# 2

# Introduction to 8085 Assembly Language Programming

As defined in Chapter 1, an assembly language program is a set of instructions written in the mnemonics of a given microprocessor. These instructions are the commands to the microprocessor to be executed in the given sequence to accomplish a task. To write such programs for the 8085 microprocessor, we should be familiar with the programming model and the instruction set of the microprocessor. This chapter provides such an overview of the 8085 microprocessor.

The 8085 instruction set is classified into five different groups: data transfer, arithmetic, logic, branch, and machine control; each of these groups is illustrated with examples. The chapter also discusses the instruction format and various addressing modes. A simple problem of adding to Hex numbers is used to illustrate writing, assembling, and executing a program. The flowcharting technique and symbols are discussed in the context of the problem. The chapter concludes with a list of selected 8085 instructions.

## OBJECTIVES

- ☐ Explain the various functions of the registers in the 8085 programming model.
- ☐ Define the term *flag* and explain how the flags are affected.
- ☐ Explain the terms *operation code* (opcode) and the *operand,* and illustrate these terms by writing instructions.
- ☐ Classify the instructions in terms of their word size and specify the number of memory registers required to store the instructions in memory.
- ☐ List the five categories of the 8085 instruction set.
- ☐ Define and explain the term *addressing mode.*

- ☐ Write logical steps to solve a simple programming problem.
- ☐ Draw a flowchart from the logical steps of a given programming problem.
- ☐ Translate the flowchart into mnemonics and convert the mnemonics into Hex code for a given programming problem.

## 2.1 THE 8085 PROGRAMMING MODEL

A model is a conceptual representation of a real object. It can take many forms, such as text description, a drawing, or a built structure. Most of us have seen an architectural model of a building. Similarly, the microprocessor can be represented in terms of its hardware (physical electronic components) and a programming model (information needed to write programs). In Chapter 1, we described a simplified hardware model of a microprocessor as a part of the microprocessor-based system (Figure 1.3). It showed three components: ALU, register array, and control. Figure 2.1 shows a hardware model and a programming model specific to the 8085 microprocessor.

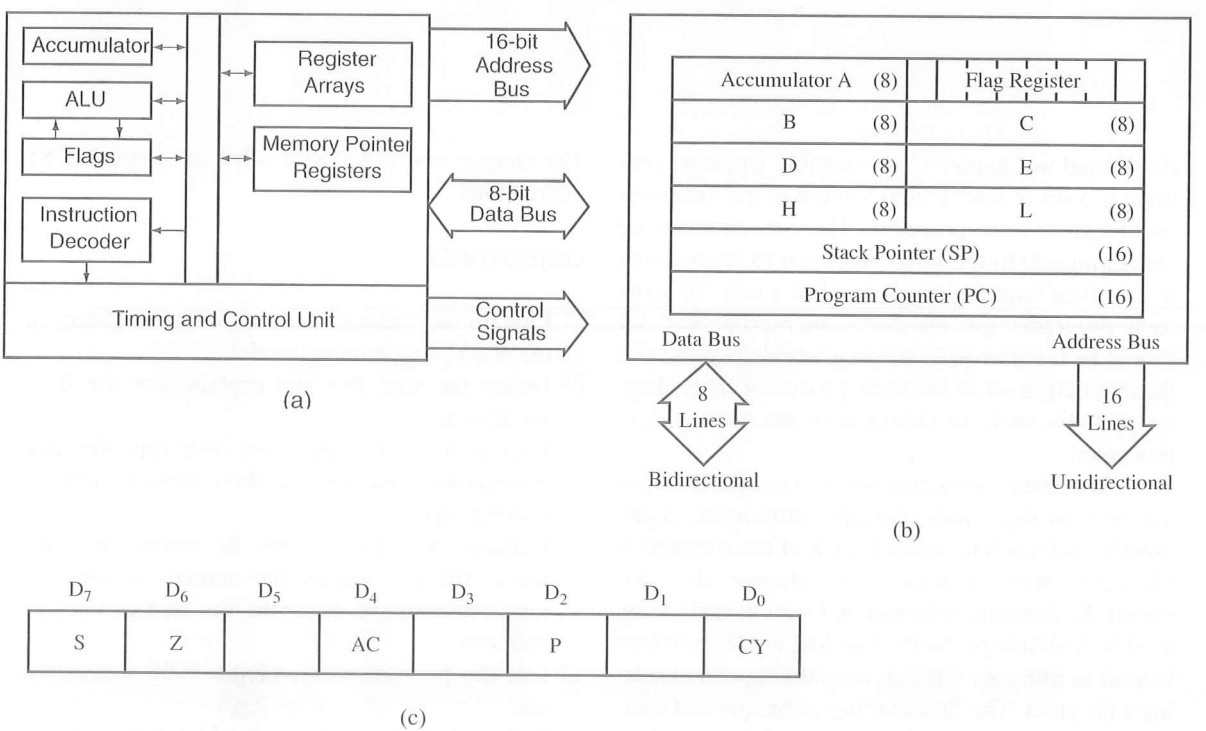



FIGURE 2.1
8085 Hardware Model (a), Programming Model (b), and Flag Register (c)

### 2.1.1 8085 Hardware Model

The hardware model in Figure 2.1(a) shows two major segments. One segment includes the arithmetic/logic unit (ALU) and an 8-bit register called an accumulator, instruction decoder, and flags. The second segment shows 8-bit and 16-bit registers. Both segments are connected with various internal connections called an internal bus. The arithmetic and logical operations are performed in the ALU. Results are stored in the accumulator, and flip-flops, called flags, are set or reset to reflect the results (see Figure 2.1a). There are three buses: a 16-bit unidirectional address bus, an 8-bit bidirectional data bus, and a control bus. In Chapter 1, these three buses were shown as one system bus. The 8085 processor uses the 16-bit address bus to send out memory addresses, the 8-bit data bus to transfer data, and the control bus for timing signals. The details of the hardware model are included in later chapters.

### 2.1.2 8085 Programming Model

The programming model consists of some segments of the ALU and the registers. This model does not reflect the physical structure of the 8085 but includes the information that is critical in writing assembly language programs. The model includes six registers, one accumulator, and one flag register, as shown in Figure 2.1(b). In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows.

#### REGISTERS

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B, C, D, E, H, and L, as shown in Figure 2.1(b). They can be combined as register pairs—BC, DE, and HL—to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

#### ACCUMULATOR

The accumulator is an 8-bit register that is part of the arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

#### FLAGS

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; they are listed in Table 2.1 and their bit positions in the flag register are shown in Figure 2.1(c). The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through software instructions. For example, the instruction JC (Jump On Carry) is implemented to change the sequence of a program when the CY flag is set. The thorough understanding of flags is essential in writing assembly language programs.

TABLE 2.1
The 8085 Flags

The following flags are set or reset after the execution of an arithmetic or logic operation; data copy instructions do not affect any flags. See the instruction set (Appendix F) to find how flags are affected by an instruction.

- ☐ Z—Zero: The Zero flag is set to 1 when the result is zero; otherwise it is reset.
- ☐ CY—Carry: If an arithmetic operation results in a carry, the CY flag is set; otherwise it is reset.
- ☐ S—Sign: The Sign flag is set if bit $D_7$ of the result = 1; otherwise it is reset.
- ☐ P—Parity: If the result has an even number of 1s, the flag is set; for an odd number of 1s, the flag is reset.
- ☐ AC—Auxiliary Carry: In an arithmetic operation, when a carry is generated by digit $D_3$ and passed to digit $D_4$, the AC flag is set. This flag is used internally for BCD (binary-coded decimal) operations; there is no Jump instruction associated with this flag.

### PROGRAM COUNTER (PC) AND STACK POINTER (SP)*

These are two 16-bit registers used to hold memory addresses. The size of these registers is 16 bits because the memory addresses are 16 bits.

The microprocessor uses the PC register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading a 16-bit address in the stack pointer. The stack concept is explained in Chapter 9, "Stack and Subroutines."

This programming model will be used in subsequent chapters to examine how these registers are affected after the execution of an instruction.

## 2.2 INSTRUCTION CLASSIFICATION

An **instruction** is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the **instruction set,** determines what functions the microprocessor can perform. The 8085 microprocessor includes the instruction set of its predecessor, the 8080A, plus two additional instructions.

*The concept of stack memory is difficult to explain at this time; it is not necessary for the reader to understand stack memory until subroutines are discussed. It is included here only to provide continuity in the discussion of programmable registers and microprocessor operations. This concept will be explained more fully in Chapter 9.

### 2.2.1 The 8085 Instruction Set

The 8085 instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

#### DATA TRANSFER (COPY) OPERATIONS

This group of instructions copies data from a location called a source to another location, called a destination, without modifying the contents of the source. In technical manuals, the term *data transfer* is used for this copying function. However, the term *transfer* is misleading; it creates the impression that the contents of a source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

| Types | Examples |
|---|---|
| ☐ Between registers | Copy the contents of register B into register D. |
| ☐ Specific data byte to a register or a memory location | Load register B with the data byte 32H. |
| ☐ Between a memory location and a register | From the memory location 2000H to register B. |
| ☐ Between an I/O device and the accumulator | From an input keyboard to the accumulator. |

#### ARITHMETIC OPERATIONS

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

- ☐ **Addition**—Any 8-bit number, or the contents of a register, or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g., the contents of register B cannot be added directly to the contents of register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.
- ☐ **Subtraction**—Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2's complement, and the results, if negative, are expressed in 2's complement. No two other registers can be subtracted directly.
- ☐ **Increment/Decrement**—The 8-bit contents of a register or a memory location can be incremented or decremented by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decremented by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e., they can be performed in any one of the registers or in a memory location.

LOGICAL OPERATIONS

These instructions perform various logical operations with the contents of the accumulator.

- **AND, OR, Exclusive-OR**—Any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, ORed, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.
- **Rotate**—Each bit in the accumulator can be shifted either left or right to the next position.
- **Compare**—Any 8-bit number, or the contents of a register, or a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.
- **Complement**—The contents of the accumulator can be complemented; all 0s are replaced by 1s and all 1s are replaced by 0s.

BRANCHING OPERATIONS

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

- **Jump**—Conditional jumps are an important aspect of the decision-making process in programming. These instructions test for a certain condition (e.g., Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called *unconditional jump.*
- **Call, Return, and Restart**—These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

MACHINE CONTROL OPERATIONS

These instructions control machine functions such as Halt, Interrupt, or do nothing.

### 2.2.2 Review of the 8085 Operations

The microprocessor operations related to data manipulation can be summarized in four functions:

1. copying data
2. performing arithmetic operations
3. performing logical operations
4. testing for a given condition and altering the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Arithmetic and logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some exceptions). The flags are affected according to the results.

3. Any register including memory can be used for increment and decrement.
4. A program sequence can be changed either conditionally or by testing for a given data condition.

## 2.3 INSTRUCTION, DATA FORMAT, AND STORAGE

An **instruction** is a command to the microprocessor to perform a given task on specified data. Each instruction has two parts: one is the task to be performed, called the **operation code** (opcode), and the second is the data to be operated on, called the **operand.** The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or an 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

### 2.31 Instruction Word Size

The 8085 instruction set is classified into the following three groups according to word size or byte size.

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

1. 1-byte instructions
2. 2-byte instructions
3. 3-byte instructions

#### ONE-BYTE INSTRUCTIONS

A 1-byte instruction includes the opcode and the operand in the same byte. For example:

| Task | Opcode | Operand* | Binary Code | Hex Code |
|---|---|---|---|---|
| Copy the contents of the accumulator in register C. | MOV | C,A | 0100 1111 | 4FH |
| Add the contents of register B to the contents of the accumulator. | ADD | B | 1000 0000 | 80H |
| Invert (complement) each bit in the accumulator. | CMA | | 0010 1111 | 2FH |

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8-bit binary format in memory; each requires one memory location.

*In the operand, the destination register C is shown first, followed by the source register A.

### TWO-BYTE INSTRUCTIONS

In a 2-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. For example:

| Task | Opcode | Operand | Binary Code | Hex Code | |
|---|---|---|---|---|---|
| Load an 8-bit data byte in the accumulator. | MVI | A,32H | 0011 1110 | 3E | First Byte |
| | | | 0011 0010 | 32 | Second Byte |
| Load an 8-bit data byte in register B. | MVI | B,F2H | 0000 0110 | 06 | First Byte |
| | | | 1111 0010 | F2 | Second Byte |

These instructions would require two memory locations each to store the binary codes. The data bytes 32H and F2H are selected arbitrarily as examples.

### THREE-BYTE INSTRUCTIONS

In a 3-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address. For example:

| Task | Opcode | Operand | Binary Code | Hex Code* | |
|---|---|---|---|---|---|
| Load contents of memory 2050H into A. | LDA | 2050H | 0011 1010 | 3A | First Byte |
| | | | 0101 0000 | 50 | Second Byte |
| | | | 0010 0000 | 20 | Third Byte |
| Transfer the program sequence to memory location 2085H. | JMP | 2085H | 1100 0011 | C3 | First Byte |
| | | | 1000 0101 | 85 | Second Byte |
| | | | 0010 0000 | 20 | Third Byte |

These instructions would require three memory locations each to store the binary codes.

These commands are in many ways similar to our everyday conversation. For example, while eating in a restaurant, we may make the following requests and orders:

1. Pass (the) butter.
2. Pass (the) bowl.
3. (Let us) eat.

---

*The 16-bit addresses are stored in memory locations in reversed order, the low-order byte first, followed by the high-order byte.

4. I will have combination 17 (on the menu).
5. I will have what Susie ordered.

The first request specifies the exact item; it is similar to the instruction for loading a specific data byte in a register. The second request mentions the bowl rather than the contents, even though one is interested in the contents of the bowl. It is similar to the instruction MOV C,A where registers (bowls) are specified rather than data. The third suggestion (let us eat) assumes that one knows what to eat. It is similar to the instruction Complement, which implicitly assumes that the operand is the accumulator. In the fourth sentence, the location of the item on the menu is specified and not the actual item. It is similar to the instruction: Transfer the data byte from the location 2050H. The last order (what Susie ordered) is specified indirectly. It is similar to an instruction that specifies a memory location through the contents of a register pair. (Examples of the last two types of instruction are illustrated in later chapters.)

These various ways of specifying data are called the **addressing modes.** Although microprocessor instructions require one or more words to specify the operands, the notations and conventions used in specifying the operands have very little to do with the operation of the microprocessor. The mnemonic letters used to specify a command are chosen (somewhat arbitrarily) by the manufacturer. When an instruction is stored in memory, it is stored in binary code, the only code the microprocessor is capable of reading and understanding. The conventions used in specifying the instructions are valuable in terms of keeping uniformity in different programs and in writing assemblers. The important point to remember is that the microprocessor neither reads nor understands mnemonics or hexadecimal numbers.

### 2.3.2 Opcode Format

To understand operation codes, we need to examine how an instruction is designed into the microprocessor. This information will be useful in reading a user's manual, in which operation codes are specified in binary format and 8-bits are divided in various groups. However, this information is not necessary to understand assembly language programming.

