have higher densities than Static RAMs (SRAMs). Typical examples of DRAMs are 4464 (64K x 4-bit), 44256 (256K x 4-bit), and 41000 (1M x 1-bit). DRAMs are inexpensive, occupy less space , and dissipate less power compared to SRAMs. Two enhanced versions of DRAM are EDO DRAM (Extended Data Output DRAM) and SDRAM (Synchronous DRAM). The EDO DRAM provides fast access by allowing the DRAM controller to output the next address at the same time the current data is being read. An SDRAM contains multiple DRAMs (typically 4) internally. SDRAMs utilize the multiplexed addressing of conventional DRAMs . That is, SDRAMs provide row and column addresses in two steps like DRAMs. However, the control signals and address inputs are sampled by the SDRAM at the leading edge of a common clock signal (133 MHz maximum). SDRAMs provide higher densities by further reducing the need for support circuitry and faster speeds than conventional DRAMs. The SDRAM has become popular with PC (Personal Computer) memory.

6.4.2 Read-Only Memory (ROM)

ROMs can only be read. This memory is nonvolatile. From the technology point of view, ROMs are divided into two main types, bipolar and MOS. As can be expected, bipolar ROMs are faster than MOS ROMs. Each type is further divided into two common types, mask ROM and programmable ROM. MOS ROMs contain one more type, erasable PROM (EPROM such as Intel 2732 and EAROM or EEPROM or E^2PROM such as Intel 2864). Mask ROMs are programmed by a masking operation performed on the chip during the manufacturing process. The contents of mask ROMs are permanent and cannot be changed by the user. On the other hand, the programmable ROM (PROM) can be programmed by the user by means of proper equipment. However, once this type of memory is programmed, its contents cannot be changed. Erasable PROMs (EPROMs and EAROMs) can be programmed, and their contents can also be altered by using special equipment, called the PROM programmer. When designing a microcomputer for a particular application, the permanent programs are stored in ROMs. Control memories are ROMs. PROMs can be programmed by the user. PROM chips are normally designed using transistors and fuses.

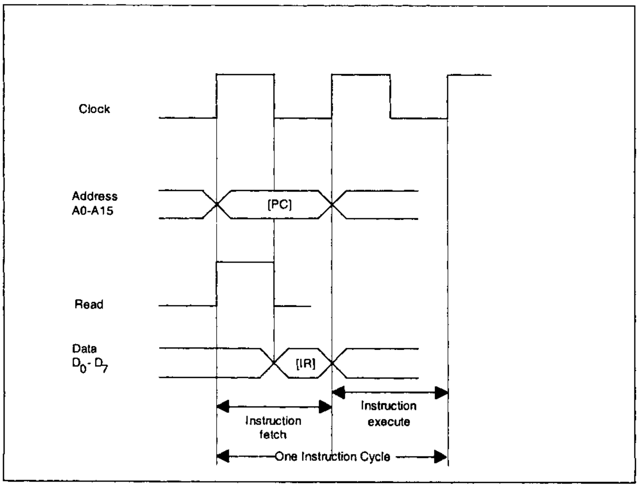


FIGURE 6.27 Typical Instruction Fetch Timing Diagram for an 8-bit Microprocessor

These transistors can be selected by addressing via the pins on the chip. In order to program this memory, the selected fuses are "blown" or "burned" by applying a voltage on the appropriate pins of the chip. This causes the memory to be permanently programmed.

Erasable PROMs (EPROMs) can be reprogrammed and erased. The chip must be removed from the microcomputer system for programming. This memory is erased by exposing the chip via a lid or window on the chip to ultraviolet light. Typical erase times vary between 10 and 30 min. The EPROM can be programmed by inserting the chip into a socket of the PROM programmer and providing proper addresses and voltage pulses at the appropriate pins of the chip. Electrically alterable ROMs (EAROMs) can be programmed without removing the memory from the ROM's sockets. These memories are also called read mostly memories (RMMs), because they have much slower write times than read times. Therefore, these memories are usually suited for operations when mostly reading rather that writing will be performed. Another type of memory called "Flash memory" (nonvolatile) invented in the mid 1980s by Toshiba is designed using a combination of EPROM and E^2PROM technologies. Flash memory can be reprogrammed electrically while being embedded on the board. One can change multiple bytes at a time. An example of Flash memory is the Intel 28F020 (256K x 8). Flash memory is typically used in cellular phones and digital cameras.

6.4.3 READ and WRITE Operations

To execute an instruction, the microprocessor reads or fetches the op-code via the data bus from a memory location in the ROM/RAM external to the microprocessor. It then places the op-code (instruction) in the instruction register. Finally, the microprocessor executes the instruction. Therefore, the execution of an instruction consists of two portions, instruction fetch and instruction execution. We will consider the instruction fetch, memory READ and memory WRITE timing diagrams in the following using a single clock signal. Figure 6.27 shows a typical instruction fetch timing diagram.

In Figure 6.27, to fetch an instruction, when the clock signal goes to HIGH, the microprocessor places the contents of the program counter on the address bus via the address pins A_0–A_{15} on the chip. Note that since each one of these lines A_0–A_{15} can be either HIGH or LOW, both transitions are shown for the address in Figure 6.27. The instruction fetch is basically a memory READ operation. Therefore, the microprocessor raises the signal

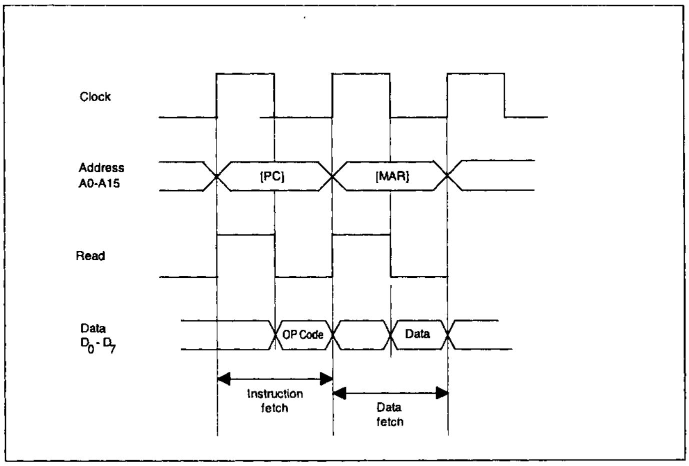


FIGURE 6.28 Typical Memory READ Timing Diagram

on the READ pin to HIGH. As soon as the clock goes to LOW, the logic external to the microprocessor gets the contents of the memory location addressed by A_0–A_{15} and places them on the data bus D_0–D_7. The microprocessor then takes the data and stores it in the instruction register so that it gets interpreted as an instruction. This is called "instruction fetch." The microprocessor performs this sequence of operations for every instruction.

We now describe the READ and WRITE timing diagrams. A typical READ timing diagram is shown in Figure 6.28. Memory READ is basically loading the contents of a memory location of the main ROM/RAM into an internal register of the microprocessor. The address of the location is provided by the contents of the memory address register (MAR). Let us now explain the READ timing diagram of Figure 6.28 as follows:

1. The microprocessor performs the instruction fetch cycle as before to READ the op-code.
2. The microprocessor interprets the op-code as a memory READ operation.
3. When the clock pin signal goes to HIGH, the microprocessor places the contents of the memory address register on the address pins A_0–A_{15} of the chip.
4. At the same time, the microprocessor raises the READ pin signal to HIGH.
5. The logic external to the microprocessor gets the contents of the location in the main ROM/RAM addressed by the memory address register and places them on the data bus.
6. Finally, the microprocessor gets this data from the data bus via its pins D_0 – D_7 and stores it in an internal register.

Memory WRITE is basically storing the contents of an internal register of the microprocessor into a memory location of the main RAM. The contents of the memory address register provide the address of the location where data is to be stored. Figure 6.29 shows a typical WRITE timing diagram. It can be explained in the following way:

1. The microprocessor fetches the instruction code as before.
2. The microprocessor interprets the instruction code as a memory WRITE instruction and then proceeds to perform the DATA STORE cycle.
3. When the clock pin signal goes to HIGH, the microprocessor places the contents of the

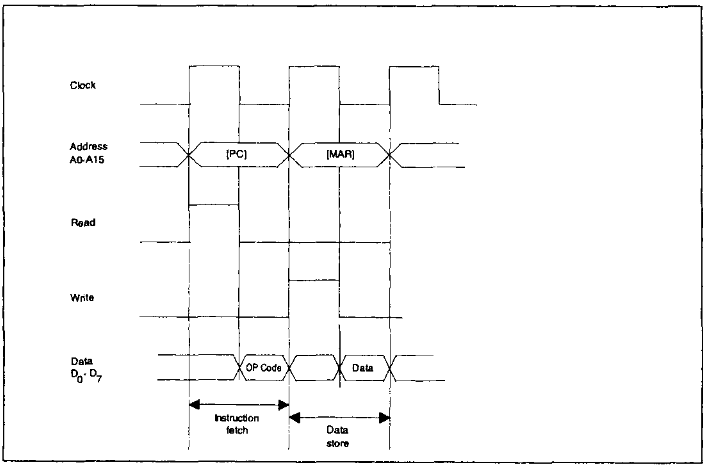


FIGURE 6.29 Typical Memory WRITE Timing Diagram

memory address register on the address pins A_0–A_{15} of the chip.