In the design of the 8085 microprocessor chip, all operations, registers, and status flags are identified with a specific code. For example, all internal registers are identified as follows:

| Code | Registers | Code | Register Pairs |
|---|---|---|---|
| 000 | B | 00 | BC |
| 001 | C | 01 | DE |
| 010 | D | 10 | HL |
| 011 | E | 11 | AF OR SP |
| 100 | H | | |
| 101 | L | | |
| 111 | A | | |
| 110 | Reserved for Memory-Related operation | | |

Some of the operation codes are identified as follows:

| Function | Operation Code |
|---|---|
| **1.** Rotate each bit of the accumulator to the left by one position. | 00000111 = 07H (8-bit opcode) |
| **2.** Add the contents of a register to the accumulator. | 10000 SSS (5-bit opcode—3 bits are reserved for a register) |

This instruction is completed by adding the code of the register. For example,

Add : 10000
Register B : 000
to A : Implicit
Binary Instruction: 10000 000 = 80H
Add Reg.B

In assembly language, this is expressed as

| Opcode | Operand | Hex Code |
|---|---|---|
| ADD | B | 80H |

| | | | |
|---|---|---|---|
| **3.** MOVE (Copy) the content of register Rs (source) to register Rd (destination) | 01 2-bit Opcode for MOVE | DDD Reg. Rd | SSS Reg. Rs |

This instruction is completed by adding the codes of two registers. For example,

Move (copy) the content: 0 1
To register C : 0 0 1 (DDD)
From register A : 1 1 1 (SSS)
Binary Instruction : 0 1 0 0 1 1 1 1 → 4FH
Opcode Operand

In assembly language, this is expressed as

| Opcode | Operand | Hex Code |
|---|---|---|
| MOV | C,A | 4F |

Please note that the first register is the destination and the second register is the source—from A to C—which appears reversed for a general pattern from left to right. Typically, in the 8085 user's manual the data transfer (copy) instruction is shown as follows:

MOV rl, r2*

| 0 | 1 | D | D | D | S | S | S |
|---|---|---|---|---|---|---|---|

### 2.3.3 Data Format

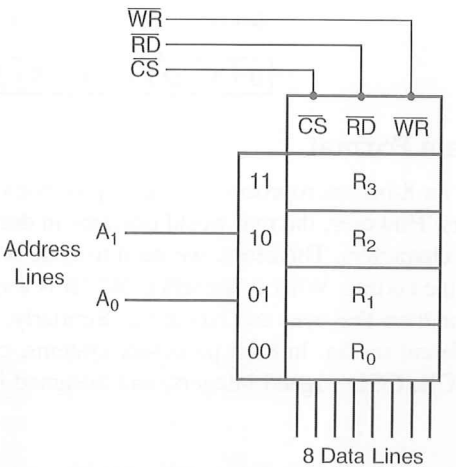
The 8085 is an 8-bit microprocessor, and it processes (copy, add, subtract, etc.) only binary numbers. However, the real world operates in decimal numbers and languages of alphabets and characters. Therefore, we need to code binary numbers into different media. Let us examine coding. What is the letter "A"? It is a symbol representing a certain sound in a visual medium that eyes can recognize. Similarly, we can represent or code groups of bits into different media. In 8-bit processor systems, commonly used codes and data formats are ASCII, BCD, signed integers, and unsigned integers. They are explained as follows.

- **ASCII Code**—This is a 7-bit alphanumeric code that represents decimal numbers, English alphabets, and nonprintable characters such as carriage return. Extended ASCII is an 8-bit code. The additional numbers (beyond 7-bit ASCII code) represent graphical characters. This code was discussed in Chapter 1 (Section 1.24).
- **BCD Code**—The term *BCD* stands for binary-coded decimal; it is used for decimal numbers. The decimal numbering system has ten digits, 0 to 9. Therefore, we need only four bits to represent ten digits from 0000 to 1001. The remaining numbers, 1010 (A) to 1111 (F), are considered invalid. An 8-bit register in the 8085 can accommodate two BCD numbers.
- **Signed Integer**—A signed integer is either a positive number or a negative number. In an 8-bit processor, the most significant digit, $D_7$, is used for the sign; 0 represents the positive sign and 1 represents the negative sign. The remaining seven bits, $D_6$–$D_0$, represent the magnitude of an integer. Therefore, the largest positive integer that can be processed by the 8085 at one time is 0111 1111 (7FH); the remaining Hex numbers, 80H to FFH, are considered negative numbers. However, all negative numbers in this microprocessor are represented in 2's complement format (see Appendix A.2 for additional explanation).
- **Unsigned Integers**—An integer without a sign can be represented by all the 8 bits in a microprocessor register. Therefore, the largest number that can be processed at one time is FFH. However, this does not imply that the 8085 microprocessor is limited to handling only 8-bit numbers. Numbers larger than 8 bits (such as 16-bit or 24-bit numbers) are processed by dividing them in groups of 8 bits.

Now let us examine how the microprocessor interprets any number. Let us assume that after performing some operations the result in the accumulator is 0100 0001 (41H). This number can have many interpretations: (1) It is an unsigned number equivalent to 65

*In this text, rl is specified as Rd and r2 is specified as Rs to indicate destination and source.

FIGURE 2.2
Simplified Memory Model




in decimal; (2) it is a BCD number representing 41 decimal; (3) it is the ASCII capital letter "A"; or (4) it is a group of 8 bits where bits $D_6$ and $D_0$ turn on and the remaining bits turn off output devices. The processor processes binary bits; it is up to the user to interpret the result. In our example, the number 41H can be displayed on a screen as an ASCII "A" or 41 BCD.

### 2.3.4 Instruction and Data Storage: Memory

Now the next question is: How do we provide this information to the processor? It is provided by another electronic storage chip called memory. In some ways, the term *memory* is a misnomer; it is a storage of binary bits. Memory chips used in most systems are nothing but 8-bit registers stacked one above the other as shown in our memory model in Figure 2.2. It includes only four registers, and each register can store 8 bits. This chip can be referred to as a 4-byte or 32 ($4 \times 8$) bits memory chip. It has two address lines, $A_0$ and $A_1$, to identify four registers, 8 data lines to store 8 bits, and three timing or control signals: Read ($\overline{RD}$), Write ($\overline{WR}$), and Chip Select ($\overline{CS}$); all control signals are designed to be active low, indicated by bars over the symbols. The processor can select this chip and identify its register, and store (Write) or access (Read) 8 bits at a time. Figure 2.2 shows only four registers to simplify the explanation; in reality, the size of a memory chip is in kilo- or megabytes. The memory addresses assigned to these registers are determined by the interfacing logic used in the system. In Chapters 3 and 4, we will discuss memory addressing and interfacing in more detail.

## 2.4 HOW TO WRITE, ASSEMBLE, AND EXECUTE A SIMPLE PROGRAM

A program is a sequence of instructions written to tell a computer to perform a specific function. The instructions are selected from the instruction set of the microprocessor. To write a program, divide a given problem in small steps in terms of the operations the 8085

can perform, then translate these steps into instructions. Writing a simple program of adding two numbers in the 8085 language is illustrated below.

## 2.4.1 Illustrative Program: Adding Two Hexadecimal Numbers

### PROBLEM STATEMENT

Write instructions to load the two hexadecimal numbers 32H and 48H in registers A and B, respectively. Add the numbers, and display the sum at the LED output port PORT1.

### PROBLEM ANALYSIS

Even though this is a simple problem, it is necessary to divide the problem into small steps to examine the process of writing programs. The wording of the problem provides sufficient clues for the necessary steps. They are as follows:

1. Load the numbers in the registers.
2. Add the numbers.
3. Display the sum at the output port PORT1.

### FLOWCHART

The steps listed in the problem analysis and the sequence can be represented in a block diagram, called a flowchart. Figure 2.3 shows such a flowchart representing the above steps. This is a simple flowchart, and the steps are self-explanatory. We will discuss flowcharting in the next chapter.

### ASSEMBLY LANGUAGE PROGRAM

To write an assembly language program, we need to translate the blocks shown in the flowchart into 8085 operations and then, subsequently, into mnemonics. By examining the blocks, we can classify them into three types of operations: Blocks 1 and 3 are copy op-

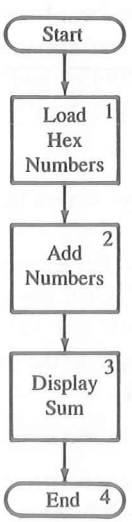



FIGURE 2.3
Flowchart: Adding Two Numbers

erations; Block 2 is an arithmetic operation; and Block 4 is a machine-control operation. To translate these steps into assembly and machine languages, you should review the instruction set. The translation of each block into mnemonics with comments is shown as follows:

| | | |
|---|---|---|
| Block 1: | MVI A,32H | Load register A with 32H |
| | MVI B,48H | Load register B with 48H |
| Block 2: | ADD B | Add two bytes and save the sum in A |
| Block 3: | OUT 01H | Display accumulator contents at port 01H |
| Block 4: | HALT | End |

### FROM ASSEMBLY LANGUAGE TO HEX CODE

To convert the mnemonics into Hex code, we need to look up the code in the 8085 instruction set; this is called either manual or hand assembly.

| Mnemonics | Hex Code | |
|---|---|---|
| MVI A,32H | 3E | 2-byte instruction |
| | 32 | |
| MVI B,48H | 06 | 2-byte instruction |
| | 48 | |
| ADD B | 80 | 1-byte instruction |
| OUT 01H | D3 | 2-byte instruction |
| | 01 | |
| HLT | 76 | 1-byte instruction |

### STORING IN MEMORY AND CONVERTING FROM HEX CODE TO BINARY CODE

To store the program in R/W memory of a single-board microcomputer and display the output, we need to know the memory addresses and the output port address. Let us assume that R/W memory ranges from 2000H to 20FFH, and the system has an LED output port with the address 01H. Now, to enter the program:

1. Reset the system by pushing the RESET key.
2. Enter the first memory address using Hex keys where the program should be stored. Let us assume it is 2000H.
3. Enter each machine code by pushing Hex keys. For example, to enter the first machine code, push the 3, E, and STORE keys. (The STORE key may be labeled differently in different systems.) When you push the STORE key, the program will store the machine code in memory location 2000H and upgrade the memory address to 2001H.
4. Repeat Step 3 until the last machine code, 76H.
5. Reset the system.

Now the question is: How does the Hex code get converted into binary code? The answer lies with the Monitor program stored in Read-Only memory (or EPROM) of the micro-

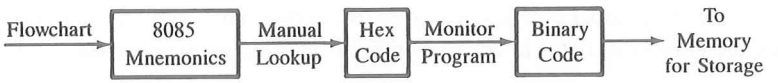



FIGURE 2.4
Manual Assembly Process

computer system. An important function of the Monitor program is to check the keys and convert Hex code into binary code. The entire process of manual assembly is shown in Figure 2.4.

In this illustrative example, the program will be stored in memory as follows:

| Mnemonics | Hex Code | Memory Contents | Memory Address |
|---|---|---|---|
| MVI A,32H | 3E | 0 0 1 1 1 1 1 0 | 2000 |
| | 32 | 0 0 1 1 0 0 1 0 | 2001 |
| MVI B,48H | 06 | 0 0 0 0 0 1 1 0 | 2002 |
| | 48 | 0 1 0 0 1 0 0 0 | 2003 |
| ADD B | 80 | 1 0 0 0 0 0 0 0 | 2004 |
| OUT 01H | D3 | 1 1 0 1 0 0 1 1 | 2005 |
| | 01 | 0 0 0 0 0 0 0 1 | 2006 |
| HLT | 76 | 0 1 1 1 1 1 1 0 | 2007 |

This program has eight machine codes and will require eight memory locations to store the program. The critical concept that needs to be emphasized here is that the microprocessor can understand and execute only the binary instructions (or data); everything else (mnemonics, Hex code, comments) is for the convenience of human beings.

## EXECUTING THE PROGRAM

To execute the program, we need to tell the microprocessor where the program begins by entering the memory address 2000H. Now, we can push the Execute key (or the key with a similar label) to begin the execution. As soon as the Execute function key is pushed, the microprocessor loads 2000H in the program counter, and the program control is transferred from the Monitor program to our program.

The microprocessor begins to read one machine code at a time, and when it fetches the complete instruction, it executes that instruction. For example, it will fetch the machine codes stored in memory locations 2000H and 2001H and execute the instruction MVI A,32H; thus it will load 32H in register A. The ADD instruction will add the two numbers, and the OUT instruction will display the answer 7AH (32H + 48H = 7AH) at the LED port. It continues to execute instructions until it fetches the HLT instruction.

## RECOGNIZING THE NUMBER OF BYTES IN AN INSTRUCTION

Students who are introduced to an assembly language for the first time should hand assemble at least a few small programs. Such exercises can clarify the relationship among instruction codes, data, memory registers, and memory addressing. One of the stumbling blocks in hand

assembly is in recognizing the number of bytes in a given instruction. The following clues can be used to recognize the number of bytes in an instruction of the 8085 microprocessor.

1. One-byte instruction—A mnemonic followed by a letter (or two letters) representing the registers (such as A, B, C, D, E, H, L, M, and SP) is a one-byte instruction. Instructions in which registers are implicit are also one-byte instructions.
   Examples: (a) MOV A, B; (b) DCX SP; (c) RRC
2. Two-byte instruction—A mnemonic followed by 8-bit (byte) is a two-byte instruction.
   Examples: (a) MVI A, 8-bit; (b) ADI 8-bit
3. Three-byte instruction—A mnemonic followed by 16-bit (also terms such as adr or dble) is a three-byte instruction.
   Examples: (a) LXI B, 16-bit (dble); (b) JNZ 16-bit (adr); (c) CALL 16-bit (adr)

In writing assembly language programs, we can assign memory addresses in a sequence once we know the number of bytes in a given instruction. For example, a three-byte instruction has three Hex codes and requires three memory locations in a sequence. In hand assembly, omitting a byte inadvertently can have a disastrous effect on program execution, as explained in the next section.

### 2.4.2 How Does a Microprocessor Differentiate Between Data and Instruction Code?

The microprocessor is a sequential machine. As soon as a microprocessor-based system is turned on, it begins the execution of the code in memory. The execution continues in a sequence, one code after another (one memory location after another) at the speed of its clock until the system is turned off (or the clock stops). If an unconditional loop is set up in a program, the execution will continue until the system is either reset or turned off.

Now a puzzling question is: How does the microprocessor differentiate between a code and data when both are binary numbers? The answer lies in the fact that the microprocessor interprets the first byte it fetches as an opcode. When the 8085 is reset, its program counter is cleared to 0000H and it fetches the first code from the location 0000H. In the example of the previous section, we tell the processor that our program begins at location 2000H. The first code it fetches is 3EH. When it decodes that code, it knows that it is a two-byte instruction. Therefore, it assumes that the second code, 32H, is a data byte. If we forget to enter 32H and enter the next code, 06H, instead, the 8085 will load 06H in the accumulator, interpret the next code, 48H, as an opcode, and continue the execution in sequence. As a consequence, we may encounter a totally unexpected result.

## 2.5 OVERVIEW OF THE 8085 INSTRUCTION SET

The 8085 microprocessor instruction set has 74 operation codes that result in 246 instructions. The set includes all the 8080A instructions plus two additional instructions (SIM and RIM, related to serial I/O). It is an overwhelming experience for a beginner to

study these instructions. You are strongly advised not to attempt to read all these instructions at one time. However, you should be able to grasp an overview of the set by examining the frequently used instructions listed below.*

The following notations are used in the description of the instructions.

| | |
|---|---|
| R = 8085 8-bit register | (A, B, C, D, E, H, L) |
| M = Memory register (location) | |
| Rs = Register source | |
| Rd = Register destination | (A, B, C, D, E, H, L) |
| Rp = Register pair | (BC, DE, HL, SP) |
| ( ) = Contents of | |

**1. Data Transfer (Copy) Instructions.** These instructions perform the following six operations.

- Load an 8-bit number in a register
- Copy from register to register
- Copy between I/O and accumulator
- Load 16-bit number in a register pair
- Copy between register and memory
- Copy between registers and stack memory

| Mnemonics | | Examples | Operation |
|---|---|---|---|
| 1.1 | MVI R,** 8-bit | MVI B, 4FH | Load 8-bit data (byte) in a register |
| 1.2 | MOV Rd, Rs** | MOV B, A<br>MOV C, B | Copy data from source register Rs into destination register Rd |
| 1.3 | LXI Rp,** 16-bit | LXI B, 2050H | Load 16-bit number in a register pair |
| 1.4 | OUT 8-bit (port address) | OUT 01H | Send (write) data byte from the accumulator to an output device |
| 1.5 | IN 8-bit (port address) | IN 07H | Accept (read) data byte from an input device and place it in the accumulator |
| 1.6 | LDA 16-bit | LDA 2050H | Copy the data byte into A from the memory specified by 16-bit address |
| 1.7 | STA 16-bit | STA 2070H | Copy the data byte from A into the memory specified by 16-bit address |
| 1.8 | LDAX Rp | LDAX B | Copy the data byte into A from the memory specified by the address in the register pair |
| 1.9 | STAX Rp | STAX D | Copy the data byte from A into the memory specified by the address in the register pair |

*These instructions are explained and illustrated in Chapters 6 and 7. The complete instruction set is explained alphabetically in Appendix F for easy reference; the appendix also includes three lists of instruction summaries arranged according to the functions, hexadecimal sequence of machine codes, and alphabetical order.