4. At the same time, the microprocessor raises the WRITE pin signal to HIGH.
5. The microprocessor places data to be stored from the contents of an internal register onto the data pins D_0–D_7.
6. The logic external to the microprocessor stores the data from the register into a RAM location addressed by the memory address register.

6.4.4 Memory Organization

Microcomputer memory typically consists of ROMs / EPROMs, and RAMs. Because RAMs can be both read from and written into, the logic required to implement RAMs is more complex than that for ROMs / EPROMs. A microcomputer system designer is normally interested in how the microcomputer memory is organized or, in other words, how to connect the ROMS /EPROMs and RAMs and then determine the memory map of the microcomputer. That is, the designer would be interested in finding out what memory locations are assigned to the ROMs / EPROMs and RAMs. The designer can then implement the permanent programs in ROMs / EPROMs and the temporary programs in RAMs. Note that RAMs are needed when subroutines and interrupts requiring stack are desired in an application.

As mentioned before, DRAMs (Dynamic RAMs) use MOS capacitors to store information and need to be refreshed. DRAMs are inexpensive compared to SRAMs, provide larger bit densities and consume less power. DRAMs are typically used when memory requirements are 16k words or larger. DRAM is addressed via row and column addressing. For example, one megabit DRAM requiring 20 address bits is addressed using 10 address lines and two control lines, $\overline{\text{RAS}}$ (Row Address Strobe) and $\overline{\text{CAS}}$ (Column Address Strobe). To provide a 20-bit address into the DRAM, a LOW is applied to $\overline{\text{RAS}}$ and 10 bits of the address are latched. The other 10 bits of the address are applied next and $\overline{\text{CAS}}$ is then held LOW.

The addressing capability of the DRAM can be increased by a factor of 4 by adding one more bit to the address line. This is because one additional address bit results into one additional row bit and one additional column bit. This is why DRAMs can be expanded to larger memory very rapidly with inclusion of additional address bits. External logic is required to generate the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals, and to output the current address bits to the DRAM.

DRAM controller chips take care of refreshing and timing requirements needed by the DRAMs. DRAMs typically require 4 millisecond refresh time. The DRAM controller performs its task independent of the microprocessor. The DRAM controller sends a wait signal to the microprocessor if the microprocessor tries to access memory during a refresh cycle.

Because of large memory, the address lines should be buffered using 74LS244 or 74HC244 (Unidirectional buffer), and data lines should be buffered using 74LS245 or 74HC245 (Bidirectional buffer) to increase the drive capability. Also, typical multiplexers such as 74LS157 or 74HC157 can be used to multiplex the microprocessors address lines into separate row and column addresses.

6.5 Input/Output

Input/Output (I/O) operation is typically defined as the transfer of information between the microcomputer system and an external device. There are typically three main ways of transferring data between the microcomputer system and the external devices. These are programmed I/O, interrupt I/O, and direct memory access. We now define them.

- **Programmed I/O.** Using this technique, the microprocessor executes a program to perform all data transfers between the microcomputer system and the external devices. The main characteristic of this type of I/O technique is that the external device carries out the functions as dictated by the program inside the microcomputer memory. In other words, the microprocessor completely controls all the transfers.
- **Interrupt I/O.** In this technique, an external device or an exceptional condition such as overflow can force the microcomputer system to stop executing the current program temporarily so that it can execute another program, known as the "interrupt service routine." This routine satisfies the needs of the external device or the exceptional condition. After having completed this program, the microprocessor returns to the program that it was executing before the interrupt.
- **Direct Memory Access (DMA).** This is a type of I/O technique in which data can be transferred between the microcomputer memory and external devices without any microprocessor (CPU) involvement. Direct memory access is typically used to transfer blocks of data between the microcomputer's main memory and an external device such as hard disk. An interface chip called the DMA controller chip is used with the microprocessor for transferring data via direct memory access.

6.6 Microcomputer Programming Concepts

This section includes the fundamental concepts of microcomputer programming. Typical programming characteristics such as programming languages, microprocessor instruction sets, addressing modes, and instruction formats are discussed.

6.6.1 Microcomputer Programming Languages

Microcomputers are typically programmed using semi-English-language statements (assembly language). In addition to assembly languages, microcomputers use a more understandable human-oriented language called the "high-level language." No matter what type of language is used to write the programs, the microcomputers only understand binary numbers. Therefore, the programs must eventually be translated into their appropriate binary forms. The main ways of accomplishing this are discussed later.

Microcomputer programming languages can typically be divided into three main types:

1. Machine language
2. Assembly language
3. High-level language

A machine language program consists of either binary or hexadecimal op-codes. Programming a microcomputer with either one is relatively difficult, because one must deal only with numbers. The architecture and microprograms of a microprocessor determine

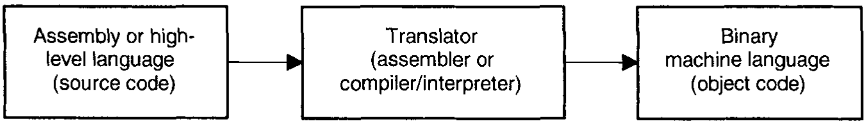

Assembly or high-level language (source code)
Translator (assembler or compiler/interpreter)
Binary machine language (object code)

FIGURE 6.30 Translating assembly or a high-level language into binary machine language

all its instructions. These instructions are called the microprocessor's "instruction set." Programs in assembly and high-level languages are represented by instructions that use English- language-type statements. The programmer finds it relatively more convenient to write the programs in assembly or a high-level language than in machine language. However, a translator must be used to convert the assembly or high-level programs into binary machine language so that the microprocessor can execute the programs. This is shown in Figure 6.30.

An assembler translates a program written in assembly language into a machine language program. A compiler or interpreter, on the other hand, converts a high-level language program such as C or C++ into a machine language program. Assembly or high-level language programs are called "source codes." Machine language programs are known as "object codes." A translator converts source codes to object codes. Next, we discuss the three main types of programming language in more detail.

6.6.2 Machine Language

A microprocessor has a unique set of machine language instructions defined by its manufacturer. No two microprocessors by two different manufacturers have the same machine language instruction set. For example, the Intel 8086 microprocessor uses the code $01D8_{16}$ for its addition instruction whereas the Motorola 68000 uses the code $D282_{16}$. Therefore, a machine language program for one microcomputer will not usually run on another microcomputer of a different manufacturer.

At the most elementary level, a microprocessor program can be written using its instruction set in binary machine language. As an example, a program written for adding two numbers using the Intel 8086 machine language is

```
1011 1000 0000 0001 0000 0000
1011 1011 0000 0010 0000 0000
0000 0001 1101 1000
1111 0100
```

Obviously, the program is very difficult to understand, unless the programmer remembers all the 8086 codes, which is impractical. Because one finds it very inconvenient to work with 1's and 0's, it is almost impossible to write an error-free program at the first try. Also, it is very tiring for the programmer to enter a machine language program written in binary into the microcomputer's RAM. For example, the programmer needs a number of binary switches to enter the binary program. This is definitely subject to errors.

To increase the programmer's efficiency in writing a machine language program, hexadecimal numbers rather than binary numbers are used. The following is the same addition program in hexadecimal, using the Intel 8086 instruction set:

```
B80100
BB0200
01D8
F4
```

It is easier to detect an error in a hexadecimal program, because each byte contains only two hexadecimal digits. One would enter a hexadecimal program using a hexadecimal keyboard. A keyboard monitor program in ROM, usually offered by the manufacturer, provides interfacing of the hexadecimal keyboard to the microcomputer. This program converts each key actuation into binary machine language in order for the microprocessor to understand the program. However, programming in hexadecimal is not normally used.

6.6.3 Assembly Language

The next programming level is to use the assembly language. Each line in an assembly language program includes four fields:

1. Label field
2. Instruction, mnemonic, or op-code field
3. Operand field
4. Comment field

As an example, a typical program for adding two 16-bit numbers written in 8086 assembly language is

Label	Mnemonic	Operand	Comment
START	MOV	AX,1	move 1 into AX
	MOV	BX,2	move 2 into BX
	ADD	AX,BX	add the contents of AX with BX
	JMP	START	jump to the beginning of the program

Obviously, programming in assembly language is more convenient than programming in machine language, because each mnemonic gives an idea of the type of operation it is supposed to perform. Therefore, with assembly language, the programmer does not have to find the numerical op-codes from a table of the instruction set, and programming efficiency is significantly improved.

The assembly language program is translated into binary via a program called an "assembler." The assembler program reads each assembly instruction of a program as ASCII characters and translates them into the respective binary op-codes. As an example, consider the HLT instruction for the 8086. Its binary op-code is 1111 0100. An assembler would convert HLT into 111 0100 as shown in Figure 6.31.