**The letters R, Rd, Rs, Rp represent generic registers. In the 8085 instructions, these are replaced by registers such as A, B, C, D, E, H, and L or register pairs.

| | | | |
|---|---|---|---|
| 1.10 | MOV R, M | MOV B, M | Copy the data byte into register from the memory specified by the address in HL register |
| 1.11 | MOV M, R | MOV M, C | Copy the data byte from the register into memory specified by the address in HL register |

**2. Arithmetic Instructions.** The frequently used arithmetic operations are:

- Add
- Subtract
- Increment (Add 1)
- Decrement (Subtract 1)

| Mnemonics | | Examples | Operation |
|---|---|---|---|
| 2.1 | ADD R | ADD B | Add the contents of a register to the register to the contents of A |
| 2.2 | ADI 8-bit | ADI 37H | Add 8-bit data to the contents of A |
| 2.3 | ADD M | ADD M | Add the contents of memory to A; the address of memory is in HL register |
| 2.4 | SUB R | SUB C | Subtract the contents of a register from the contents of A |
| 2.5 | SUI 8-bit | SUI 7FH | Subtract 8-bit data from the contents of A |
| 2.6 | SUB M | SUB M | Subtract the contents of memory from A; the address of memory is in HL register |
| 2.7 | INR R | INR D | Increment the contents of a register |
| 2.8 | INR M | INR M | Increment the contents of memory, the address of which is in HL |
| 2.9 | DCR R | DCR E | Decrement the contents of a register |
| 2.10 | DCR M | DCR M | Decrement the contents of a memory, the address of which is in HL |
| 2.11 | INX Rp | INX H | Increment the contents of a register pair |
| 2.12 | DCX Rp | DCX B | Decrement the contents of a register pair |

**3. Logic and Bit Manipulation Instructions.** These instructions include the following operations:

- AND
- OR
- X-OR (Exclusive OR)
- Compare
- Rotate Bits

| Mnemonics | | Examples | Operation |
|---|---|---|---|
| 3.1 | ANA R | ANA B | Logically AND the contents of a register with the contents of A |

| | | | |
|---|---|---|---|
| 3.2 | ANI 8-bit | ANI 2FH | Logically AND 8-bit data with the contents of A |
| 3.3 | ANA M | ANA M | Logically AND the contents of memory with the contents of A; the address of memory is in HL register |
| 3.4 | ORA R | ORA E | Logically OR the contents of a register with the contents of A |
| 3.5 | ORI 8-bit | ORI 3FH | Logically OR 8-bit data with the contents of A |
| 3.6 | ORA M | ORA M | Logically OR the contents of memory with the contents of A; the address of memory is in HL register |
| 3.7 | XRA R | XRA B | Exclusive-OR the contents of a register with the contents of A |
| 3.8 | XRI 8-bit | XRI 6AH | Exclusive-OR 8-bit data with the contents of A |
| 3.9 | XRA M | XRA M | Exclusive-OR the contents of memory with the contents of A; the address of memory is in HL register |
| 3.10 | CMP R | CMP B | Compare the contents of register with the contents of A for less than, equal to, or greater than |
| 3.11 | CPI 8-bit | CPI 4FH | Compare 8-bit data with the contents of A for less than, equal to, or greater than |

**4. Branch Instructions.** The following instructions change the program sequence.

| | | | |
|---|---|---|---|
| 4.1 | JMP 16-bit address | JMP 2050H | Change the program sequence to the specified 16-bit address |
| 4.2 | JZ 16-bit address | JZ 2080H | Change the program sequence to the specified 16-bit address if the Zero flag is set |
| 4.3 | JNZ 16-bit address | JNZ 2070H | Change the program sequence to the specified 16-bit address if the Zero flag is reset |
| 4.4 | JC 16-bit address | JC 2025H | Change the program sequence to the specified 16-bit address if the Carry flag is set |
| 4.5 | JNC 16-bit address | JNC 2030H | Change the program sequence to the specified 16-bit address if the Carry flag is reset |
| 4.6 | CALL 16-bit address | CALL 2075H | Change the program sequence to the location of a subroutine |
| 4.7 | RET | RET | Return to the calling program after completing the subroutine sequence |

5. **Machine Control Instructions.** These instructions affect the operation of the processor.

| | | | |
|---|---|---|---|
| 5.1 | HLT | HLT | Stop processing and wait |
| 5.2 | NOP | NOP | Do not perform any operation |

This set of instructions is a representative sample; it does not include various instructions related to 16-bit data operations, additional jump instructions, and conditional Call and Return instructions.

## 2.6 WRITING AND HAND ASSEMBLING A PROGRAM

In previous sections, we discussed the 8085 instructions, recognized the number of bytes per instruction, looked at the relationship between the number of bytes of an instruction and memory registers needed for storage, and examined the processor's computing capability in the overview of the instruction set. Now let us pull together all these concepts in a simple illustrative program.

### 2.6.1 Illustrative Program: Subtracting Two Hexadecimal Numbers and Storing the Result in Memory

PROBLEM STATEMENT

Write instructions to subtract two bytes already stored in memory registers (also referred to as memory locations or memory addresses) 2051H and 2052H. Location 2051H holds the byte 49H and location 2051H holds the byte 9FH. Subtract the first byte, 49H, from the second byte, 9FH, and store the answer in memory location 2053H. Write instructions beginning at memory location 2030H.

PROBLEM ANALYSIS

This is a problem similar to the problem in Section 2.4. However, we need to note some specific points in this problem.

1. The data bytes to be subtracted are already stored in memory registers 2051H and 2052H. We do not need to write instructions to store these bytes. You should store these bytes by using a keyboard of your trainer, or if you are using a simulator, you should observe the numbers in those memory locations when you store them.
2. The program should be written starting at memory location 2030H. This memory location is selected arbitrarily to emphasize that you can write a program beginning at any available memory location.

3. The microprocessor performs arithmetic operations in the ALU, meaning the processor must use accumulator A in performing the subtraction.
4. The data bytes must be copied from memory into the microprocessor registers (until you learn how to perform an arithmetic operation by using the accumulator and a memory register).

## WRITING MNEMONICS AND ASSEMBLING HEX CODE

The flowchart for this problem is similar to that shown in Figure 2.3. The steps in writing instructions are as follows:

1. Copy two data bytes into processor registers. By examining Section 2.5, we find two instructions, LDA and STA (instructions 1.6 and 1.7), to copy a byte from memory into A and from A into memory. There are two other instructions (MOV R, M and MOV M, R) that can copy between memory and registers, but those instructions require the concept of memory pointers, which will be discussed in Chapter 7. Now let us take a look at the instruction LDA in the 8085 Instruction Summary: Alphabetical Order, inside the back cover.
   a. **The instruction is: LDA 16-bit.** LDA is the opcode with Hex code 3A. (For a complete description of the instruction LDA, see Appendix F—all instructions are explained in alphabetical order.) As we discussed in Section 2.4 ("Recognizing the Number of Bytes in an Instruction"), this must be a 3-byte instruction and will require three memory locations.
   b. **The operand is a 16-bit address** of the memory location from which we want to copy the byte into A. First we want to copy a byte from memory location 2051H.
   c. **The 3-byte code is 3A 51 20.** See Section 2.3. A 16-bit address is always written in reverse order—low-order byte followed by high-order byte. Our program begins at location 2030H; therefore, these 3 bytes will be stored in locations with Hex addresses 2030, 31, and 32. And the instruction, when executed, will copy the first byte, 49H, into A.
   d. **Copying the second byte into A.** If we copy the second byte, 9FH, from memory location 2052H into A, we will destroy the first byte, 49H. Therefore, the previous byte, 49H, should be stored first in some other register such as B.
2. As discussed in 1d above, the second instruction should be to copy the byte from A into B. The instruction is: MOV Rd, Rs (instruction 1.2 in Section 2.5), copying from one source register Rs to another destination register Rd. The terms Rs and Rd are generic; they represent any register A through L. The instruction is:
   **MOV B, A with Hex code 47.** Copy A into B—note the reversed order of A and B. This is a 1-byte instruction.
3. Now we can copy the second byte (9FH) from memory location 2052H into A by using the instruction
   **LDA 2052H with the 3-byte Hex code 3A 52 20.**
4. The next step is to subtract B from A. If we look in Section 2.5 under Arithmetic Instructions, we find three instructions: 2.4, SUB R; 2.5, SUI 8-bit; and 2.6, SUB M.

The instruction is:
**SUB B with Hex code 90.** (Look at the inside of the back cover to find the code.) This instruction subtracts B from A and saves the result in A.

5. The result in A should be stored in memory location 2053H. The instruction is **STA 2053 with Hex code 32 53 20.**
6. Each program must be terminated; otherwise, the processor continues to fetch and execute instructions from the remaining memory registers until it gets lost or caught up in an infinite loop. This step may appear trivial, but it is essential. Therefore, the last instruction is
**HALT with Hex code 76.** In instructional trainers, the Restart (RST) instruction is used to pass the control of running programs back to the monitor program of the trainer.
7. Now we need to load the two data bytes 49H and 9FH in memory locations 2051H and 2052H. This step is a manual entry of the data bytes, independent of the program.
8. So far we have completed two steps: writing the program (as shown in Column 1 in Table 2.2) and entering the Hex code in the memory registers of a memory chip (as shown in Column 2 in Table 2.2). If you observe this program in a simulator (see Appendix H), it should appear as shown in Column 2. Using the analogy of putting a radio kit together, we now have a page of instructions. Now we need to find the page, begin to read, understand the instructions, and perform the task until the kit is

TABLE 2.2
Illustrative Program: Assembly

| Column 1 | Column 2 | | Column 3 |
|---|---|---|---|
| Instructions | Memory Addresses | Hex Code | Comments |
| LDA 2051H | 2030 | 3A | Copy the first byte, 49H, from memory location 2051H into A |
| | 2031 | 51 | |
| | 2032 | 20 | |
| MOV B, A | 2033 | 47 | Save the first byte in B |
| LDA 2052H | 2034 | 3A | Copy the second byte, 9FH, from memory location 2052H into A |
| | 2035 | 52 | |
| | 2036 | 20 | |
| SUB B | 2037 | 90 | Subtract 49H from 9FH and save the result in A |
| STA 2053H | 2038 | 32 | Save the result in memory location 2053H |
| | 2039 | 53 | |
| | 203A | 20 | |
| HLT or RST7 | 203B | 76/FF | End of the program |
| | 2051 | 49 | |
| | 2052 | 9F | These data bytes must be manually loaded; they are not part of writing the program |
| | 2053 | 00 | |

built or we take a break. Entering Hex code in memory is similar to the instruction page of a radio kit. Now we need to tell the processor where our instructions begin and ask it to execute those instructions.

9. Our program begins at location 2030H. We need to let the processor know the starting location through a keyboard or simulation.
10. Now we have two choices: let the processor execute one instruction at a time, called Single Step, or execute (run) the entire program. The Single Step execution is easily observed in a simulator (see Appendix H).
11. Initially, it is strongly advisable to follow the Single Step execution. As you step through each instruction, you can observe changes in the contents of registers. When you execute SUB B, the accumulator should have 56H (9FH–49H), and when you execute STA 2053H, the contents of memory location 2053H should change from 00 to 56H.

### COMMON ERRORS

When we learn a new language, we make errors such as misspelling and mispronouncing words. Similar errors happen in writing assembly language instructions. But the processor demands the exact syntax. A list of common errors is as follows, with examples from our problem:

1. LDA 2051H: Not entering the code of the 16-bit address in reversed order.
2. Forgetting to enter the code for the operand, such as 2051H.
3. MOV B, A: Assuming that this copies from B to A.
4. Incrementing the address in decimal, from 2039H to 2040H.
5. HLT: Not terminating a program.
6. Confusing the entering of Hex code in memory as executing a program.

## SUMMARY

This chapter described the data manipulation functions of the 8085 microprocessor, provided an overview of the instruction set, and illustrated the execution of instructions in relation to the system's clock. The important concepts in this chapter can be summarized as follows.

- ☐ The 8085 microprocessor operations are classified into five major groups: data transfer (copy), arithmetic, logic, branch, and machine control.
- ☐ An instruction has two parts: opcode (operation to be performed) and operand (data to be operated on). The operand can be data (8- or 16-bit), address, or register, or it can be implicit. The method of specifying an operand (directly, indirectly, etc.) is called the addressing mode.
- ☐ The instruction set is classified in three groups according to the word size: 1-, 2-, or 3-byte instructions.
- ☐ To write an assembly language program, divide the given problem into small steps in terms of the microprocessor operations, translate these steps into assembly language instructions, and then translate them into the 8085 machine code.

# 3

# Microprocessor Architecture and Microcomputer Systems

A microcomputer system consists of four components—the microprocessor, memory, and I/O (input/output)—as discussed in Chapter 1. The microprocessor manipulates data, controls the timing of various operations, and communicates with such peripherals (devices) as memory and I/O. The internal logic design of the microprocessor, called its **architecture,** determines how and when various operations are performed by the microprocessor. The system bus provides paths for the flow of binary information (data and instructions).

This chapter expands on the bus concept discussed in Chapter 1 and shows how binary information flows externally among the components of the system. The chapter deals with the internal architecture and various operations of the microprocessor in the context of the 8085. It also expands on topics such as memory and I/O, and reviews interfacing devices, such as buffers, decoders, and latches.

## OBJECTIVES

- ☐ List the four operations commonly performed by the microprocessor or the microprocessing unit (MPU).
- ☐ Define the address bus, the data bus, and the control bus, and explain their functions in reference to the 8085 microprocessor.
- ☐ Explain the functions Reset, Interrupt, Wait, and Hold.
- ☐ Explain memory organization and memory map, and explain how memory addresses are assigned to a memory chip.

- ☐ List the types of memory and their functions.
- ☐ Explain the difference between the peripheral I/O (also known as I/O-mapped I/O) and the memory-mapped I/O.
- ☐ Describe the steps in executing an instruction in a bus-oriented system.
- ☐ Define tri-state logic and explain the functions of such MSI devices as buffers, decoders, encoders, and latches.

## 3.1 MICROPROCESSOR ARCHITECTURE AND ITS OPERATIONS

The microprocessor is a programmable digital device, designed with registers, flip-flops, and timing elements. The microprocessor has a set of instructions, designed internally, to manipulate data and communicate with peripherals. This process of data manipulation and communication is determined by the logic design of the microprocessor, called the **architecture.**

The microprocessor can be programmed to perform functions on given data by selecting necessary instructions from its set. These instructions are given to the microprocessor by writing them into its memory. Writing (or entering) instructions and data is done through an input device such as a keyboard. The microprocessor reads or transfers one instruction at a time, matches it with its instruction set, and performs the data manipulation indicated by the instruction. The result can be stored in memory or sent to such output devices as LEDs or a CRT terminal. In addition, the microprocessor can respond to external signals. It can be interrupted, reset, or asked to wait to synchronize with slower peripherals. All the various functions performed by the microprocessor can be classified in three general categories:

- ☐ Microprocessor-initiated operations
- ☐ Internal operations
- ☐ Peripheral (or externally initiated) operations

To perform these functions, the microprocessor requires a group of logic circuits and a set of signals called control signals. However, early processors did not have the necessary circuitry on one chip; the complete units were made up of more than one chip. Therefore, the term *microprocessing unit* (MPU) is defined here as a group of devices that can perform these functions with the necessary set of control signals. This term is similar to the term *central processing unit* (CPU). However, later microprocessors include most of the necessary circuitry to perform these operations on a single chip. Therefore, the terms *MPU* and *microprocessor* often are used synonymously.

The microprocessor functions listed above are explained here in relation to the 8085 MPU but without the details of the MPUs. However, the general concepts discussed here are applicable to any microprocessor. The devices necessary to make up the 8085 MPUs will be discussed in the next chapter.