An advantage of the assembler is address computation. Most programs use addresses within the program as data storage or as targets for jumps or calls. When programming in machine language, these addresses must be calculated by hand. The assembler solves this problem by allowing the programmer to assign a symbol to an address. The programmer may then reference that address elsewhere by using the symbol. The assembler computes the actual address for the programmer and fills it in automatically. One can obtain hands-

Assembly Code	Binary form of ASCII Codes as Seen by Assembler	Binary OP Code Created by Assembler
H	0100 1000	1111 0100
L	0100 1100	
T	0101 0100	

FIGURE 6.31 Conversion of HLT into its binary op-code

on experience with a typical assembler for a microprocessor by downloading it from the Internet.

Most assemblers use two passes to assemble a program. This means that they read the input program text twice. The first pass is used to compute the addresses of all labels in the program. In order to find the address of a label, it is necessary to know the total length of all the binary code preceding that label. Unfortunately, however, that address may be needed in that preceding code. Therefore, the first pass computes the addresses of all labels and stores them for the next pass, which generates the actual binary code. Various types of assemblers are available today. We define some of them in the following paragraphs.

- **One-Pass Assembler.** This assembler goes through the assembly language program once and translates it into a machine language program. This assembler has the problem of defining forward references. This means that a `JUMP` instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.
- **Two-Pass Assembler.** This assembler scans the assembly language program twice. In the first pass, this assembler creates a symbol table. A symbol table consists of labels with addresses assigned to them. This way labels can be used for `JUMP` statements and no address calculation has to be done by the user. On the second pass, the assembler translates the assembly language program into the machine code. The two-pass assembler is more desirable and much easier to use.
- **Macroassembler.** This type of assembler translates a program written in macrolanguage into the machine language. This assembler lets the programmer define all instruction sequences using macros. Note that, by using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program. The programmer can thus avoid writing an instruction sequence that is required many times in a program by using macros. The macroassembler replaces a macroname with the appropriate instruction sequence each time it encounters a macroname.

  It is interesting to see the difference between a subroutine and a macroprogram. A specific subroutine occurs once in a program. A subroutine is executed by CALLing it from a main program. The program execution jumps out of the main program and then executes the subroutine. At the end of the subroutine, a RET instruction is used to resume program execution following the CALL SUBROUTINE instruction in the main program. A macro, on the other hand, does not cause the program execution to branch out of the main program. Each time a macro occurs, it is replaced with the appropriate instruction sequence in the main program. Typical advantages of using macros are shorter source programs and better program documentation. A disadvantage is that effects on registers and flags may not be obvious.

  Conditional macroassembly is very useful in determining whether or not an instruction sequence is to be included in the assembly depending on a condition that is true or false. If two different programs are to be executed repeatedly based on a condition that can be either true or false, it is convenient to use conditional macros. Based on each condition, a particular program is assembled. Each condition and the appropriate program are typically included within IF and ENDIF pseudo-instructions.
- **Cross Assembler.** This type of assembler is typically resident in a processor and assembles programs for another for which it is written. The cross assembler program is written in a high-level language so that it can run on different types of processors that understand the same high-level language.
- **Resident Assembler.** This type of assembler assembles programs for a processor

in which it is resident. The resident assembler may slow down the operation of the processor on which it runs.

- **Meta-assembler.** This type of assembler can assemble programs for many different types of processors. The programmer usually defines the particular processor being used.

As mentioned before, each line of an assembly language program consists of four fields: label, mnemonic or op-code, operand, and comment. The assembler ignores the comment field but translates the other fields. The label field must start with an uppercase alphabetic character. The assembler must know where one field starts and another ends. Most assemblers allow the programmer to use a special symbol or delimiter to indicate the beginning or end of each field. Typical delimiters used are spaces, commas, semicolons, and colons:

- Spaces are used between fields.
- Commas (,) are used between addresses in an operand field.
- A semicolon (;) is used before a comment.
- A colon (:) or no delimiter is used after a label.

To handle numbers, most assemblers consider all numbers as decimal numbers unless specified. Most assemblers will also allow binary, octal, or hexadecimal numbers. The user must define the type of number system used in some way. This is usually done by using a letter following the number. Typical letters used are

- `B` for binary
- `Q` for octal
- `H` for hexadecimal

Assemblers generally require hexadecimal numbers to start with a digit. A 0 is typically used if the first digit of the hexadecimal number is a letter. This is done to distinguish between numbers and labels. For example, most assemblers will require the number A5H to be represented as 0A5H.

Assemblers use pseudo-instructions or directives to make the formatting of the edited text easier. These pseudo-instructions are not directly translated into machine language instructions. They equate labels to addresses, assign the program to certain areas of memory, or insert titles, page numbers, and so on. To use the assembler directives or pseudo-instructions, the programmer puts them in the op-code field, and, if the pseudo-instructions require an address or data, the programmer places them in the label or data field. Typical pseudo-instructions are ORIGIN (`ORG`), EQUATE (`EQU`), DEFINE BYTE (`DB`), and DEFINE WORD (`DW`).

ORIGIN (ORG)

The pseudo-instruction **ORG** lets the programmer place the programs anywhere in memory. Internally, the assembler maintains a program-counter-type register called the "address counter." This counter maintains the address of the next instruction or data to be processed.

An `ORG` pseudo-instruction is similar in concept to the `JUMP` instruction. Recall that the `JUMP` instruction causes the processor to place a new address in the program counter. Similarly, the `ORG` pseudo-instruction causes the assembler to place a new value in the address counter.

Typical `ORG` statements are

```
ORG 7000H
CLC
```

The 8086 assembler will generate the following code for these statements:

```
7000 F8
```

Most assemblers assign a value of zero to the starting address of a program if the programmer does not define this by means of an `ORG`.

Equate (EQU)

The pseudo-instruction **EQU** assigns a value in its operand field to an address in its label field. This allows the user to assign a numeric value to a symbolic name. The user can then use the symbolic name in the program instead of its numeric value. This reduces errors.

A typical example of `EQU` is `START EQU 0200H`, which assigns the value 0200 in hexadecimal to the label START. Another example is

```
PORTA         EQU    40H
              MOV    AL, 0FFH
              OUT    PORTA, AL
```

In this example, the `EQU` gives PORTA the value 40 hex, and FF hex is the data to be written into register AL by `MOV AL,0FFH`. `OUT PORTA,AL` then outputs this data FF hex to port 40, which has already been equated to PORTA before.

Note that, if a label in the operand field is equated to another label in the label field, then the label in the operand field must be previously defined. For example, the `EQU` statement

```
BEGIN   EQU     START
```

will generate an error unless START is defined previously with a numeric value.

Define Byte (DB)

The pseudo-instruction **DB** is usually used to set a memory location to certain byte value. For example,

```
START   DB      45H
```

will store the data value 45 hex to the address START.

With some assemblers, the DB pseudo-instruction can be used to generate a table of data as follows:

```
                ORG     7000H
TABLE   DB      20H,30H,40H,50H
```

In this case, 20 hex is the first data of the memory location 7000; 30 hex, 40 hex, and 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

```
7000    20
7001    30
7002    40
7003    50
```

Note that some assemblers use DC.B instead of DB. DC stands for Define Constant.

Define Word (DW)

The pseudo-instruction DW is typically used to assign a 16-bit value to two memory locations. For example,

```
                ORG     7000H
START   DW      4AC2H
```

will assign C2 to location 7000 and 4A to location 7001. It is assumed that the assembler will assign the low byte first (C2) and then the high byte (4A).

With some assemblers, the DW pseudo-instruction can be used to generate a table of 16-bit data as follows:

```
                    ORG     8000H
POINTER             DW      5000H,6000H,7000H
```

In this case, the three 16-bit values 5000H, 6000H, and 7000H are assigned to memory locations starting at the address 8000H. That is, the array would look like this:

```
8000        00
8001        50
8002        00
8003        60
8004        00
8005        70
```

Note that some assemblers use DC.W instead of DW.

Assemblers also use a number of housekeeping pseudo-instructions. Typical housekeeping pseudo-instructions are `TITLE`, `PAGE`, `END`, and `LIST`. The following are the housekeeping pseudo-instructions that control the assembler operation and its program listing.

TITLE prints the specified heading at the top of each page of the program listing. For example,

```
TITLE "Square Root Algorithm"
```

will print the name "Square Root Algorithm" on top of each page.

PAGE skips to the next line.

END indicates the end of the assembly language source program.

LIST directs the assembler to print the assembler source program.

In the following, assembly language instruction formats, instruction sets, and addressing modes available with typical microprocessors will be discussed.

Assembly Language Instruction Formats

Depending on the number of addresses specified, we have the following instruction formats:

- Three address
- Two address
- One address
- Zero address

Because all instructions are stored in the main memory, instruction formats are designed in such a way that instructions take less space and have more processing capabilities. It should be emphasized that the microprocessor architecture has considerable influence on a specific instruction format. The following are some important technical points that have to be considered while designing an instruction format:

- The size of an instruction word is chosen in such a way that it facilitates the specification of more operations by a designer. For example, with 4- and 8-bit op-code fields, we can specify 16 and 256 distinct operations respectively.
- Instructions are used to manipulate various data elements such as integers, floating-point numbers, and character strings. In particular, all programs written in a symbolic language such as C are internally stored as characters. Therefore, memory space will not be wasted if the word length of the machine is some integral multiple of the number

of bits needed to represent a character. Because all characters are represented using typical 8-bit character codes such as ASCII or EBCDIC, it is desirable to have 8-, 16-, 32-, or 64-bit words for the word length.

- The size of the address field is chosen in such a way that a high resolution is guaranteed. Note that in any microprocessor, the ultimate resolution is a bit. Memory resolution is function of the instruction length, and in particular, short instructions provide less resolution. For example, in a microcomputer with 32K 16-bit memory words, at least 19 bits are required to access each bit of the word. (This is because 2^{15} = 32K and 2^4 = 16)

The general form of a *three address instruction* is shown below:

<op-code> Addr1, Addr2, Addr3

Some typical three-address instructions are

```
MUL   A, B, C          ;    C  <- A * B
ADD   A, B, C          ;    C  <- A + B
SUB   R1, R2, R3       ;    R3 <- R1 - R2
```

In this specification, all alphabetic characters are assumed to represent memory addresses, and the string that begins with the letter R indicates a register. The third address of this type of instruction is usually referred to as the "destination address." The result of an operation is always assumed to be saved in the destination address.

Typical programs can be written using these three address instructions. For example, consider the following sequence of three address instructions

```
MUL   A, B, R1         ;    R1 <- A * B
MUL   C, D, R2         ;    R2 <- C * D
MUL   E, F, R3         ;    R3 <- E * F
ADD   R1, R2, R1       ;    R1 <- R1 + R2
SUB   R1, R3, Z        ;    Z  <- R1 - R3
```

This sequence implements the statement `Z = A * B + C * D - E * F`. The three-address format is normally used by 32-bit microprocessors in addition to the other formats.

If we drop the third address from the three-address format, we obtain the two-address format. Its general form is

<op-code> Addr1, Addr2

Some typical *two-address* instructions are

```
MOV   A, R1        ;    R1 <- A
ADD   C, R2        ;    R2 <- R2 + C
SUB   R1, R2       ;    R2 <- R2 - R1
```

In this format, the addresses Addr1 and Addr2 respectively represent source and destination addresses. The following sequence of two-address instructions is equivalent to the program using three-address format presented earlier:

```
MOV   A, R1        ;    R1 <- A
MUL   B, R1        ;    R1 <- R1 * B
MOV   C, R2        ;    R2 <- C
MUL   D, R2        ;    R2 <- R2 * D
MOV   E, R3        ;    R3 <- E
MUL   F, R3        ;    R3 <- R3 * F
ADD   R2, R1       ;    R1 <- R1 + R2
SUB   R3, R1       ;    R1 <- R1 - R3
MOV   R1, Z        ;    Z  <- R1
```

This format is predominant in typical general-purpose microprocessors such as the Intel 8086 and the Motorola 68000. Typical 8-bit microprocessors such as the Intel 8085 and the Motorola 6809 are accumulator based. In these microprocessors, the accumulator register is assumed to be the destination for all arithmetic and logic operations. Also, this register always holds one of the source operands. Thus, we only need to specify one address in the instruction, and therefore, this idea reduces the instruction length. The one-address format is predominant in 8-bit microprocessors. Some typical one-address instructions are

```
LDA   B    ;    Acc <- B
ADD   C    ;    Acc <- Acc + C
MUL   D    ;    Acc <- Acc * D
STA   E    ;    E <- Acc
```

The following program illustrates how one can translate the statement `Z = A * B + C * D - E * F` into a sequence of one-address instructions:

```
LDA   E    ;    Acc <- E
MUL   F    ;    Acc <- Acc * F
STA   T1   ;    T1  <- Acc
LDA   C    ;    Acc <- C
MUL   D    ;    Acc <- Acc * D
STA   T2   ;    T2 <- Acc
LDA   A    ;    Acc <- A
MUL   B    ;    Acc <- Acc * B
ADD   T2   ;    Acc <- Acc + T2
SUB   T1   ;    Acc <- Acc - T1
STA   Z    ;    Z <- Acc
```

In this program, T1 and T2 represent the addresses of memory locations used to store temporary results. Instructions that do not require any addresses are called "zero-address instructions." All microprocessors include some zero-address instructions in the instruction set. Typical examples of zero-address instructions are `CLC` (clear carry) and `NOP`.

Typical Assembly Language Instruction Sets

An instruction set of a specific microprocessor consists of all the instructions that it can execute. The capabilities of a microprocessor are determined, to some extent, by the types of instructions it is able to perform. Each microprocessor has a unique instruction set designed by its manufacturer to do a specific task. We discuss some of the instructions that are common to all microprocessors. We will group chunks of these instructions together which have similar functions. These instructions typically include

- **Data Processing Instructions.** These operations perform actual data manipulations. The instructions typically include arithmetic/logic operations and increment/decrement and rotate/shift operations. Typical arithmetic instructions include ADD, SUBTRACT, COMPARE, MULTIPLY, AND DIVIDE. Note that the SUBTRACT instruction provides the result and also affects the status flags while the COMPARE instruction performs subtraction without any result and affects the flags based on the result. Typical logic instructions perform traditional Boolean operations such as AND, OR, and EXCLUSIVE-OR. The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where unknown bit value of Y is to be determined) can be obtained as follows:

```
          0 1 0 0  1 Y 1 0  -- 8-bit number
AND       0 0 0  0  0 1 0  0 -- Masking data
          --------------------
          0 0 0  0  0 Y 0 0 -- Result
```

If the bit value Y at bit 2 is 1, then the result is nonzero (Flag Z=0); otherwise, the result is zero (Flag Z=1). The Z flag can be tested using typical conditional JUMP instructions such as JZ (Jump if Z=1) or JNZ(Jump if Z=0) to determine whether Y is 0 or 1. This is called masking operation. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the Least Significant bit (LSB) of the number (LSB=0 for even and LSB=1 for odd). The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
      0 1 1 1 0 0 1 1 -- 8-bit  number
OR    0 0 0 0 1 0 0 0 -- data for inserting a 1 at  bit number 3
       ------------------
      0 1 1 1 1 0 1 1 -- Result
```

The Exclusive-OR instruction can be used to find the ones complement of a binary number by XORing the number with all 1's as follows:

```
          0 1 0 1 1 1 0 0 - -  8-bit number
XOR       1 1 1 1 1 1 1 1 - -  data
   --------------------------
          1 0 10 0 0 1 1 -- Result (Ones Complement  of the 8-bit number
                                    0 1 0 1 1 1 0 0 )
```

- **Instructions for Controlling Microprocessor Operations.** These instructions typically include those that set the reset specific flags and halt or stop the microprocessor.
- **Data Movement Instructions.** These instructions move data from a register to memory and vice versa, between registers, and between a register and an I/O device.
- **Instructions Using Memory Addresses.** An instruction in this category typically contains a memory address, which is used to read a data word from memory into a microprocessor register or for writing data from a register into a memory location. Many instructions under data processing and movement fall in this category.
- **Conditional and Unconditional JUMPS.** These instructions typically include one of the following:
  1. Unconditional JUMP, which always transfers the memory address specified in the instruction into the program counter.
  2. Conditional JUMP, which transfers the address portion of the instruction into the program counter based on the conditions set by one of the status flags in the flag register.

Typical Assembly Language Addressing Modes

One of the tasks performed by a microprocessor during execution of an instruction is the determination of the operand and destination addresses. The manner in which a microprocessor accomplishes this task is called the "addressing mode." Now, let us present the typical microprocessor addressing modes, relating them to the instruction sets of Motorola 68000.

An instruction is said to have "implied or inherent addressing mode" if it does not have any operand. For example, consider the following instruction: RTS, which means "return from a subroutine to the main program." The RTS instruction is a no-operand instruction. The program counter is implied in the instruction because although the program counter is not included in the RTS instruction, the return address is loaded in the program counter after its execution.

Whenever an instruction/operand contains data, it is called an "immediate mode" instruction. For example, consider the following 68000 instruction:

```
ADD    #15, D0      ;       D0 <- D0 + 15
```

In this instruction, the symbol # indicates to the assembler that it is an immediate mode instruction. This instruction adds 15 to the contents of register D0 and then stores the result in D0. An instruction is said to have a register mode if it contains a register as opposed to a memory address. This means that the operand values are held in the microprocessor registers. For example, consider the following 68000 instruction:

```
ADD    D1, D0       ; D0 <- D1 + D0
```

This ADD instruction is a two-operand instruction. Both operands (source and destination) have register mode. The instruction adds the 16-bit contents of D0 to the 16-bit contents of D1 and stores the 16-bit result in D0.

An instruction is said to have an absolute or direct addressing mode if it contains a memory address in the operand field. For example, consider the 68000 instruction

```
ADD    3000, D2
```

This instruction adds the 16-bit contents of memory address 3000 to the 16-bit contents of D2 and stores the 16-bit result in D2. The source operand to this ADD instruction contains 3000 and is in absolute or direct addressing mode. When an instruction specifies a microprocessor register to hold the address, the resulting addressing mode is known as the "register indirect mode." For example, consider the 68000 instruction:

```
CLR (A0)
```

This instruction clears the 16-bit contents of a memory location whose address is in register A0 to zero. The instruction is in register indirect mode.

The conditional branch instructions are used to change the order of execution of a program based on the conditions set by the status flags. Some microprocessors use conditional branching using the absolute mode. The op-code verifies a condition set by a particular status flag. If the condition is satisfied, the program counter is changed to the value of the operand address (defined in the instruction). If the condition is not satisfied, the program counter is incremented, and the program is executed in its normal order.

Typical 16-bit microprocessors use conditional branch instructions. Some conditional branch instructions are 16 bits wide. The first byte is the op-code for checking a particular flag. The second byte is an 8-bit offset, which is added to the contents of the program counter if the condition is satisfied to determine the effective address. This offset is considered as a signed binary number with the most significant bit as the sign bit. It means that the offset can vary from -128_{10} to $+127_{10}$ (0 being positive). This is called relative mode.

Consider the following 68000 example, which uses the branch not equal (`BNE`) instruction:

`BNE 8`

Suppose that the program counter contains 2000 (address of the next instruction to be executed) while executing this `BNE` instruction. Now, if $Z = 0$, the microprocessor will load 2000 + 8 = 2008 into the program counter and program execution resumes at address 2008. On the other hand, if $Z = 1$, the microprocessor continues with the next instruction.

In the last example the program jumped forward, requiring positive offset. An example for branching with negative offset is

`BNE -14`

Suppose that the current program counter value = 2004_{16}

$$= 0010\ 0000\quad 0000\ 0100$$

offset = 2's complement of 14_{10} = $F2_{16}$

$$= \boxed{1111\ 1111}\ \ 1111\ 0010$$

$$\underset{\text{ignore}}{\nearrow} 1\ \underbrace{0001}_{1}\ \underbrace{1111}_{F}\ /\ \underbrace{1111}_{F}\ \underbrace{0110}_{6_{16}}$$

reflect this 1 to the high byte
(sign extension)

Therefore, to branch backward to $1FF6_{16}$, the assembler uses an offset of F2 following the op-code for BNE.

An advantage of relative mode is that the destination address is specified relaive to the address of the instruction after the instruction. Since these conditional Jump instructions do not contain an absolute address, the program can be placed anywhere in memory which can still be excuted properly by the microprocessor. A program which can be placed anywhere in memory, and can still run correctly is called a "relocatable" program. It is a good practice to write relocatable programs.

Subroutine Calls in Assembly Language

It is sometimes desirable to execute a common task many times in a program. Consider the case when the sum of squares of numbers is required several times in a program. One could write a sequence of instructions in the main program for carrying out the sum of squares every time it is required. This is all right for short programs. For long programs, however, it is convenient for the programmer to write a small program known as a "subroutine" for performing the sum of squares, and then call this program each time it is needed in the main program.

Therefore, a subroutine can be defined as a program carrying out a particular function that can be called by another program known as the "main program." The subroutine only needs to be placed once in memory starting at a particular memory location. Each time the main program requires this subroutine, it can branch to it, typically by using a jump to subroutine (`JSR`) instruction along with its starting address. The subroutine is then executed. At the end of the subroutine, a `RETURN` instruction takes control back to the main program.

The 68000 includes two subroutine call instructions. Typical examples include `JSR 4000` and `BSR 24`. `JSR 4000` is an instruction using absolute mode. In response to the execution of `JSR`, the 68000 saves (pushes) the current program counter contents (address of the next instruction to be executed) onto the stack. The program counter is then loaded, with 4000 included in the `JSR` instruction. The starting address of the subroutine is 4000. The `RTS` (return from subroutine) at the end of the subroutine reads (pops) the return address saved into the stack before jumping to the subroutine into the program counter. The program execution thus resumes in the main program. `BSR 24` is an instruction using relative mode. This instruction works in the same way as the `JSR 4000` except that displacement `24` is added to the current program counter contents to jump to the subroutine.

The stack must always be balanced. This means that a PUSH instruction in a subroutine must be followed by a POP instruction before the RETURN from subroutine instruction so that the stack pointer points to the right return address saved onto the stack. This will ensure returning to the desired location in the main program after execution of the subroutine. If multiple registers are PUSHED in a subroutine, one must POP them in the reverse order before the subroutine RETURN instruction.

6.6.4 High-Level Languages

As mentioned before, the programmer's efficiency with assembly language increases significantly compared to machine language. However, the programmer needs to be well acquainted with the microprocessor's architecture and its instruction set. Further, the programmer has to provide an op-code for each operation that the microprocessor has to carry out in order to execute a program. As an example, for adding two numbers, the programmer would instruct the microprocessor to load the first number into a register, add the second number to the register, and then store the result in memory. However, the programmer might find it tedious to write all the steps required for a large program. Also, to become a reasonably good assembly language programmer, one needs to have a lot of experience.

High-level language programs composed of English-language-type statements rectify all these deficiencies of machine and assembly language programming. The programmer does not need to be familiar with the internal microprocessor structure or its instruction set. Also, each statement in a high-level language corresponds to a number of assembly or machine language instructions. For example, consider the statement `F = A + B` written in a high-level language called FORTRAN. This single statement adds the contents of A with B and stores the result in F. This is equivalent to a number of steps in machine or assembly language, as mentioned before. It should be pointed out that the letters A, B, and F do not refer to particular registers within the microprocessor. Rather, they are memory locations.

A number of high-level languages such as C and C++ are widely used these days. Typical microprocessors, namely, the Intel 8086, the Motorola 68000, and others, can be programmed using these high-level languages. A high-level language is a problem-oriented language. The programmer does not have to know the details of the architecture of the microprocessor and its instruction set. Basically, the programmer follows the rules of the particular language being used to solve the problem at hand. A second advantage is that a program written in a particular high-level language can be executed by two different microcomputers, provided they both understand that language. For example, a program written in C for an Intel 8086–based microcomputer will run on a Motorola 68000-based microcomputer because both microprocessors have a compiler to translate the C language into their particular machine language; minor modifications are required for input/output programs.

As mentioned before, like the assembly language program, a high-level language program requires a special program for converting the high-level statements into object codes. This program can be either an interpreter or a compiler. They are usually very large programs compared to assemblers.

An interpreter reads each high-level statement such as `F = A + B` and directs the microprocessor to perform the operations required to execute the statement. The interpreter converts each statement into machine language codes but does not convert the entire program into machine language codes prior to execution. Hence, it does not generate an object program. Therefore, an interpreter is a program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function. A compiler, however, converts each statement into a set of machine language instructions and also produces an object program that is stored in memory. This program must then be executed by the microprocessor to perform the required task in the high-level program. In summary, an interpreter executes each statement as it proceeds, without generating an object code, whereas a compiler converts a high-level program into an object program that is stored in memory. This program is then executed. Compilers normally provide inefficient machine codes because of the general guidelines that must be followed for designing them. C, C++, and Java are the only high-level languages that include Input/ Output instructions. However, the compiled codes generate many more lines of machine code than an equivalent assembly language program. Therefore, the assembled program will take up less memory space and will execute much faster compared to the compiled C, C++, or Java codes. I/O programs written in C are compared with assembly language programs written in 8086 and 68000 in Chapters 9 and 10. C language is a popular high-level language, the C++ language, based on C, is also very popular, and Java, developed by Sun Microsystems, is gaining wide acceptance.

Therefore, one of the main uses of assembly language is in writing programs for real-time applications. "Real-time" means that the task required by the application must be completed before any other input to the program can occur which will change its operation. Typical programs involving non-real-time applications and extensive mathematical computations may be written in C, C++, or Java. A brief description of these languages is given in the following.

C Language

The C Programming language was developed by Dennis Ritchie of Bell Labs in 1972. C has become a very popular language for many engineers and scientists, primarily because it is portable except for I/O and however, can be used to write programs requiring I/O operations with minor modifications. This means that a program written in C for the 8086 will run on the 68000 with some modifications related to I/O as long as C compilers for both microprocessors are available.

C is case sensitive. This means that uppercase letters are different from lowercase letters. Hence Start and start are two different variables. C is a general-purpose programming language and is found in numerous applications as follows:

- **Systems Programming.** Many operating systems, compilers, and assemblers are written in C. Note that an operating system typically is included with the personal computer when it is purchased. The operating system provides an interface between the user and the hardware by including a set of commands to select and execute the software on the system
- **Computer-Aided Design (CAD) Applications.** CAD programs are written in C. Typical tasks to be accomplished by a CAD program are logic synthesis and

simulation.

- **Numerical Computation.** To solve mathematical problems such as integration and differentiation
- **Other Applications.** These include programs for printers and floppy disk controllers, and digital control algorithms using single-chip microcomputers.

A C program may be viewed as a collection of functions. Execution of a C program will always begin by a call to the function called "main." This means that all C programs should have its main program named as **`main`**. However, one can give any name to other functions.

A simple C program that prints "I wrote a C-program" is

```
/* First C-program */
#include <stdio.h>
main ( )
{
   printf("I wrote a C-program");
}
```

Here, `main` is a function of no arguments, indicated by ( ). The parenthesis must be present even if there are no arguments. The braces { } enclose the statements that make up the function.

The line `printf("I wrote a C-program");` is a function call that calls a function named `printf`, with the argument "`I wrote a C-program`." `printf` is a library function that prints output on the terminal. Note that `/* */` is used to enclose comments. These are not translated by the compiler.

A variation of the C program just described is