### 3.1.1 Microprocessor-Initiated Operations and 8085 Bus Organization

The MPU performs primarily four operations:*

1. Memory Read: Reads data (or instructions) from memory.
2. Memory Write: Writes data (or instructions) into memory.
3. I/O Read: Accepts data from input devices.
4. I/O Write: Sends data to output devices.

All these operations are part of the communication process between the MPU and peripheral devices (including memory). To communicate with a peripheral (or a memory location), the MPU needs to perform the following steps:

**Step 1:** Identify the peripheral or the memory location (with its address).
**Step 2:** Transfer binary information (data and instructions).
**Step 3:** Provide timing or synchronization signals.

The 8085 MPU performs these functions using three sets of communication lines called buses: the address bus, the data bus, and the control bus (Figure 3.1). In Chapter 1, these buses are shown as one group, called the system bus.

#### ADDRESS BUS

The address bus is a group of 16 lines generally identified as $A_0$ to $A_{15}$. The address bus is **unidirectional:** bits flow in one direction—from the MPU to peripheral devices. The MPU uses the address bus to perform the first function: identifying a peripheral or a memory location (Step 1).

In a computer system, each peripheral or memory location is identified by a binary number, called an **address,** and the address bus is used to carry a 16-bit address. This is sim-

*Other operations are omitted here for clarity and discussed in the next chapter.

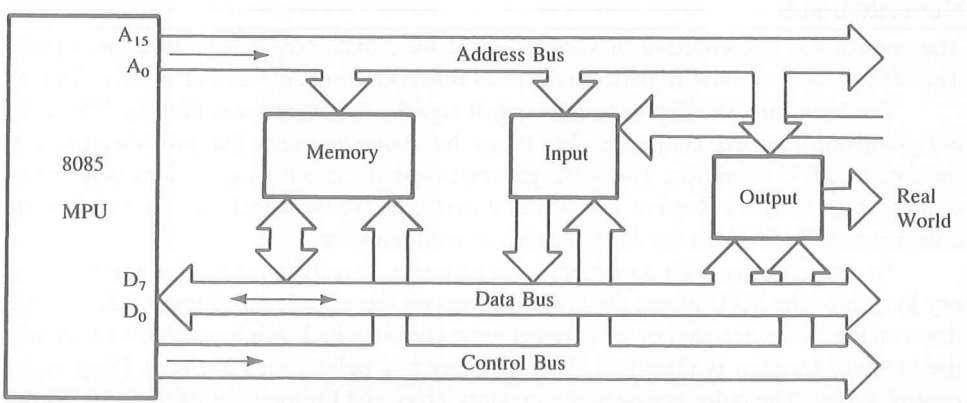



FIGURE 3.1
The 8085 Bus Structure

ilar to the postal address of a house. A house can be identified by various number schemes. For example, the forty-fifth house in a lane can be identified by the two-digit number 45 or by the four-digit number 0045. The two-digit numbering scheme can identify only a hundred houses, from 00 to 99. On the other hand, the four-digit scheme can identify ten thousand houses, from 0000 to 9999. Similarly, the number of address lines of the MPU determines its capacity to identify different memory locations (or peripherals). The 8085 MPU with its 16 address lines is capable of addressing $2^{16} = 65{,}536$ (generally known as 64K) memory locations. As explained in Chapter 1, 1K memory is determined by rounding off 1024 to the nearest thousand; similarly, 65,536 is rounded off to 64,000 as a multiple of 1K.

Most 8-bit microprocessors have 16 address lines. This may explain why microcomputer systems based on 8-bit microprocessors have 64K memory. However, not every microcomputer system has 64K memory. In fact, most single-board microcomputers have less than 4K of memory, even if the MPU is capable of addressing 64K memory. The number of address lines is arbitrary; it is determined by the designer of a microprocessor based on such considerations as availability of pins and intended applications of the processor. For example, the Intel 8088 processor has 20 and the Pentium processor has 32 address lines.

### DATA BUS

The data bus is a group of eight lines used for data flow (Figure 3.1).* These lines are **bidirectional**—data flow in both directions between the MPU and memory and peripheral devices. The MPU uses the data bus to perform the second function: transferring binary information (Step 2).

The eight data lines enable the MPU to manipulate 8-bit data ranging from 00 to FF ($2^8 = 256$ numbers). The largest number that can appear on the data bus is 11111111 ($255_{10}$). The 8085 is known as an 8-bit microprocessor. Microprocessors such as the Intel 8086, Zilog Z8000, and Motorola 68000 have 16 data lines; thus they are known as 16-bit microprocessors. The Intel 80386/486 have 32 data lines; thus they are classified as 32-bit microprocessors.

### CONTROL BUS

The control bus is comprised of various single lines that carry synchronization signals. The MPU uses such lines to perform the third function: providing timing signals (Step 3).

The term **bus,** in relation to the control signals, is somewhat confusing. These are not groups of lines like address or data buses, but individual lines that provide a pulse to indicate an MPU operation. The MPU generates specific control signals for every operation (such as Memory Read or I/O Write) it performs. These signals are used to identify a device type with which the MPU intends to communicate.

To communicate with a memory—for example, to read an instruction from a memory location—the MPU places the 16-bit address on the address bus (Figure 3.2). The address on the bus is decoded by an external logic circuit, which will be explained later, and the memory location is identified. The MPU sends a pulse called Memory Read as the control signal. The pulse activates the memory chip, and the contents of the memory location (8-bit data) are placed on the data bus and brought inside the microprocessor.

---

*The term *data* refers to any binary information that may include an instruction, an address, or a number.

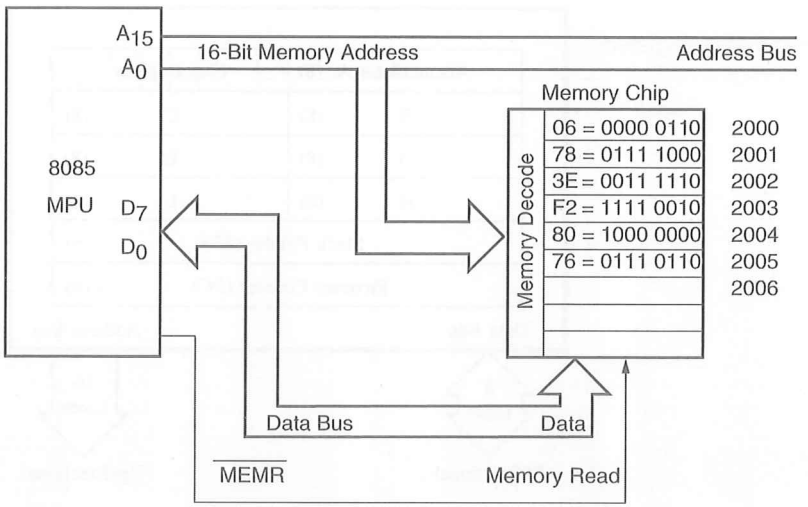



FIGURE 3.2
Memory Read Operation

What happens to the data byte brought into the MPU depends on the internal architecture of the microprocessor, which we will describe in the next section.

### 3.1.2 Internal Data Operations and the 8085 Registers

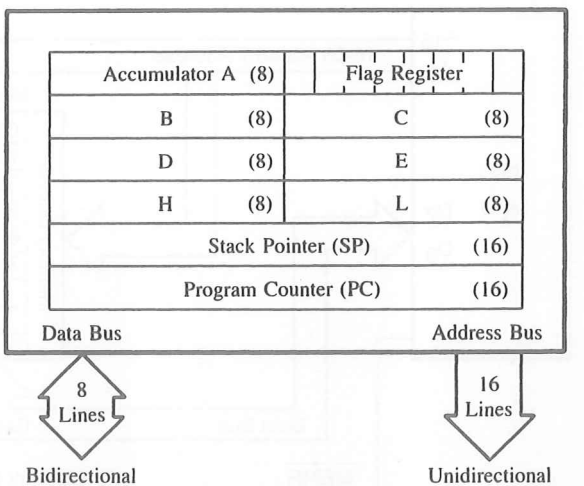
The internal architecture of the 8085 microprocessor determines how and what operations can be performed with the data. These operations are:

1. Store 8-bit data.
2. Perform arithmetic and logical operations.
3. Test for conditions.
4. Sequence the execution of instructions.
5. Store data temporarily during execution in the defined R/W memory locations called the stack.

To perform these operations, the microprocessor requires registers, an arithmetic/logic unit (ALU) and control logic, and internal buses (paths for information flow). Figure 3.3 (same as Figure 2.1(b); it is repeated here for reference) shows the programming model of the 8085 displaying the internal registers and the accumulator. The functions of these registers are described in reference to the five operations when the processor executes the following three instructions. The Hex codes of these instructions are stored in memory locations from 2000H to 2005H as shown in Figure 3.2.

```
2000    06      MVI B, 76H
2001    78
2002    3E      MVI A, F2H
2003    F2
2004    80      ADD B
2005    76      HLT
```

FIGURE 3.3
The 8085 Programmable Registers




When the user enters the memory address 2000H and pushes the execute key of the trainer, the processor places the address 2000H in the program counter (PC).

1. The program counter is a 16-bit register that performs the fourth operation in the list: sequencing the execution of the instructions. When the processor begins execution, it places the address 2000H on the address bus and increments the address in the PC to 2001 for the next operation. It brings the code 06, interprets the code, places the address 2001H on the address bus, and then gets byte 78H and increments the address in PC to 2002H. The processor repeats the same process for the next instruction, MVI A, F2H.
2. When the processor executes the first two instructions, it uses register B to store 78H and A to store F2H in binary (Operation 1).
3. When the processor executes the instruction ADD B in the ALU (Operation 2), it adds 78H to F2H, resulting in the sum 16AH (78H + F2H = 16AH). It replaces F2H by 6AH in A and sets the Carry flag as described next.
4. In our example, the addition operation generates a carry because the sum is larger than the size of the accumulator (8 bits). To indicate the carry, the processor sets the flip-flop called Carry (CY flag) to 1 and places logic 1 in the flag register at the designated bit position for the carry.
5. The fifth operation deals with the concept of the stack. The stack pointer is a 16-bit register used as a memory pointer to identify the stack, part of the R/W memory defined and used by the processor for temporary storage of data during the execution. This is fully described in Chapter 9.

### 3.1.3 Peripheral or Externally Initiated Operations

External devices (or signals) can initiate the following operations, for which individual pins on the microprocessor chip are assigned: Reset, Interrupt, Ready, Hold.

- Reset: When the reset pin is activated by an external key (also called a reset key), all internal operations are suspended and the program counter is cleared (it holds 0000H). Now the program execution can again begin at the zero memory address.
- Interrupt: The microprocessor can be interrupted from the normal execution of instructions and asked to execute some other instructions called a **service routine** (for example, emergency procedures). The microprocessor resumes its operation after completing the service routine (see Chapter 12).
- Ready: The 8085 has a pin called READY. If the signal at this READY pin is low, the microprocessor enters into a Wait state. This signal is used primarily to synchronize slower peripherals with the microprocessor.
- Hold: When the HOLD pin is activated by an external signal, the microprocessor relinquishes control of buses and allows the external peripheral to use them. For example, the HOLD signal is used in Direct Memory Access (DMA) data transfer (see Chapter 15).

These operations are listed here to provide an overview of the capabilities of the 8085. They will be discussed in Part III.

## 3.2 MEMORY

Memory is an essential component of a microcomputer system; it stores binary instructions and data for the microprocessor. There are various types of memory, which can be classified in two groups: prime (or main) memory and storage memory. In the last chapter, we discussed briefly two examples of prime memory: Read/Write memory (R/WM) and Read-Only memory (ROM). Magnetic tapes or disks can be cited as examples of storage memory. First, we will focus on prime memory and then, briefly discuss storage memory when we examine various types of memory.

The R/W memory is made of registers, and each register has a group of flip-flops or field-effect transistors that store bits of information; these flip-flops are called memory cells. The number of bits stored in a register is called a **memory** word; memory devices (chips) are available in various word sizes. The user can use this memory to hold programs and store data. On the other hand, the ROM stores information permanently in the form of diodes; the group of diodes can be viewed as a register. In a memory chip, all registers are arranged in a sequence and identified by binary numbers called memory addresses. To communicate with memory, the MPU should be able to

- select the chip,
- identify the register, and
- read from or write into the register.

The MPU uses its address bus to send the address of a memory register and uses the data bus and control lines to read from (as shown in Figure 3.2) or write into that register. In the following sections, we will examine the basic concepts related to memory: its

structure, its addressing, and its requirements to communicate with the MPU and build a model for R/W memory. However, except for slight differences in Read/Write control signals, the discussion is equally applicable to ROM.

### 3.2.1 Flip-Flop or Latch as a Storage Element

What is memory? It is a circuit that can store bits—high or low, generally voltage levels or capacitive charges representing 1 and 0. A flip-flop or a latch* is a basic element of memory. To write or store a bit in the latch, we need an input data bit ($D_{IN}$) and an enable signal (EN), as shown in Figure 3.4(a). In this latch, the stored bit is always available on the output line $D_{OUT}$. To avoid unintentional change in the input and control the availability of the output, we can use two tri-state* buffers on the latch, as shown in Figure 3.4(b). Now we can write into the latch by enabling the input buffer and read from it by enabling the output buffer. Figure 3.4(b) shows the Write signal as $\overline{WR}$ and the Read signal as $\overline{RD}$; these are active low signals indicated by the bar. This latch, which can store one binary bit, is called a memory cell. Figure 3.5(a) shows four such cells or latches grouped together; this is a register, which has four input lines and four output lines and can store four bits; thus the size of the memory word is four bits. The size of this register is specified either as 4-bit or 1 × 4-bit, which indicates one register with four cells or four I/O lines. Figures 3.5(b) and (c) show simplified block diagrams of the 4-bit register.

In Figure 3.6, four registers with eight cells (or an 8-bit memory word) are arranged in a sequence. To write into or read from any one of the registers, a specific register should be identified or enabled. This is a simple decoding function; a 2-to-4 decoder can perform that function. However, two more input lines $A_1$ and $A_0$, called address lines, are required to the decoder. These two input lines can have four different bit combinations (00, 01, 10, 11), and each combination can identify or enable one of the registers named as Register 0 through Register 3. Thus the Enable signal of the flip-flops in Figure 3.5 is replaced by two address lines in Figure 3.6. Figure 3.6(a) has 8-bit registers and Figure

*If you are not familiar with these devices, review Section 3.5; flip-flops (latches), tri-state buffers, and decoders are discussed briefly in Section 3.5.