```
/* Another C program */
#include <stdio.h>
main ( )
{
   printf("I wrote");
   printf(" a C-");
   printf("program");
   printf("\n");
}
```

Here, `#include` is a preprocessor directive for the C language compiler. These directives give instructions to the compiler that are performed before the program is compiled. The directive `#include <stdio.h>` inserts additional statements in the program. These statements are contained in the file stdio.h. The file `stdio.h` is included with the standard C library. The `stdio.h` file contains information related to the input/output statement.

The `\n` in the last line of the program is C notation for the newline character. Upon printing, the cursor moves forward to the left margin on the next line. `printf` never supplies a newline automatically. Therefore, multiple `printf`'s may be used to output "I wrote a C-program" on a single line in a few steps. The escape sequence `\n` can be used to print three statements on three different lines. An illustration is given in the following:

```
#include <stdio.h>
main ( )
{
   printf("I wrote a C-Program \n");
   printf("This will be printed on a new line \n");
   printf("So also is this line \n");
}
```

All variables in C must be declared before use, normally at the start of the function before any executable statements. The compiler provides an error message if one forgets a declaration. A declaration includes a type and a list of variables that have that type. For example, the declaration `int a, b` implies that the variables a and b are integers. Next, write a program to add and subtract two integers a and b where a = 100 and b = 200. The C program is

```
#include <stdio.h>
main ( )
{
   int a = 100, b = 200;          /*a and b are integers
*/
   printf("The sum is: %d \n", a + b);
   printf("The difference is: %d \n", a - b);
}
```

The `%d` in the `printf` statement represents "decimal integer." Note that `printf` is not part of the C language; there is no input or output defined in C itself. `printf` is a function that is contained in the standard library of routines that can be accessed by C programs. The values of a and b can be entered via the keyboard by using the scanf function. The scanf allows the programmer to enter data from the keyboard. A typical expression for `scanf` is

```
scanf("%d%d", &a, &b);
```

This expression indicates that the two values to be entered via the keyboard are in decimal. These two decimal numbers are to be stored in addresses a and b. Note that the symbol & is an address operator.

The C program for adding and subtracting two integers a and b using `scanf` is

```
/* C Program that performs basic I/O */
#include <stdio.h>
main ( )
{
   int a,b;
   printf("Input two integers: ");
   scanf("%d%d", &a, &b);
   printf("Their sum is: %d\n", a + b);
   printf("Their difference is: %d\n", a - b);
}
```

In summary, writing a working C program involves four steps as follows:

Step 1: Using a text editor, prepare a file containing the C code. This file is called the "source file."

Step 2 Preprocess the code. The preprocessor makes the code ready for compiling. The preprocessor looks through the source file for lines that start with a #. In the previous programming examples, `#include <stdio.h>` is a preprocessor. This preprocessor instruction copies the contents of the standard header file `stdio.h` into the source code. This header file `stdio.h` describes typical input/output functions such as `scanf( )` and `printf( )` functions.

Step 3: The compiler translates the preprocessed code into machine code. The output from the compiler is called object code.

Step 4: The linker combines the object file with code from the C libraries. For instance, in the examples shown here, the actual code for the library function `printf ( )` is inserted from the standard library to the object code by the linker. The linker generates an executable file. Thus, the linker makes a complete program.

Before writing C programs, the programmer must make sure that the computer runs either the UNIX or MS-DOS operating system. Two essential programming tools are required. These are a text editor and a C compiler. The text editor is a program provided with a computer system to create and modify compiler files. The C compiler is also a program that translates C code into machine code.

C++

C++ is a modified version of C language. C++ was developed by Bjarne Stroustrup of Bell Labs in 1980. It includes all features of C and also supports object-oriented programming (OOP). A program can be divided into subprograms using OOP. Each subprogram is an independent object with its own instructions and data. Thus, complexity of programming is reduced. It is therefore easier for the programmer to manage larger programs.

All OOP languages including C++, have three characteristics: encapsulation, polymorphism, and inheritance. *Encapsulation* is a technique that keeps code and data together in such a way that they are protected form outside interference and misuse. A subprogram thus created is called an "object."

Code, data, or both may be private or public. Private code and/or data may be accessed by another part of the same object. On the other hand, public code and/or data may be accessed by a program resident outside the object containing them. One of the most important characteristic of C++ is the class. The class declaration is a technique for creating an object. Note that a class consists of data and functions.

Encapsulation is available with C to some extent. For example, when a library function such as `printf` is used, one uses a black box program. When `printf` is used, several internal variables are created and intialized that are not accessible to the programmer.

Polymorphism (from Greek word meaning "several forms") allows one to define a general class of actions. Within a general class, the specific action is determined by the type of data. For example, in C, the absolute value actions `abs ( )` and `fabs ( )` compute the absolute values of an integer and a floating point number respectively. In C++, on the other hand, one absolute value action, `abs ( )` is used for both data types. The type of data is then used to call `abs ( )` to determine which specific version of the function is actually used. Thus, one function name for two different data items is used.

Inheritance is the ability by which one class called subclass obtains the properties of another class called a superclass. Inheritance is convenient for code reusability. Inheritance supports hierarchy classes.

Following are some basic differences between C and C++:

1. In C, one must use `void` with the prototype for a function with no arguments. For example, in C, the prototype `int rand(void);` returns an integer that is a random number.
   In C++, the `void` is optional. Therefore, in C++, the prototype for `rand( )` can be written as `int rand( );`. Of course, `int rand(void);` is a

valid prototype in C++. This means that both prototypes are allowed in C++

2. C++ can use the C type of comment mechanism. That is, a comment can start with `/*` and end with `*/`. C++ can also use a simple line comment that starts with a `//` and stops at the end of the line terminated by a carriage return. Typically, C++ uses C-like comments for multiline comments and the C++ comment mechanism for short comments.
3. In C++, local variables can be declared anywhere. In contrast, in C, local variables must be declared at the start of a block before any action statements.
4. In C++, all functions need to be prototyped. In C, prototypes are optional. Note that a function prototype allows the compiler to check that the function is called with the proper number and types of arguments. It also tells the compiler the type of value that the function is supposed to return. In C, if the function prototype is omitted, the compiler will return an integer. An example of a prototype function is `int abs(int n)` , this provides an integer that is an absolute value of `n`.

Java

Introduced in 1991 by Sun MicroSystems, Java is based on C++ and is a true object oriented language. That is, everything in a Java program is an object and everything is obtained from a single object class.

A Java program must include at least one class. A class includes data type declarations and statements. Every Java standalone program requires a main method at the beginning. Java only supports class methods and not separate functions. There is no preprocessor in Java. However, there is an `import` statement, which is similar to the `#include` preprocessor statement in C. The purpose of the `import` statement in Java is to instruct the interpreter to load the class, which exists in another compilation statement. Java uses the same comment syntax, `/*  */` and `//`, as C and C++. In addition, a special comment syntax, `/**  */`, that can precede declarations is used in Java.

Java does not require pointers. In C, a pointer may be substituted for the array name to access array elements. In Java, arrays are created by using the "new" operator by including the size of the array in the new expression (rather than in the declaration) as follows:

```
int array [ ] = new int[6];
```

Also, all arrays store the specified size in a variable named `length` as follows:

```
int stringsize = array.length;
```

Therefore, in Java, arrays and strings are not subject to the errors or confusion that is common to arrays and strings in C.

6.7 Monitors

A monitor consists of a number of subroutines grouped together to provide "intelligence" to a microcomputer system. This intelligence gives the microcomputer with the capabilities for software development of user programs such as assembling and debugging. The monitor is typically offered by the microprocessor manufacturers and others in a ROM or CD memory. When a microcomputer is designed by connecting the microprocessor, memory, and I/O, a monitor program can be used for development of user programs.

An example of a monitor is the Intel SDK-86 monitor, which contains debugging routines, a display routine, and many other programs. The user can assemble, debug, execute and display results for user-written 8086 assembly language programs using the monitor provided by Intel with the SDK-86 microcomputer.

6.8 Flowcharts

Before writing an assembly language program for a specific operation, it is convenient to represent the program in a schematic form called *flowchart.* A brief listing of the basic shapes used in a flowchart and their functions is given in Figure 6.32.

6.9 Basic Features of Microcomputer Development Systems

A microcomputer development system is a tool that allows the designer to develop, debug, and integrate error-free application software in microprocessor systems.

Development systems fall into one of two categories: systems supplied by the device manufacturer (nonuniversal systems) and systems built by after-market manufacturers (universal systems). The main difference between the two categories is the range of microprocessors that a system will accommodate. Nonuniversal systems are supplied by the microprocessor manufacturer (Intel, Motorola) and are limited to use for the particular microprocessor manufactured by the supplier. In this manner, an Intel development system may not be used to develop a Motorola-based system. The universal development systems (Hewlett-Packard, Tektronix) can develop hardware and software for several microprocessors.

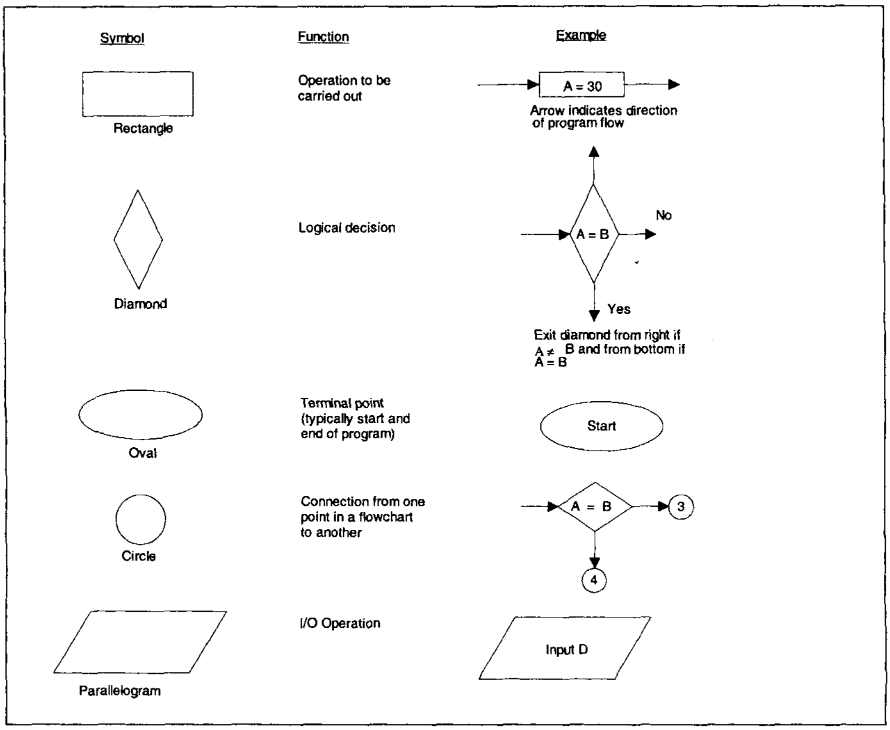

Symbol
Function
Example
Operation to be carried out
A = 30
Arrow indicates direction of program flow
Rectangle
Logical decision
A = B
No
Yes
Diamond
Exit diamond from right if A ≠ B and from bottom if A = B
Terminal point (typically start and end of program)
Start
Oval
Connection from one point in a flowchart to another
A = B
3
4
Circle
I/O Operation
Input D
Parallelogram

FIGURE 6.32 Flowchart symbols

Within both categories of development systems, there are basically three types available: single-user systems, time-shared systems, and networked systems. A single-user system consists of one development station that can be used by one user at a time. Single-user systems are low in cost and may be sufficient for small systems development. Time-shared systems usually consist of a "dumb" type of terminal connected by data lines to a centralized microcomputer-based system that controls all operations. A networked system usually consists of a number of smart cathode ray tubes (CRTs) capable of performing most of the development work and can be connected over data lines to a central microcomputer. The central microcomputer in a network system usually is in charge of allocating disk storage space and will download some programs into the user's workstation microcomputer. A microcomputer development system is a combination of the hardware necessary for microprocessor design and the software to control the hardware. The basic components of the hardware are the central processor, the CRT terminal, mass storage device (floppy or hard disk), and usually an in-circuit emulator (ICE).

In a single-user system, the central processor executes the operating system software, handles the input/output (I/O) facilities, executes the development programs (editor, assembler, linker), and allocates storage space for the programs in execution. In a large multiuser networked system the central processor may be responsible for the I/O facilities and execution of development programs. The CRT terminal provides the interface between the user and the operating system or program under execution. The user enters commands or data via the CRT keyboard, and the program under execution displays data to the user via the CRT screen. Each program (whether system software or user program) is stored in an ordered format on disk. Each separate entry on the disk is called a *file*. The operating system software contains the routines necessary to interface between the user and the mass storage unit. When the user requests a file by a specific *file name*, the operating system finds the program stored on disk by the file name and loads it into mean memory. More advanced development systems contain *memory management* software that protects a user's files from unauthorized modification by another user. This is accomplished via a unique user identification code called USER ID. A user can only access files that have the user's unique code. The equipment listed here makes up a basic development system, but most systems have other devices such as printers and EPROM and PAL programmers attached. A printer is needed to provide the user with a hard copy record of the program under development.

After the application system software has been completely developed and debugged, it needs to be permanently stored for execution in the target hardware. The EPROM (erasable/programmable read-only memory) programmer takes the machine code and programs it into an EPROM. EPROMs are more generally used in system development because they may be erased and reprogrammed if the program changes. EPROM programmers usually interface to circuits particularly designed to program a specific EPROM.

Most development systems support one or more in-circuit emulators (ICEs). The ICE is one of the most advanced tools for microprocessor hardware development. To use an ICE, the microprocessor chip is removed from the system under development (called the target processor) and the emulator is plugged into the microprocessor socket. The ICE will functionally and electrically act identically to the target processor with the exception that the ICE is under the control of development system software. In this manner the development system may exercise the hardware that is being designed and monitor all status information available about the operation of the target processor. Using an ICE, processor register contents may be displayed on the CRT and operation of the hardware observed in a single-stepping mode. In-circuit emulators can find hardware and software bugs quickly that might take many hours to locate using conventional hardware testing methods.

Architectures for development systems can be generally divided into two categories: the master/slave configuration and the single-processor configuration. In a master/slave configuration, the master (host) processor controls the mass storage device and processes all I/O (CRT, printer). The software for development systems is written for the master processor, which is usually not the same as the slave (target) processor. The slave microprocessor is typically connected to the user prototype via a connector which links the slave processor to the master processor.

Some development systems such as the HP 64000 completely separate the system bus from the emulation bus and therefore use a separate block of memory for emulation. This separation allows passive monitoring of the software executing on the target processor without stopping the emulation process. A benefit of the separate emulation facilities allows the master processor to be used for editing, assembling, and so on while the slave processor continues the emulation. A designer may therefore start an emulation running, exit the emulator program, and at some future time return to the emulation program.

Another advantage of the separate bus architecture is that an operating system needs to be written only once for the master processor and will be used no matter what type of slave processor is being emulated. When a new slave processor is to be emulated, only the emulator probe needs to be changed.

A disadvantage of the master/slave architecture is that it is expensive. In single-processor architecture, only one processor is used for system operation and target emulation. The single processor does both jobs, executing system software as well as acting as the target processor. Because there is only one processor involved, the system software must be rewritten for each type of processor that is to be emulated. Because the system software must reside in the same memory used by the emulator, not all memory will be available to the emulation process, which may be a disadvantage when large prototypes are being developed. The single-processor systems are inexpensive.

The programs provided for microprocessor development are the operating system, editor, assembler, linker, compiler, and debugger. The operating system is responsible for executing the user's commands. The operating system handles I/O functions, memory management, and loading of programs from mass storage into RAM for execution. The editor allows the user to enter the source code (either assembly language or some high-level language) into the development system.

Almost all current microprocessor development systems use the character-oriented editor, more commonly referred to as the screen editor. The editor is called a "screen editor" because the text is dynamically displayed on the screen and the display automatically updates any edits made by the user.

The screen editor uses the pointer concept to point to the character(s) that need editing. The pointer in a screen editor is called the "cursor," and special commands allow the user to position the cursor to any location displayed on the screen. When the cursor is positioned, the user may insert characters, delete characters, or simply type over the existing characters.

Complete lines may be added or deleted using special editor commands. By placing the editor in the insert mode, any text typed will be inserted at the cursor position when the cursor is positioned between two existing lines. If the cursor is positioned on a line to be deleted, a single command will remove the entire line from the file.

Screen editors implement the editor commands in different fashions. Some editors use dedicated keys to provide some cursor movements. The cursor keys are usually marked with arrows to show the direction of the cursor movement. More advanced editors (such as the HP 64000) use soft keys. A soft key is an unmarked key located on the keyboard directly below the bottom of the CRT screen. The mode of the editor decides what functions the keys are to perform. The function of each key is displayed on the screen directly above the appropriate key. The soft key approach is valuable because it allows the editor to reassign a key to a new function when necessary.

The source code generated on the editor is stored as ASCII or text characters and cannot be executed by a microprocessor. Before the code can be executed, it must be converted to a form accessible by the microprocessor. An assembler is the program used to translate the assembly language source code generated with an editor into object code (machine code), which may be executed by a microprocessor.

The output file from most development system assemblers is an object file. The object file is usually relocatable code that may be configured to execute at any address. The function of the linker is to convert the object file to an *absolute* file, which consists of the actual machine code at the correct address for execution. The absolute files thus created are used for debugging and finally for programming EPROMs.

Debugging a microprocessor-based system may be divided into two categories: software debugging and hardware debugging. Both debugging processes are usually carried out separately because software debugging can be carried out on an out-of-circuit emulator (OCE) without having the final system hardware.

The usual software development tools provided with the development system are

- Single-step facility
- Breakpoint facility

A single stepper simply allows the user to execute the program being debugged one instruction at a time. By examining the register and memory contents during each step, the debugger can detect such program faults as incorrect jumps, incorrect addressing, erroneous op-codes, and so on. A breakpoint allows the user to execute an entire section of a program being debugged.

There are two types of breakpoints: hardware and software. The hardware breakpoint uses the hardware to monitor the system address bus and detect when the program is executing the desired breakpoint location. When the breakpoint is detected, the hardware uses the processor control lines to halt the processor for inspection or cause the processor to execute an interrupt to a breakpoint routine. Hardware breakpoints can be used to debug both ROM- and RAM-based programs. Software breakpoint routines may only operate on a system with the program in RAM because the breakpoint instruction must be inserted into the program that is to be executed.

Single-stepper and breakpoint methods complement each other. The user may insert a breakpoint at the desired point and let the program execute up to that point. When the program stops at the breakpoint the user may use a single-stepper to examine the program one instruction at a time. Thus, the user can pinpoint the error in a program.

There are two main hardware-debugging tools: the logic analyzer and the in-circuit emulator. Logic analyzers are usually used to debug hardware faults in a system. The logic analyzer is the digital version of an oscilloscope because it allows the user to view logic levels in the hardware. In-circuit emulators can be used to debug and integrate software and hardware. PC-based workstations are extensively used as development systems.

6.10 System Development Flowchart

The total development of a microprocessor-based system typically involves three phases: software design, hardware design, and program diagnostic design. A systems programmer will be assigned the task of writing the application software, a logic designer will be assigned the task of designing the hardware, and typically both designers will be assigned the task of developing diagnostics to test the system. For small systems, one engineer may do all three phases, while on large systems several engineers may be assigned to each phase. Figure 6.33 shows a flowchart for the total development of a system. Notice that software and hardware development may occur in parallel to save time.

The first step in developing the software is to take the system specifications and write a flowchart to accomplish the desired tasks that will implement the specifications. The assembly language or high-level source code may now be written from the system flowchart. The complete source code is then assembled. The assembler is the object code and a program listing. The object code will be used later by the linker. The program listing may be sent to a disk file for use in debugging, or it may be directed to the printer.

The linker can now take the object code generated by the assembler and create

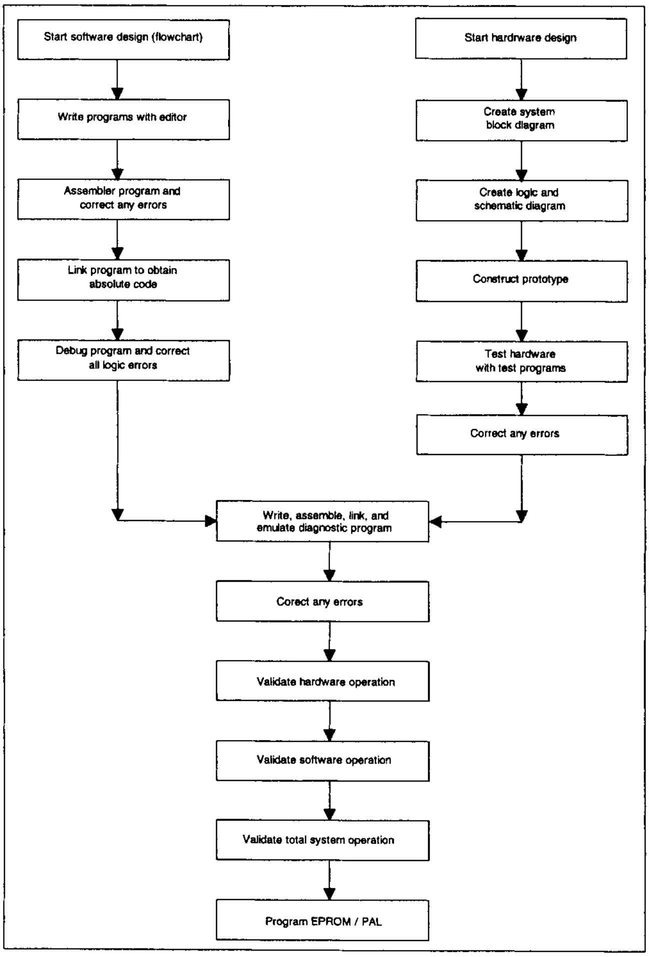

Start software design (flowchart)
Start hardrware design
Write programs with editor
Create system block diagram
Assembler program and correct any errors
Create logic and schematic diagram
Link program to obtain absolute code
Construct prototype
Debug program and correct all logic errors
Test hardware with test programs
Correct any errors
Write, assemble, link, and emulate diagnostic program
Corect any errors
Validate hardware operation
Validate software operation
Validate total system operation
Program EPROM / PAL

FIGURE 6.33 Microprocessor system development flowchart

the final absolute code that will be executed on the target system. The emulation phase will take the absolute code and load it into the development system RAM. From here, the program may be debugged using breakpoints or single stepping.

Working from the system specifications, a block diagram of the hardware must be developed. The logic diagram and schematics may now be drawn using the block diagram as a guide, and a prototype may now be constructed and tested for wiring errors. When the prototype has been constructed it may be debugged for correct operation using standard electronic testing equipment such as oscilloscopes, meters, logic probes, and logic analyzers, all with test programs created for this purpose. After the prototype has been debugged electrically, the development system in-circuit emulator may be used to check it functionally. The ICE will verify the memory map, correct I/O operation, and so on. The next step in system development is to validate the complete system by running operational checks on the prototype with the finalized application software installed. The EPROMs and/or PALs are then programmed with the error-free programs.

QUESTIONS AND PROBLEMS

6.1 What is the difference between a single-chip microprocessor and a single-chip microcomputer?

6.2 What is a microcontroller? Name one commercially available microcontroller.

6.3 What is the difference between:
(a) The program counter (PC) and the memory address register (MAR)?
(b) The accumulator (A) and the instruction register (IR)?
(c) General-purpose register-based microprocessor and accumulator-based microprocessor. Name a commercially available microprocessor of each type.

6.4 Assuming signed numbers, find the sign, carry, zero, and overflow flags of:
(a) $09_{16} + 17_{16}$.
(b) $A5_{16} - A5_{16}$
(c) $71_{16} - A9_{16}$
(d) $6E_{16} + 3A_{16}$
(e) $7E_{16} + 7E_{16}$

6.5 What is meant by PUSH and POP operations in the stack?

6.6 Suppose that an 8-bit microprocessor has a 16-bit stack pointer and uses a 16-bit register to access the stack from the top. Assume that initially the stack pointer and the 16-bit register contain $20C0_{16}$ and 0205_{16} respectively. After the PUSH operation:
(a) What are the contents of the stack pointer?
(b) What are the contents of memory locations $20BE_{16}$ and $20BF_{16}$?

6.7 Assuming the microprocessor architecture of Figure 6.18, write down a possible sequence of microinstructions for finding the ones complement of an 8-bit number. Assume that the number is already in the register.

6.8 What do you mean by a multiplexed address and data bus?

6.9 Name four general-purpose registers in the 8086.

6.10 Name one 8086 register that can be used to hold an address in a segment.

6.11 What is the difference between EPROM and PROM? Are both types available with bipolar and also MOS technologies?

6.12 Assuming a single clock signal and four registers (PC, MAR, Reg, and IR) for a microprocessor, draw a timing diagram for loading the memory address register. Explain the sequence of events relating them to the four registers.

6.13 Given a memory with a 14-bit address and 8-bit word size.
(a) How many bytes can be stored in this memory?
(b) If this memory were constructed from 1K × 1-bit RAMs, how many memory chips would be required?
(c) How many bits would be used for chip select?

6.14 Define the three types of I/O. Identify each one as either "microprocessor initiated" or "device initiated."

6.15 What is the basic difference between a compiler and an assembler?

6.16 Write a program equivalent to the Pascal assignment statement:

```
Z := (A + (B * C) + (D * E) -(F / G) -(H * I)
```

Use only
(a) Three-address instructions
(b) Two-address instructions

6.17 Describe the meaning of each one of the following addressing modes.

(a)	Immediate	(d)	Register indirect
(b)	Absolute	(e)	Relative
(c)	Register	(f)	Implied

6.18 Assume that a microprocessor has only two registers R1 and R2 and that only the following instruction is available:

```
        XOR   Ri, Rj        ;      Rj <- Ri ⊕ Rj
                            ;      i,j = 1,2
```

Using this XOR instruction, find an instruction sequence in order to exchange the contents of registers R1 and R2

6.19 What are the advantages of subroutines?

6.20 Explain the use of a stack in implementing subroutine calls.

6.21 Determine the contents of address 5004_{16} after assembling the following:

(a)
```
ORG  5002H
DB   00H, 05H, 07H, 00H, 03H
```

(b)
```
ORG  5000H
DW   0702H, 123FH, 7020H, 0000H
```

6.22 What is the difference between:
- (a) A cross assembler and a resident assembler
- (b) A two-pass assembler and meta-assembler
- (c) Single step and breakpoint

6.23 Identify some of the differences between C, C++, and Java.

6.24 How does a microprocessor obtain the address of the first instruction to be executed?

6.25 Summarize the basic features of a typical microcomputer development system.

6.26 Discuss the steps involved in designing a microprocessor-based system.