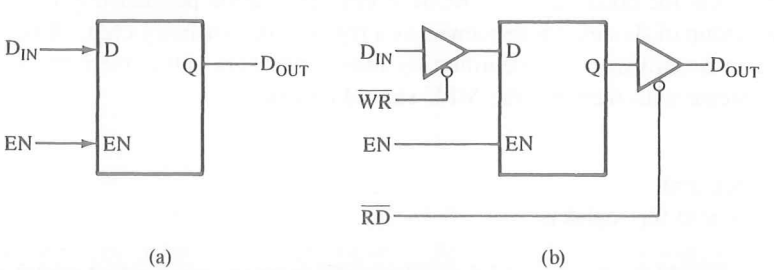



FIGURE 3.4
Latches as Storage Element: Basic Latch (a) and Latch with Two Tri-State Buffers (b)

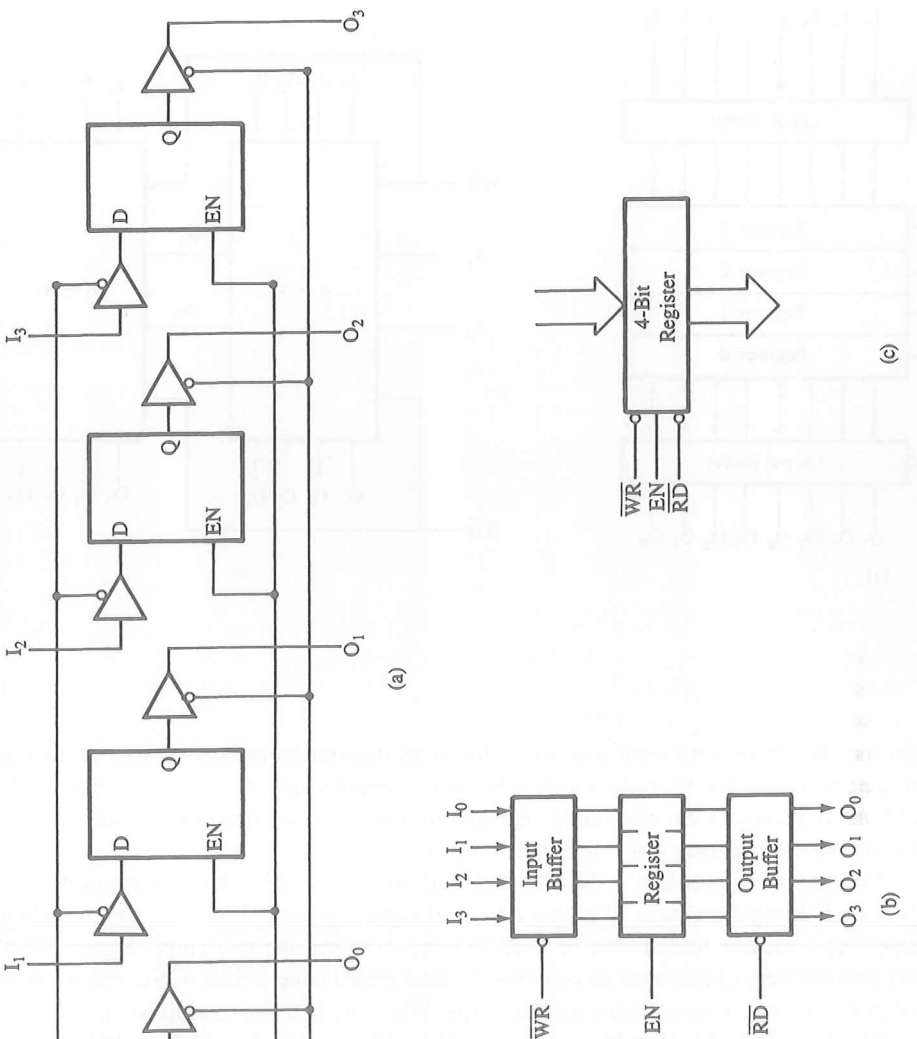



FIGURE 3.5
Four Latches as a 4-Bit Register (a) and Block Diagrams of a 4-Bit Register (b and c)

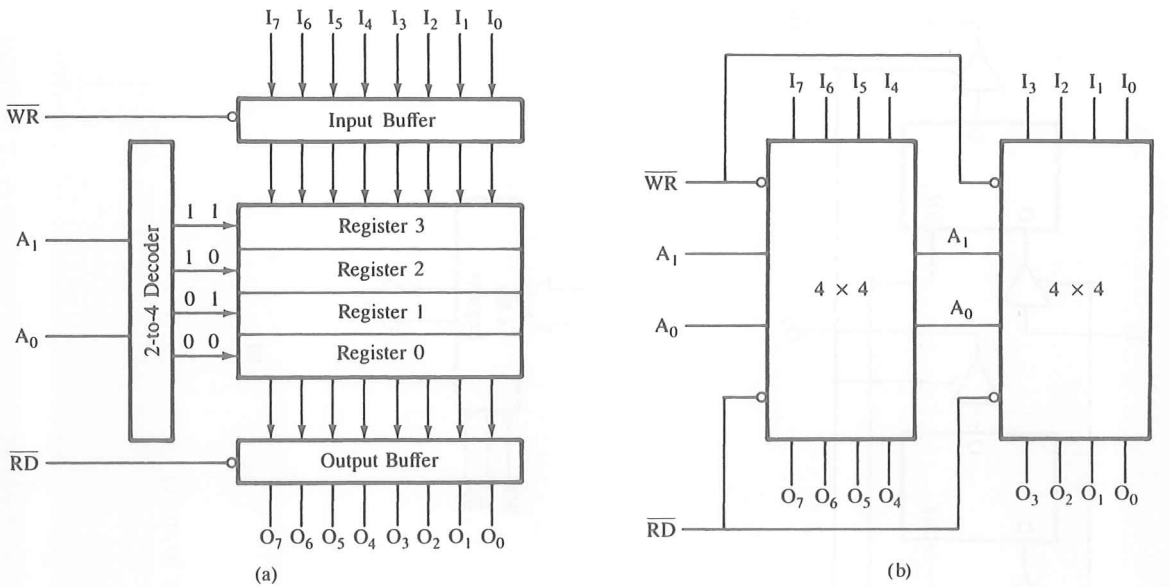



FIGURE 3.6
4 × 8-Bit Register

3.6(b) has two chips with 4-bit registers. This is an illustration of how smaller word size chips can be connected to make up an 8-bit word memory size. Now we can expand the number of registers. If we have eight registers on one chip, we need three address lines, and if we have 16 registers, we need four address lines.

An interesting problem is how to deal with more than one chip; for example, two chips with four registers each. We have a total of eight registers; therefore, we need three address lines, but one line should be used to select between the two chips. Figure 3.7(b) shows two memory chips, with an additional signal called Chip Select ($\overline{CS}$), and $A_2$ (with an inverter) is used to select between the chips. When $A_2$ is 0 (low), chip $M_1$ is selected, and when $A_2$ is 1 (high), chip $M_2$ is selected. The addresses on $A_1$ and $A_0$ will determine the registers to be selected; thus, by combining the logic on $A_2$, $A_1$, and $A_0$, the memory addresses range from 000 to 111. The concept of the Chip Select signal gives us more flexibility in designing chips and allows us to expand memory size by using multiple chips.

Now let us examine the problem from a different perspective. Assume that we have available four address lines and two memory chips with four registers each as before. Four address lines are capable of identifying 16 ($2^4$) registers; however, we need only three address lines to identify eight registers. What should we do with the fourth line? One of the solutions is shown in Figure 3.8. Memory chip $M_1$ is selected when $A_3$ and $A_2$ are both 0; therefore, registers in this chip are identified with the addresses ranging from 0000 to 0011 (0 to 3). Similarly, the addresses of memory chip $M_2$ range from 1000 to

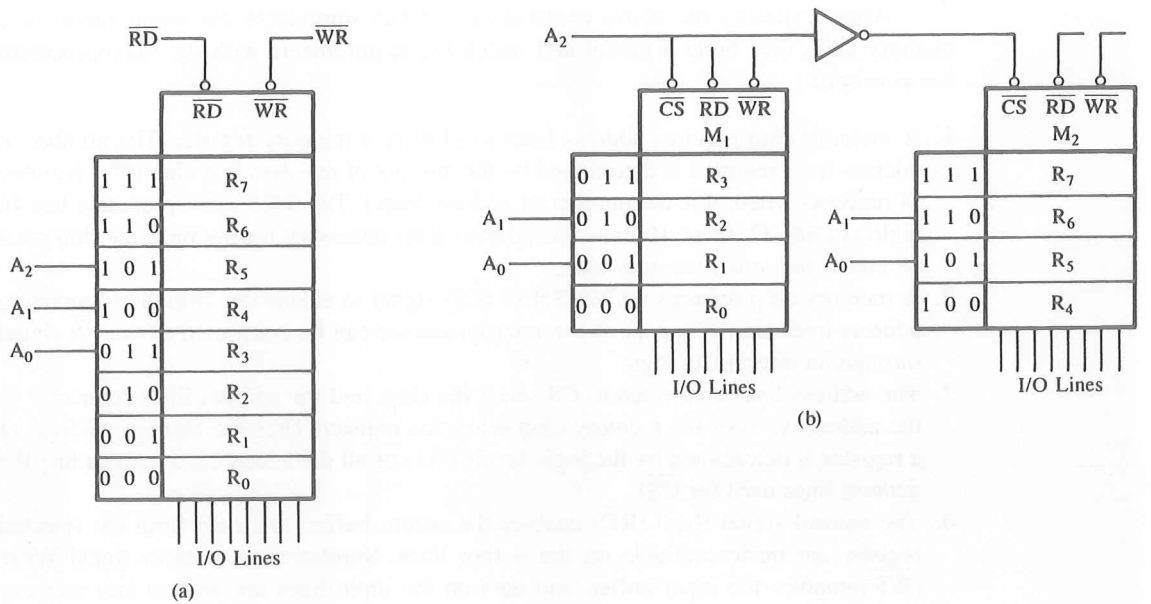



FIGURE 3.7
Two Memory Chips with Four Registers Each and Chip Select

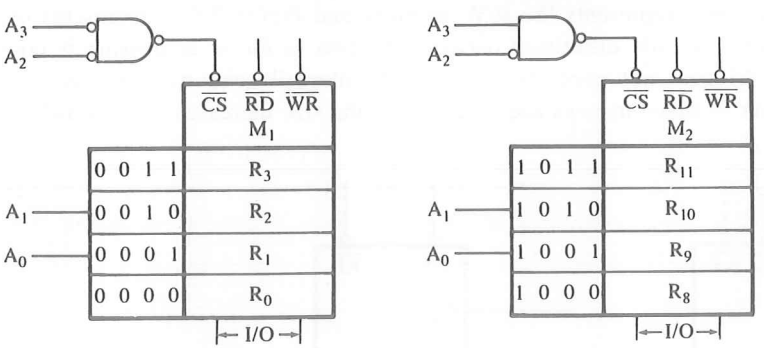



FIGURE 3.8
Addressing Eight Registers with Four Address Lines

1011 (8 to B); this chip is selected only when $A_3$ is 1 and $A_2$ is 0. In this example, we need three lines to identify eight registers: two for registers and one for Chip Select. However, we used the fourth line for Chip Select also. This is called complete or absolute decoding. Another option is to leave the fourth line as don't care; we will further explore this concept later.

After reviewing the above explanation, we can summarize the requirements of a memory chip, then build a model and match the requirements with the microprocessor bus concepts:

1. A memory chip requires address lines to identify a memory register. The number of address lines required is determined by the number of registers in a chip ($2^n$ = Number of registers where $n$ is the number of address lines). The 8085 microprocessor has 16 address lines. Of these 16 lines, the address lines necessary for the memory chip must be connected to the memory chip.
2. A memory chip requires a Chip Select ($\overline{CS}$) signal to enable the chip. The remaining address lines (from Step 1) of the microprocessor can be connected to the $\overline{CS}$ signal through an interfacing logic.
3. The address lines connected to $\overline{CS}$ select the chip, and the address lines connected to the address lines of the memory chip select the register. Thus the memory address of a register is determined by the logic levels (0/1) of all the address lines (including the address lines used for $\overline{CS}$).
4. The control signal Read ($\overline{RD}$) enables the output buffer, and data from the selected register are made available on the output lines. Similarly, the control signal Write ($\overline{WR}$) enables the input buffer, and data on the input lines are written into memory cells. The microprocessor can use its Memory Read and Memory Write control signals to enable the buffers and the data bus to transport the contents of the selected register between the microprocessor and memory.

A model of a typical memory chip representing the above requirements is shown in Figure 3.9. Figure 3.9(a) represents the R/W memory and Figure 3.9(b) represents the Read-Only memory; the only difference between the two as far as addressing is concerned is that ROM does not need the $\overline{WR}$ signal. Internally, the memory cells are arranged in a matrix format—in rows and columns; as the size increases, the internal de-

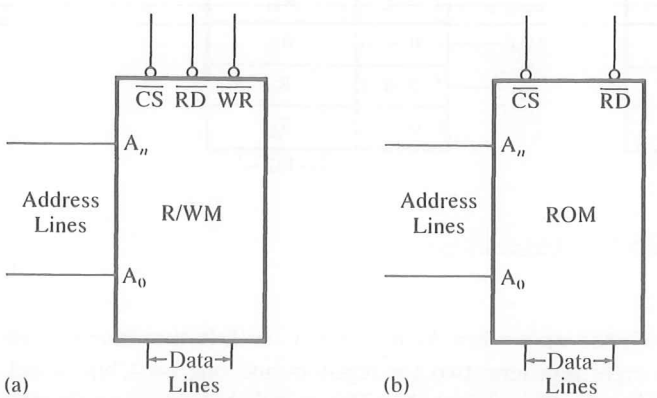



FIGURE 3.9
R/W Memory Model (a) and ROM Model (b)

coding scheme we discussed becomes impractical. For example, a memory chip with 1024 registers would require a 10-to-1024 decoder. If the cells are arranged in six rows and four columns, the internal decoding circuitry can be designed with two decoders, one for selecting a row and the other for selecting a column. However, we will not be concerned about the internal row and column arrangement because it does not affect our external interfacing logic, which is explained in the next section.

### 3.2.2 Memory Map and Addresses

Typically, in an 8-bit microprocessor system, 16 address lines are available for memory. This means it is a numbering system of 16 binary bits and is capable of identifying $2^{16}$ (65,536) memory registers, each register with a 16-bit address. The entire memory addresses can range from 0000 to FFFF in Hex. A memory map is a pictorial representation in which memory devices are located in the entire range of addresses. Memory addresses provide the locations of various memory devices in the system, and the interfacing logic defines the range of memory addresses for each memory device. The concept of memory map and memory addresses can be illustrated with an analogy of identical houses built in sequence and their postal addresses, or numbers.

Let us assume that houses are given three-digit decimal numbers, which will enable us to number one thousand houses from 000 to 999. Because it is cumbersome to direct someone to houses with large numbers, the numbering scheme can be devised with the concept of a row or block. Each block will have a hundred houses to be numbered with the last two digits from 00 to 99. Similarly, the blocks are also identified by the first decimal digit. For example, a house with the number 247 is house number 47 in block 2. With this scheme, all the houses in block 0 will be identified from 000 to 099, in block 2 from 200 to 299, and in block 9 from 900 to 999. This numbering scheme with three decimal digits is capable of giving addresses to one thousand houses from 000 to 999 (10 blocks of 100 houses each). Let us also assume that all houses are identical and have eight rooms.

The example of numbering the houses is directly applicable to assigning addresses to memory registers. In the binary number system, 16 binary digits can have 65,536 ($2^{16}$) different combinations. In the hexadecimal number system, 16 binary bits are equivalent to four Hex digits that can be used to assign addresses to 65,536 (0000H to FFFFH) memory registers in various memory chips. In our analogy, a memory chip is similar to a block in a housing development and a register can be viewed as a house with eight identical rooms.

Let us assume that we have a memory chip with 256 registers. Therefore, we need only 256 numbers (out of 65,536) that require eight address lines ($2^8 = 256$). Now the question is what we should do with the remaining eight address lines of the microprocessor. We can find a clue in our housing analogy. Let us assume that we have only 100 houses in block five. They will be numbered as 500 to 599; the first digit 5 remains constant and the next two digits vary from 00 to 99. Similarly, we can use the remaining eight address lines to assign fixed logic to generate a constant (fixed) number. This can be accomplished by using the remaining eight lines for the Chip Select through appropriate logic gates, as shown in Example 3.1.

As mentioned previously, in computer systems, we define 1024 as 1K; therefore a 1K-byte memory chip has 1024 registers with 8 bits each. Similarly, a group of 256 registers is defined as one *page* and each register is viewed as a *line* to write on. This is analogous to a notebook containing various pages, with each page having a certain number of lines. With this analogy, we can view 1K-byte memory as a chip with four pages (1024/256 = 4) with each page having 256 registers. With two Hex digits, 256 registers can be numbered from 00 to FFH; 1024 registers can be numbered with four digits from 0000 to 03FF. If we examine the high-order digits of 1K-byte memory, we find that they range from 00 to 03 representing four pages (00, 01, 02, and 03). In 8-bit microprocessor systems, this page concept is used frequently. In 16- and 32-bit microprocessor systems, the page concept (256 registers) defined here is not applicable; it is defined differently, based on the microprocessor used in a system.

So far we have been using the term *addresses* or *address range* for a given memory chip. The term *memory map* is used generally for the entire address ranges of the memory chips in a given system. The relationship of a row of houses to the road map is similar to the relationship of memory addresses to the memory map. However, these terms are also used synonymously.

---

**Example 3.1**

Illustrate the memory address range of the chip with 256 bytes of memory, shown in Figure 3.10(a), and explain how the range can be changed by modifying the hardware of the Chip Select $\overline{CS}$ line in Figure 3.10(b).

**Solution**

Figure 3.10(a) shows a memory chip with 256 registers with eight I/O lines; the memory size of the chip is expressed as 256 × 8. It has eight address lines ($A_7$–$A_0$), one Chip Select signal ($\overline{CS}$) (active low), and two control signals Read ($\overline{RD}$) and Write ($\overline{WR}$). The eight address lines ($A_7$–$A_0$) of the microprocessor are required to identify 256 memory registers. The remaining eight lines ($A_{15}$–$A_8$) are connected to the Chip Select ($\overline{CS}$) line through inverters and the NAND gate. The memory chip is enabled or selected when $\overline{CS}$ goes low. Therefore, to select the chip, the address lines $A_{15}$–$A_8$ should be at logic 0, which will cause the output of the NAND gate to go low. No other logic levels on the lines $A_{15}$–$A_8$ can select the chip. Once the chip is selected (enabled), the remaining address lines $A_7$–$A_0$ can assume any combination from 00H to FFH and identify any of the 256 memory registers through the decoder. Therefore, the memory addresses of the chip in Figure 3.10(a) will range from 0000H to 00FFH, as shown below.

| $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0000H |
| Chip Enable or Chip Select | | | | | | | | Register Select | | | | | | | | |
| | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 00FFH |

The address lines $A_{15}$–$A_8$, which are used to select the chip, must have fixed logic levels, and these lines are called high-order address lines. The address lines $A_7$–$A_0$, which

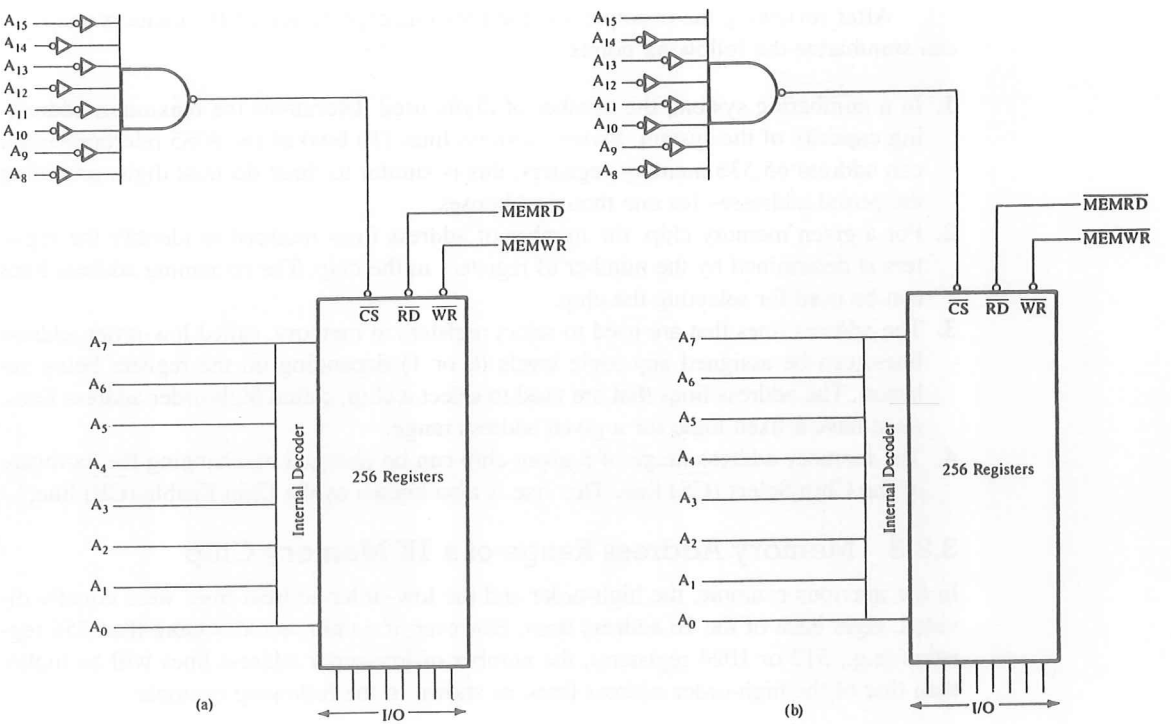



FIGURE 3.10
Memory Maps: 256 Bytes of Memory

are used to select a register, are called low-order address lines, and they can be assigned logic levels from all 0s to all 1s and any in-between combination. For example, when the address lines $A_7$–$A_0$ are all 0s, the register number 0 is selected, and when they are all 1s, the register number 255 (FFH) is selected. The Chip Select addresses are determined by the hardware (the inverters and NAND gate); therefore, the memory addresses of the chip can be changed by modifying the hardware. For example, if the inverter on line $A_{15}$ is removed, as shown in Figure 3.10(b), the address required on $A_{15}$–$A_8$ to enable the chip will be as follows:

| $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 80H |

The memory address range in Figure 3.10(b) will be 8000H to 80FFH.

The memory chips in Figures 3.10(a) and (b) are the same chips. However, by changing the hardware of the Chip Select logic, the location of the memory in the map can be changed, and memory can be assigned addresses in various locations over the entire map of 0000 to FFFFH.

After reviewing the example and the previous explanation of the memory map, we can summarize the following points.

1. In a numbering system, the number of digits used determines the maximum addressing capacity of the system. Sixteen address lines (16 bits) of the 8085 microprocessor can address 65,536 memory registers; this is similar to three decimal digits providing the postal addresses for one thousand houses.
2. For a given memory chip, the number of address lines required to identify the registers is determined by the number of registers in the chip. The remaining address lines can be used for selecting the chip.
3. The address lines that are used to select registers in memory, called low-order address lines, can be assigned any logic levels (0 or 1) depending on the register being selected. The address lines that are used to select a chip, called high-order address lines, must have a fixed logic for a given address range.
4. The memory address range of a given chip can be changed by changing the hardware of the Chip Select ($\overline{CS}$) line. This line is also known as the Chip Enable ($\overline{CE}$) line.

### 3.2.3 Memory Address Range of a 1K Memory Chip

In the previous example, the high-order and the low-order address lines were equally divided, eight each of the 16 address lines. However, if a chip includes more than 256 registers (e.g., 512 or 1024 registers), the number of low-order address lines will be higher than that of the high-order address lines, as shown in the following example.

---

**Example 3.2** Explain the memory address range of 1K (1024 × 8) memory shown in Figure 3.11 and explain the changes in the addresses if the hardware of the $\overline{CS}$ line is modified.

**Solution** The memory chip has 1024 registers; therefore 10 address lines ($A_9$–$A_0$) are required to identify the registers. The remaining six address lines ($A_{15}$–$A_{10}$) of the microprocessor are used for the Chip Select ($\overline{CS}$) signal. In Figure 3.11, the memory chip is enabled when the address lines $A_{15}$–$A_{10}$ are at logic 0. The address lines $A_9$–$A_0$ can assume any address of the 1024 registers, starting from all 0s to all 1s, as shown next.

| $A_{15}$ | $A_{14}$ | $A_{13}$ | $A_{12}$ | $A_{11}$ | $A_{10}$ | $A_9$ | $A_8$ | $A_7$ | $A_6$ | $A_5$ | $A_4$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0000H |
| ↓ | | | | | | ↓ | | | | | | | | | ↓ | |
| Chip Select Logic | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | = 03FFH |

The memory addresses range from 0000H to 03FFH.

By combining the high-order and the low-order address lines, we can specify the complete memory address range of a given chip. As explained in the previous example, the memory addresses of the 1K chip in Figure 3.11 can be changed to any other lo-

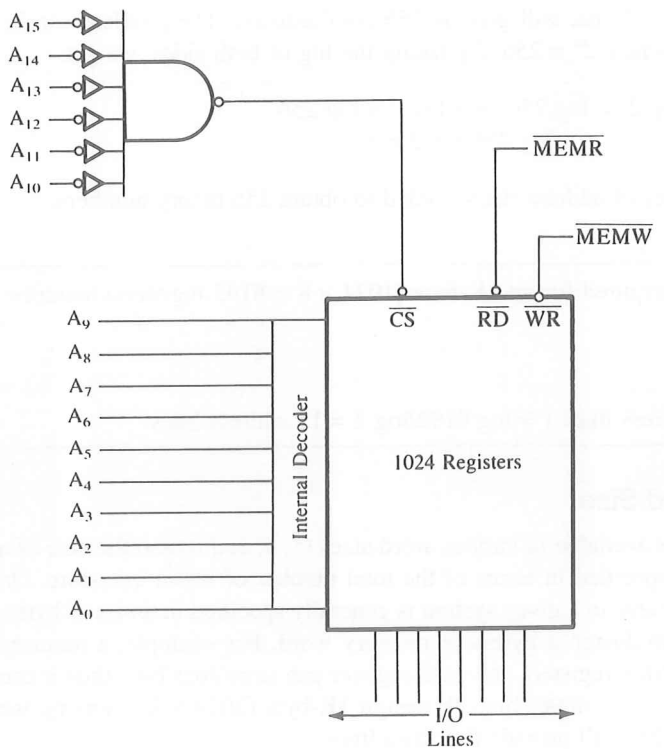



FIGURE 3.11
Memory Address Range: 1024 Bytes of Memory

cation by changing the hardware of the $\overline{CS}$ line. For example, if $A_{15}$ is connected to the NAND gate without an inverter, the memory addresses will range from 8000H to 83FFH.

The preceding discussion concerning the memory addresses is equally applicable to the Read-Only memory (ROM). The ROM is in many ways organized the same as the R/W memory. The primary difference between the organization of the two memories is in the control signals; the ROM requires only the Read signal from the MPU.

### 3.2.4 Memory Address Lines

In the last two examples, the address lines of the memory chips were given. The number of address lines necessary for a given chip can be obtained from data sheets. However, we need to know the relationship between the number of registers in a memory chip and the number of address lines. For a chip with 256 registers, we need 256 binary numbers to identify each register. Each address line can assume only two logic states (0 and 1); therefore,

we need to find the power of 2 that will give us 256 combinations. The problem can be restated as follows: Find $x$ where $2^x = 256$. By taking the log of both sides, we get:

$$\log 2^x = \log 256 \rightarrow x \log 2 = \log 256$$
$$x = \log 256/\log 2 = 8$$

Here $x$ represents the number of address lines needed to obtain 256 binary numbers.

**Example 3.3** Calculate the address lines required for an 8K-byte (1024 × 8 = 8192 registers) memory chip.

**Solution** Number of address lines $x$ = log 8192/log 2 = 13 address lines

### 3.2.5 Memory Word Size

Memory devices (chips) are available in various word sizes (1, 4, and 8) and the size of a memory chip is generally specified in terms of the total number of bits it can store. On the other hand, the memory size in a given system is generally specified in terms of bytes. Therefore, it is necessary to design a byte-size memory word. For example, a memory chip of size 1024 × 4 has 1024 registers and each register can store four bits; thus it can store a total of 4096 (1024 × 4 = 4096) bits. To design 1K-byte (1024 × 8) memory, we will need two chips; each chip will provide four data lines.

**Example 3.4** Calculate the number of memory chips needed to design 8K-byte memory if the memory chip size is 1024 × 1.

**Solution** The chip 1024 × 1 has 1024 (1K) registers and each register can store one bit with one data line. We need eight data lines for byte-size memory. Therefore, eight chips are necessary for 1K-byte memory. For 8K-byte memory, we will need 64 chips. We can arrive at the same answer by dividing 8K-byte by 1K × 1 as follows:

$$8192 \times 8 \div 1024 \times 1 = 64$$

So far we have been concerned primarily with finding necessary address lines and assigning addresses. In the next section, we will examine how the microprocessor communicates with memory using the address lines and control signals.

### 3.2.6 Memory and Instruction Fetch

The primary function of memory is to store instructions and data and to provide that information to the MPU whenever the MPU requests it. The MPU requests the information

by sending the address of a specific memory register on the address bus and enables the data flow by sending the control signal, as illustrated in the next example.

---

Example 3.5

The instruction code 0100 1111 (4FH) is stored in memory location 2005H. Illustrate the data flow and list the sequence of events when the instruction code is fetched by the MPU.

Solution

To fetch the instruction located in memory location 2005H, the following steps are performed:

1. The program counter places the 16-bit address 2005H of the memory location on the address bus (Figure 3.12).
2. The control unit sends the Memory Read control signal ($\overline{\text{MEMR}}$, active low) to enable the output buffer of the memory chip.
3. The instruction (4FH) stored in the memory location is placed on the data bus and transferred (copied) to the instruction decoder of the microprocessor.
4. The instruction is decoded and executed according to the binary pattern of the instruction.

Figure 3.12 shows how the 8085 MPU fetches the instruction using the address, the data, and the control buses. Figure 3.12 is similar to Figure 3.2, Memory Read operation, except that Figure 3.12 shows additional details.

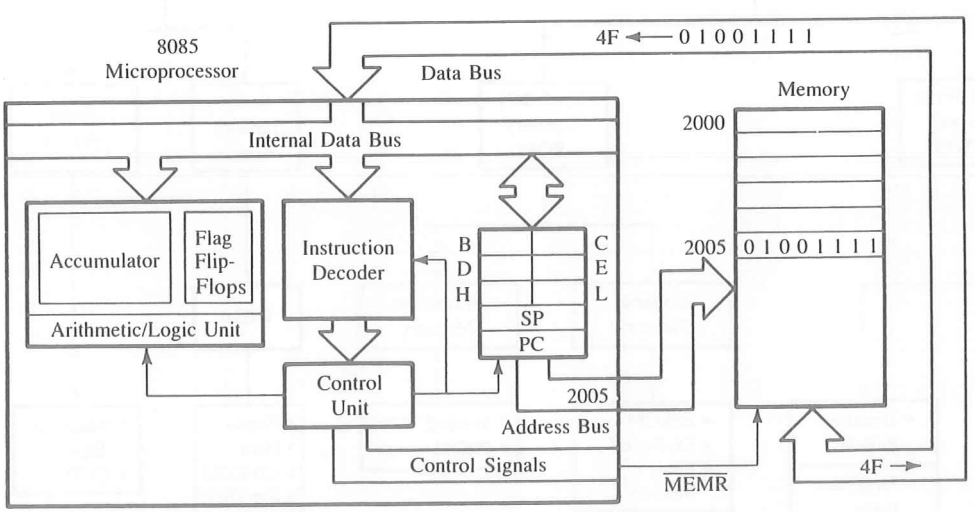



FIGURE 3.12
Instruction Fetch Operation

### 3.2.7 Memory Classification

As mentioned earlier, memory can be classified into two groups: prime (system or main) memory and storage memory. The R/WM and ROM are examples of prime memory; this is the memory the microprocessor uses in executing and storing programs. This memory should be able to respond fast enough to keep up with the execution speed of the microprocessor. Therefore, it should be random access memory, meaning that the microprocessor should be able to access information from any register with the same speed (independent of its place in the chip). The size of a memory chip is specified in terms of bits. For example, a 1K memory chip means it can store 1K (1024) bits (not bytes). On the other hand, memory in a system such as a PC is specified in bytes. For example, 4M memory in a PC means it has 4 megabytes of memory.

The other group is the storage memory, such as magnetic disks and tapes (see Figure 3.13). This memory is used to store programs and results after the completion of

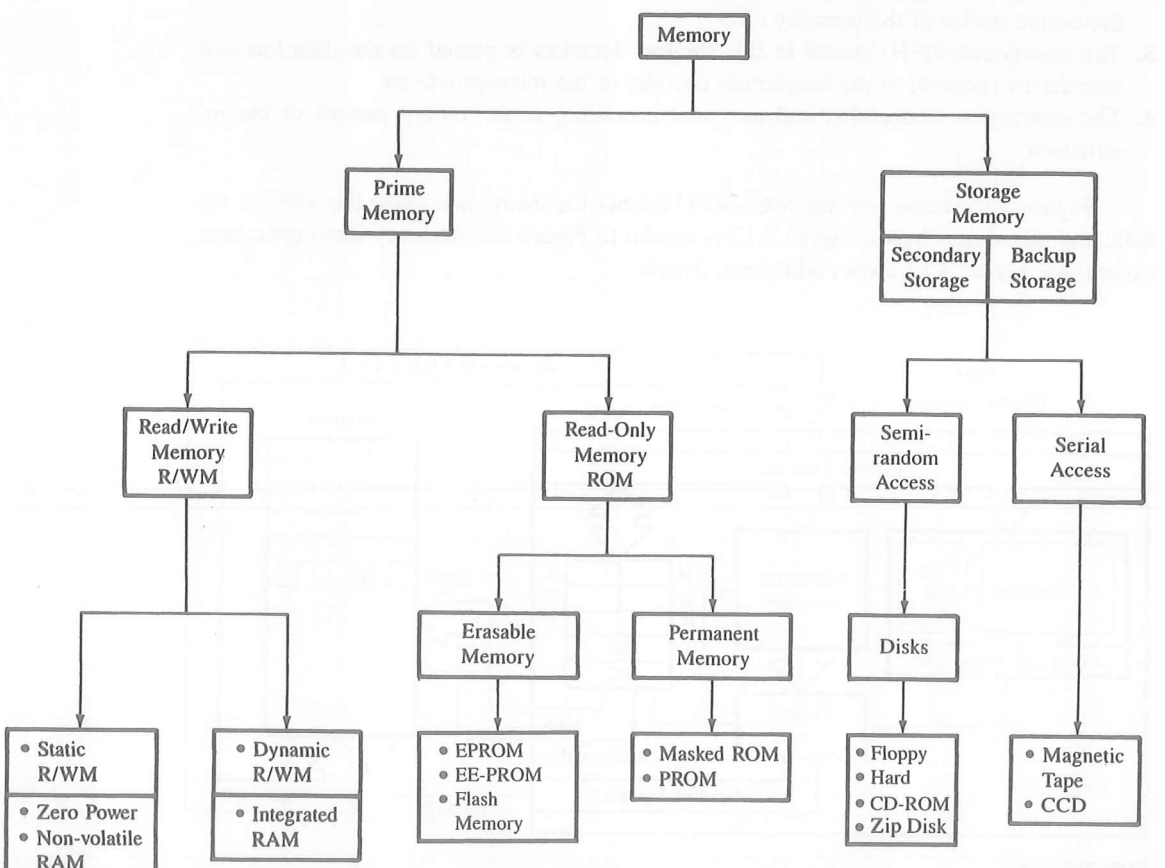



FIGURE 3.13
Memory Classification

program execution. Information stored in these memories is nonvolatile, meaning information remains intact even if the system is turned off. The microprocessor cannot directly execute or process programs stored in these devices; programs need to be copied into the R/W prime memory first. Therefore, the size of the prime memory, such as 512K or 8M (megabytes), determines how large a program the system can process. The size of the storage memory is unlimited; when one disk or tape is full, the next one can be used.

Figure 3.13 shows two groups in storage memory: secondary storage and backup storage. The secondary storage is similar to what you put on a shelf in your study, and the backup is similar to what you store in your attic. The secondary storage and the backup storage include devices such as disks, magnetic tapes, magnetic bubble memory, and charged-coupled devices, as shown in Figure 3.13. The primary features of these devices are high capacity, low cost, and slow access. A disk is similar to a record; the access to the stored information in the disk is semirandom (see Chapter 11 for additional discussion). The remaining devices shown in Figure 3.13 are serial, meaning if information is stored in the middle of the tape, it can be accessed after running half the tape. We will discuss some of these memory storage devices again in Chapter 11. In this chapter, we will focus on various types of prime memory.

Figure 3.13 shows that the prime (system) memory is divided into two main groups: Read/Write memory (R/WM) and Read-Only memory (ROM); each group includes several different types of memory, as discussed below.

### R/WM (READ/WRITE MEMORY)

As the name suggests, the microprocessor can write into or read from this memory; it is popularly known as Random Access memory (RAM). It is used primarily for information that is likely to be altered, such as writing programs or receiving data. This memory is volatile, meaning that when the power is turned off, all the contents are destroyed. Two types of R/W memories—static and dynamic—are available; they are described in the following paragraphs.

**Static Memory (SRAM)** This memory is made up of flip-flops, and it stores the bit as a voltage. Each memory cell requires six transistors; therefore, the memory chip has low density but high speed. This memory is more expensive and consumes more power than the dynamic memory described in the next paragraph. In high-speed processors (such as Intel 486 and Pentium), SRAM known as cache memory is included on the processor chip. In addition, high-speed cache memory is also included external to the processor to improve the performance of a system.

**Dynamic Memory (DRAM)** This memory is made up of MOS transistor gates, and it stores the bit as a charge. The advantages of dynamic memory are that it has high density and low power consumption and is cheaper than static memory. The disadvantage is that the charge (bit information) leaks; therefore, stored information needs to be read and written again every few milliseconds. This is called refreshing the memory, and it requires extra circuitry, adding to the cost of the system. It is generally economical to use dynamic memory when the system memory size is at least 8K; for small systems, the static mem-

ory is appropriate. However, in recent years, the processor speed has reached beyond 200 MHz, and 1000 MHz processors are in the design stage. In comparison to the processor speed, the DRAM is too slow. To increase the speed of DRAM various techniques are being used. These techniques have resulted in high-speed memory chips such as EDO (Extended Data Out), SDRAM (Synchronous DRAM), and RDRAM (Rambus DRAM).

### ROM (READ-ONLY MEMORY)

The ROM is a nonvolatile memory; it retains stored information even if the power is turned off. This memory is used for programs and data that need not be altered. As the name suggests, the information can be read only, which means once a bit pattern is stored, it is permanent or at least semipermanent. The permanent group includes two types of memory: masked ROM and PROM. The semipermanent group also includes two types of memory: EPROM and EE-PROM, as shown in Figure 3.13. The concept underlying the ROM can be explained with the diodes arranged in a matrix format, as shown in Figure 3.14. The horizontal lines are connected to vertical lines only through the diodes; they are not connected where they appear to cross in the diagram. Each of the eight horizontal rows can be viewed as a register with binary addresses ranging from 000 to 111; information is stored by the diodes in the register as 0s or 1s. The presence of a diode stores 1, and its absence stores 0. When a register is selected, the voltage of that line goes high, and the output lines, where diodes are connected, go high. For example, when the memory register 111 is selected, the data byte 0111 1000 (78H) can be read at the data lines $D_7$–$D_0$.

The diode representation is a simplified version of the actual MOSFET memory cell. The manufacturer of the ROM designs the MOSFET matrix according to the infor-

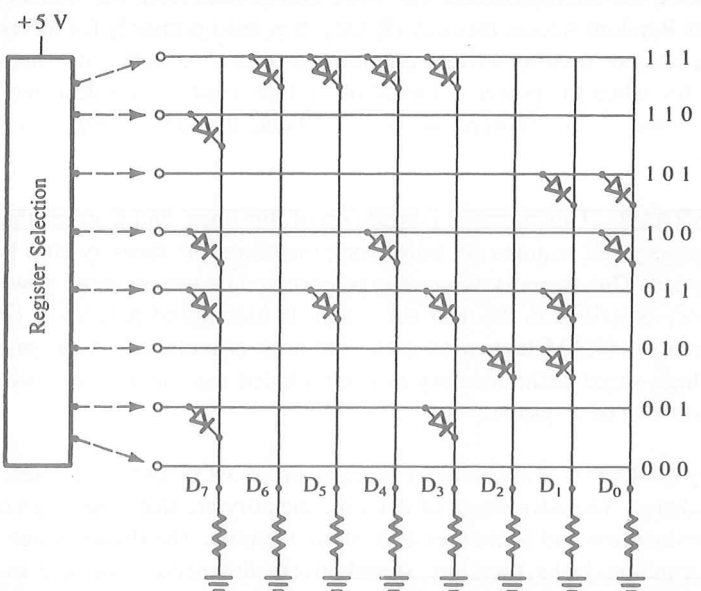



FIGURE 3.14
Functional Representation of ROM Memory Cell

mation to be stored; therefore, information is permanently recorded in the ROM, as a song is recorded on a record. Five types of ROM—masked ROM, PROM, EPROM, EE-PROM, and Flash Memory—are described in the following paragraphs.

**Masked ROM** In this ROM, a bit pattern is permanently recorded by the masking and metalization process. Memory manufacturers are generally equipped to do this process. It is an expensive and specialized process, but economical for large production quantities.

**PROM (Programmable Read-Only Memory)** This memory has nichrome or polysilicon wires arranged in a matrix; these wires can be functionally viewed as diodes or fuses. This memory can be programmed by the user with a special PROM programmer that selectively burns the fuses according to the bit pattern to be stored. The process is known as "burning the PROM," and the information stored is permanent.

**EPROM (Erasable Programmable Read-Only Memory)** This memory stores a bit by charging the floating gate of an FET. Information is stored by using an EPROM programmer, which applies high voltages to charge the gate. All the information can be erased by exposing the chip to ultraviolet light through its quartz window, and the chip can be reprogrammed. Because the chip can be reused many times, this memory is ideally suited for product development, experimental projects, and college laboratories. The disadvantages of EPROM are (1) it must be taken out of the circuit to erase it, (2) the entire chip must be erased, and (3) the erasing process takes 15 to 20 minutes.

**EE-PROM (Electrically Erasable PROM)** This memory is functionally similar to EPROM, except that information can be altered by using electrical signals at the register level rather than erasing all the information. This has an advantage in field and remote control applications. In microprocessor systems, software update is a common occurrence. If EE-PROMs are used in the systems, they can be updated from a central computer by using a remote link via telephone lines. Similarly, in a process control where timing information needs to be changed, it can be changed by sending electrical signals from a central place. This memory also includes a Chip Erase mode, whereby the entire chip can be erased in 10 ms vs. 15 to 20 min. to erase an EPROM. However, this memory is expensive compared to EPROM or flash memory (described in the next paragraph).

**Flash Memory** This is a variation of EE-PROM that is becoming popular. The major difference between the flash memory and EE-PROM is in the erasure procedure: The EE-PROM can be erased at a register level, but the flash memory must be erased either in its entirety or at the sector (block) level. These memory chips can be erased and programmed at least a million times. The power supply requirement for programming these chips was around 12 V, but now chips are available that can be programmed using a power supply as low as 1.8 V. Therefore, this memory is ideally suited for low-power systems.

In a microprocessor-based product, programs are generally written in ROM, and data that are likely to vary are stored in R/WM. For example, in a microprocessor-controlled oven, programs that run the oven are permanently stored in ROM, and data such as baking period, starting time, and temperature are entered in R/W memory through

the keyboard. On the other hand, when microcomputers are used for developing software or for learning purposes, programs are first written in R/W memory, and then stored on a storage memory such as a cassette tape or floppy disk.

### ADVANCES IN MEMORY TECHNOLOGY

Memory technology has advanced considerably in recent years. In addition to static and dynamic R/W memory, other options are also available in memory devices. Examples include Zero Power RAM, Nonvolatile RAM, and Integrated RAM.

The Zero Power RAM is a CMOS Read/Write memory with battery backup built internally. It includes lithium cells and voltage-sensing circuitry. When the external power supply voltage falls below 3 V, the power-switching circuitry connects the lithium battery; thus, this memory provides the advantages of R/W and Read-Only memory.

The Nonvolatile RAM is a high-speed static R/W memory array backed up, bit for bit, by EE-PROM array for nonvolatile storage. When the power is about to go off, the contents of R/W memory are quickly stored in the EE-PROM by activating the Store signal on the memory chip, and the stored data can be read into the R/W memory segment when the power is again turned on. This memory chip combines the flexibility of static R/W memory with the nonvolatility of EE-PROM.

The Integrated RAM (iRAM) is a dynamic memory with the refreshed circuitry built on the chip. For the user, it is similar to the static R/W memory. The user can derive the advantages of the dynamic memory without having to build the external refresh circuitry.

## 3.3 INPUT AND OUTPUT (I/O) DEVICES

Input/output devices are the means through which the MPU communicates with "the outside world." The MPU accepts binary data as input from devices such as keyboards and A/D converters and sends data to output devices such as LEDs or printers. There are two different methods by which I/O devices can be identified: one uses an 8-bit address and the other uses a 16-bit address. These methods are described briefly in the following sections.

### 3.3.1 I/Os with 8-Bit Addresses (Peripheral-Mapped I/O)

In this type of I/O, the MPU uses eight address lines to identify an input or an output device; this is known as peripheral-mapped I/O (also known as I/O-mapped I/O). This is an 8-bit numbering system for I/Os used in conjunction with Input and Output instructions. This is also known as I/O space, separate from memory space, which is a 16-bit numbering system. The eight address lines can have 256 ($2^8$ combinations) addresses; thus, the MPU can identify 256 input devices and 256 output devices with addresses ranging from 00H to FFH. The input and output devices are differentiated by the control signals; the MPU uses the I/O Read control signal for input devices and the I/O Write control signal for output devices. The entire range of I/O addresses from 00 to FF is known as an I/O map, and individual addresses are referred to as I/O device addresses or I/O port numbers.

If we use LEDs as output or switches as input, we need to resolve two issues: how to assign addresses and how to connect these I/O devices to the data bus. In a bus architecture, these devices cannot be connected directly to the data bus or the address bus; all

connections must be made through tri-state interfacing devices so they will be enabled and connected to the buses only when the MPU chooses to communicate with them. In the case of memory, we did not have to be concerned with these problems because of the internal address decoding, Read/Write buffers, and availability of $\overline{CS}$ and control signals of the memory chip. In the case of I/O devices, we need to use external interfacing devices (see Section 3.5).

The steps in communicating with an I/O device are similar to those in communicating with memory and can be summarized as follows:

1. The MPU places an 8-bit address on the address bus, which is decoded by external decode logic (explained in Chapter 4).
2. The MPU sends a control signal (I/O Read or I/O Write) and enables the I/O device.
3. Data are transferred using the data bus.

### 3.3.2 I/Os with 16-Bit Addresses (Memory-Mapped I/O)

In this type of I/O, the MPU uses 16 address lines to identify an I/O device; an I/O is connected as if it is a memory register. This is known as memory-mapped I/O. The MPU uses the same control signal (Memory Read or Memory Write) and instructions as those of memory. In some microprocessors, such as the Motorola 6800, all I/Os have 16-bit addresses; I/Os and memory share the same memory map (64K). In memory-mapped I/O, the MPU follows the same steps as if it is accessing a memory register.

The peripheral- and memory-mapped I/O techniques will be discussed in detail in the context of interfacing I/O devices (see Chapter 5).

## 3.4 EXAMPLE OF A MICROCOMPUTER SYSTEM

On the basis of the discussion in the previous sections, we can expand the microcomputer system shown in Figure 3.1 to include additional details. Figure 3.15 illustrates such a system. It shows the 8085 MPU, two types of memory (EPROM and R/WM), input and output, and the buses linking all peripherals (memory and I/Os) to the MPU.

The address lines $A_{15}$–$A_0$ are used to address memory, and the low-order address bus $A_7$–$A_0$ is used to identify the input and the output. The data bus $D_7$–$D_0$ is bidirectional and common to all the devices. The four control signals generated by the MPU are connected to different peripheral devices, as shown in Figure 3.15.

The MPU communicates with only one peripheral at a time by enabling the peripheral through its control signal. For example, to send data to the output device, the MPU places the device address (output port number) on the address bus, data on the data bus, and enables the output device using the control signal $\overline{IOW}$ (I/O Write). The output device latches and displays data if the output device happens to be LEDs. The other peripherals that are not enabled remain in a high impedance state called tri-state (explained later), similar to being disconnected from the system. Figure 3.15 is a simplified block diagram of the system; it does not show such details as data latching and tri-state devices (see Section 3.5).

Figure 3.16 shows an expanded version of the output section and the buses of Figure 3.15. The block diagram includes tri-state bus drivers, a decoder, and a latch. The

# 4

# 8085 Microprocessor Architecture and Memory Interfacing

The 8085 microprocessor is a much improved version of its predecessor, the 8080A. The 8085 includes on its chip most of the logic circuitry for performing computing tasks and for communicating with peripherals. However, eight of its bus lines are **multiplexed;** that is, they are time-shared by the low-order address and data. This chapter discusses the 8085 architecture in detail and illustrates techniques for demultiplexing the bus and generating the necessary control signals.

Later, the chapter describes a typical 8085-based microcomputer designed with general-purpose memory and I/O devices; it also illustrates the bus timing signals in executing an instruction. Then, it examines the requirements of a memory chip based on the timing signals and derives the steps necessary in interfacing memory. In addition, the chapter includes illustrations of a special-purpose device, the 8155 and its interfacing.

OBJECTIVES

- ☐ Recognize the functions of various pins of the 8085 microprocessor.
- ☐ Explain the bus timings in fetching an instruction from memory.
- ☐ Explain how to demultiplex the $AD_7$–$AD_0$ bus using a latch.
- ☐ Draw a logic schematic to generate four control signals, using the 8085 IO/$\overline{M}$, $\overline{RD}$, and $\overline{WR}$ signals: (1) $\overline{MEMR}$, (2) $\overline{MEMW}$, (3) $\overline{IOR}$, and (4) $\overline{IOW}$. Explain the functions of these control signals.
- ☐ List the various internal units that make up the 8085 architecture, and explain their functions in decoding and executing an instruction.
- ☐ Draw the block diagram of an 8085-based microcomputer.

- ☐ List the steps performed by the 8085 microprocessor, and identify the contents of buses when an instruction is being executed.
- ☐ Analyze a memory interfacing circuit, and specify the memory addresses of a given memory device.
- ☐ Recognize partial decoding and identify foldback (mirror) memory space.

# 4.1 THE 8085 MPU

The term **microprocessing unit** (MPU) is similar to the term **central processing unit** (CPU) used in traditional computers. We define the MPU as a device or a group of devices (as a unit) that can communicate with peripherals, provide timing signals, direct data flow, and perform computing tasks as specified by the instructions in memory. The unit will have the necessary lines for the address bus, the data bus, and the control signals, and would require only a power supply and a crystal (or equivalent frequency-determining components) to be completely functional.

Using this description, the 8085 microprocessor can almost qualify as an MPU, but with the following two limitations.

1. The low-order address bus of the 8085 microprocessor is **multiplexed** (time-shared) with the data bus. The buses need to be demultiplexed.
2. Appropriate control signals need to be generated to interface memory and I/O with the 8085. (Intel has some specialized memory and I/O devices that do not require such control signals.)

This section shows how to demultiplex the bus and generate the control signals after describing the 8085 microprocessor and illustrates the bus timings.

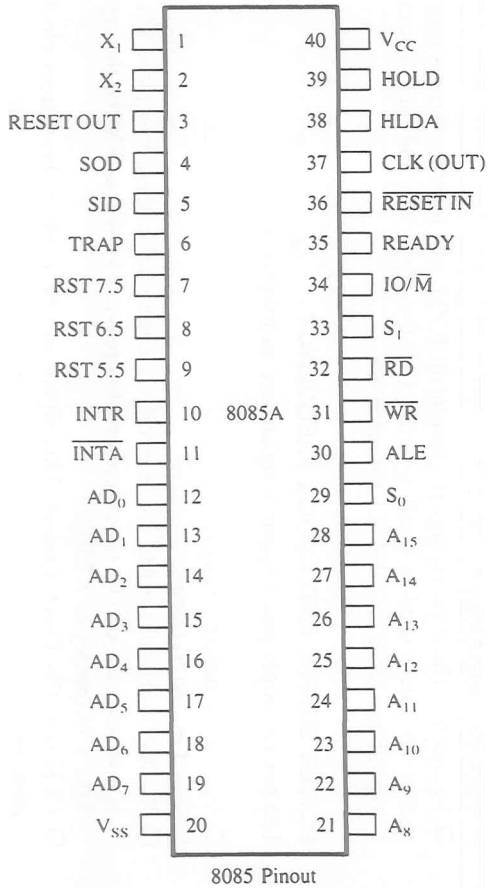
## 4.1.1 The 8085 Microprocessor

The 8085A (commonly known as the 8085) is an 8-bit general-purpose microprocessor capable of addressing 64K of memory. The device has forty pins, requires a +5 V single power supply, and can operate with a 3-MHz single-phase clock. The 8085A-2 version can operate at the maximum frequency of 5 MHz. The 8085 is an enhanced version of its predecessor, the 8080A; its instruction set is upward-compatible with that of the 8080A, meaning that the 8085 instruction set includes all the 8080A instructions plus some additional ones.

Figure 4.1 shows the logic pinout of the 8085 microprocessor. All the signals can be classified into six groups: (1) address bus, (2) data bus, (3) control and status signals, (4) power supply and frequency signals, (5) externally initiated signals, and (6) serial I/O ports.

### ADDRESS BUS

The 8085 has 16 signal lines (pins) that are used as the address bus; however, these lines are split into two segments: $A_{15}$–$A_8$ and $AD_7$–$AD_0$. The eight signal lines, $A_{15}$–$A_8$, are




8085 Pinout

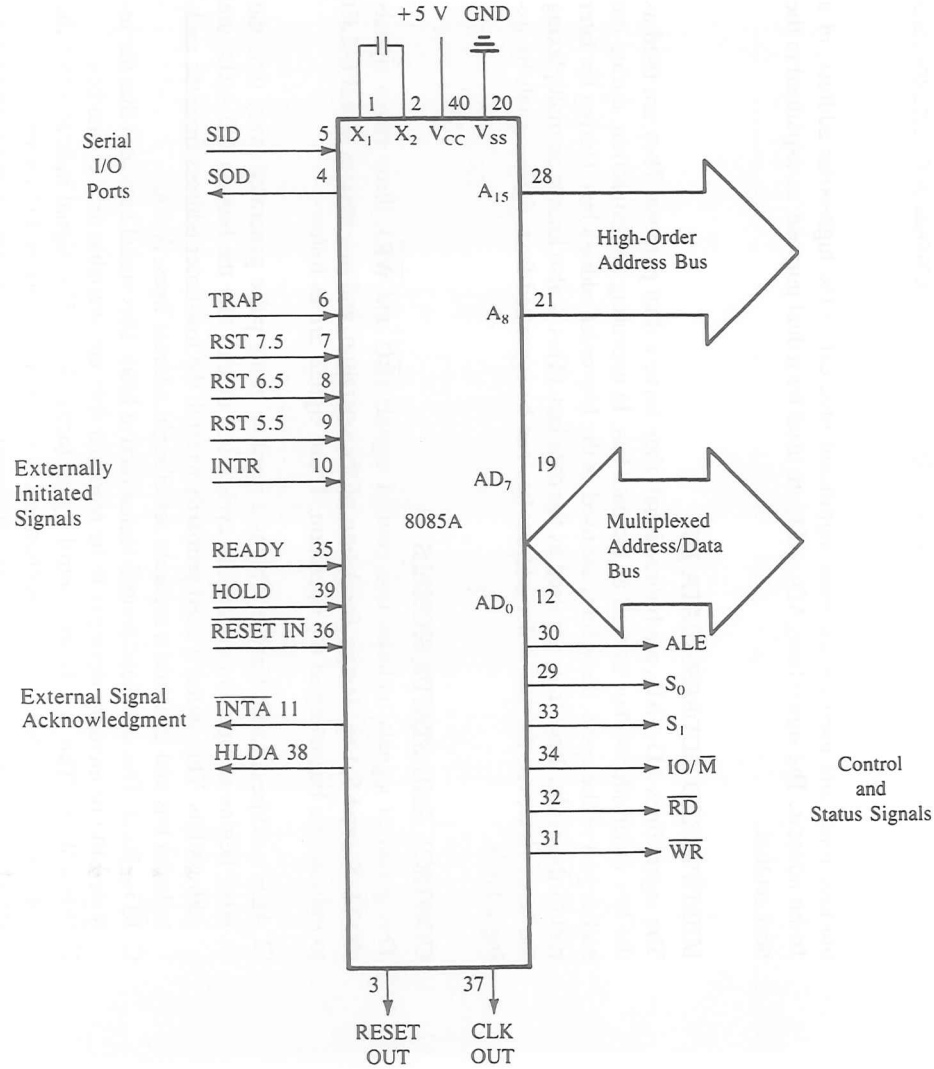



FIGURE 4.1
The 8085 Microprocessor Pinout and Signals

NOTE: The 8085A is commonly known as the 8085.
SOURCE (Pinout): Intel Corporation, *Embedded Microprocessors* (Santa Clara, Calif.: Author, 1994), pp. 1-11.

unidirectional and used for the most significant bits, called the high-order address, of a 16-bit address. The signal lines $AD_7$–$AD_0$ are used for a dual purpose, as explained in the next section.

### MULTIPLEXED ADDRESS/DATA BUS

The signal lines $AD_7$–$AD_0$ are bidirectional: they serve a dual purpose. They are used as the low-order address bus as well as the data bus. In executing an instruction, during the earlier part of the cycle, these lines are used as the low-order address bus. During the later part of the cycle, these lines are used as the data bus. (This is also known as multiplexing the bus.) However, the low-order address bus can be separated from these signals by using a latch.

### CONTROL AND STATUS SIGNALS

This group of signals includes two control signals ($\overline{RD}$ and $\overline{WR}$), three status signals ($IO/\overline{M}$, $S_1$, and $S_0$) to identify the nature of the operation, and one special signal (ALE) to indicate the beginning of the operation. These signals are as follows:

- ALE—Address Latch Enable: This is a positive going pulse generated every time the 8085 begins an operation (machine cycle); it indicates that the bits on $AD_7$–$AD_0$ are address bits. This signal is used primarily to latch the low-order address from the multiplexed bus and generate a separate set of eight address lines, $A_7$–$A_0$.
- $\overline{RD}$—Read: This is a Read control signal (active low). This signal indicates that the selected I/O or memory device is to be read and data are available on the data bus.
- $\overline{WR}$—Write: This is a Write control signal (active low). This signal indicates that the data on the data bus are to be written into a selected memory or I/O location.
- $IO/\overline{M}$: This is a status signal used to differentiate between I/O and memory operations. When it is high, it indicates an I/O operation; when it is low, it indicates a memory operation. This signal is combined with $\overline{RD}$ (Read) and $\overline{WR}$ (Write) to generate I/O and memory control signals.
- $S_1$ and $S_0$: These status signals, similar to $IO/\overline{M}$, can identify various operations, but they are rarely used in small systems. (All the operations and their associated status signals are listed in Table 4.1 for reference.)

### POWER SUPPLY AND CLOCK FREQUENCY

The power supply and frequency signals are as follows:

- $V_{CC}$: +5 V power supply.
- $V_{SS}$: Ground Reference.
- $X_1$, $X_2$: A crystal (or RC, LC network) is connected at these two pins. The frequency is internally divided by two; therefore, to operate a system at 3 MHz, the crystal should have a frequency of 6 MHz.
- CLK (OUT)—Clock Output: This signal can be used as the system clock for other devices.

TABLE 4.1
8085 Machine Cycle Status and Control Signals

| Machine Cycle | Status IO/$\overline{M}$ | Status $S_1$ | Status $S_0$ | Control Signals |
|---|---|---|---|---|
| Opcode Fetch | 0 | 1 | 1 | $\overline{RD} = 0$ |
| Memory Read | 0 | 1 | 0 | $\overline{RD} = 0$ |
| Memory Write | 0 | 0 | 1 | $\overline{WR} = 0$ |
| I/O Read | 1 | 1 | 0 | $\overline{RD} = 0$ |
| I/O Write | 1 | 0 | 1 | $\overline{WR} = 0$ |
| Interrupt Acknowledge | 1 | 1 | 1 | $\overline{INTA} = 0$ |
| Halt | Z | 0 | 0 | $\overline{RD}$, $\overline{WR}$ = Z and $\overline{INTA} = 1$ |
| Hold | Z | X | X | |
| Reset | Z | X | X | |

NOTE: Z = Tri-state (high impedance)
X = Unspecified

## EXTERNALLY INITIATED SIGNALS, INCLUDING INTERRUPTS

The 8085 has five interrupt signals (see Table 4.2) that can be used to interrupt a program execution. One of the signals, INTR (Interrupt Request), is identical to the 8080A microprocessor interrupt signal (INT); the others are enhancements to the 8080A. The microprocessor acknowledges an interrupt request by the $\overline{INTA}$ (Interrupt Acknowledge) signal. (The interrupt process is discussed in Chapter 12.)

In addition to the interrupts, three pins—RESET, HOLD, and READY—accept the externally initiated signals as inputs. To respond to the HOLD request, the 8085 has one

TABLE 4.2
8085 Interrupts and Externally Initiated Signals

| | |
|---|---|
| ☐ INTR (Input) | Interrupt Request: This is used as a general-purpose interrupt; it is similar to the INT signal of the 8080A. |
| ☐ $\overline{INTA}$ (Output) | Interrupt Acknowledge: This is used to acknowledge an interrupt. |
| ☐ RST 7.5 (Inputs)<br>RST 6.5<br>RST 5.5 | Restart Interrupts: These are vectored interrupts that transfer the program control to specific memory locations. They have higher priorities than the INTR interrupt. Among these three, the priority order is 7.5, 6.5, and 5.5. |
| ☐ TRAP (Input) | This is a nonmaskable interrupt and has the highest priority. |
| ☐ HOLD (Input) | This signal indicates that a peripheral such as a DMA (Direct Memory Access) controller is requesting the use of the address and data buses. |
| ☐ HLDA (Output) | Hold Acknowledge: This signal acknowledges the HOLD request. |
| ☐ READY (Input) | This signal is used to delay the microprocessor Read or Write cycles until a slow-responding peripheral is ready to send or accept data. When this signal goes low, the microprocessor waits for an integral number of clock cycles until it goes high. |

signal called HLDA (Hold Acknowledge). The functions of these signals were previously discussed in Section 3.1.3. The RESET is again described below, and others are listed in Table 4.2 for reference.

- $\overline{\text{RESET IN}}$: When the signal on this pin goes low, the program counter is set to zero, the buses are tri-stated, and the MPU is reset.
- RESET OUT: This signal indicates that the MPU is being reset. The signal can be used to reset other devices.

SERIAL I/O PORTS

The 8085 has two signals to implement the serial transmission: SID (Serial Input Data) and SOD (Serial Output Data). In serial transmission, data bits are sent over a single line, one bit at a time, such as the transmission over telephone lines. This will be discussed in Chapter 16 on serial I/O.

In this chapter, we will focus on the first three groups of signals; others will be discussed in later chapters.

### 4.1.2 Microprocessor Communication and Bus Timings

To understand the functions of various signals of the 8085, we should examine the process of communication (reading from and writing into memory) between the microprocessor and memory and the timings of these signals in relation to the system clock. The first step in the communication process is reading from memory or fetching an instruction. This can be easily understood using an analogy of how a package is picked up from your house by a shipping company such as Federal Express. The steps are as follows:

1. A courier gets the address from the office; he or she drives the pickup van, finds the street, and looks for your house number.
2. The courier rings the bell.
3. Somebody in the house opens the door and gives the package to the courier, and the courier returns to the office with the package.
4. The internal office staff disposes the package according to the instructions given by the customer.

Now let us examine the steps in the following example of how the microprocessor fetches or gets a machine code from memory.

---

**Example 4.1** Refer to Example 3.5 in the last chapter (Section 3.2.6): Illustrate the steps and the timing of data flow when the instruction code 0100 1111 (4FH—MOV C,A), stored in location 2005H, is being fetched.

**Solution** To fetch the byte (4FH), the MPU needs to identify the memory location 2005H and enable the data flow from memory. This is called the Fetch cycle. The data flow is shown in Figure 4.2, and the timings are explained below.

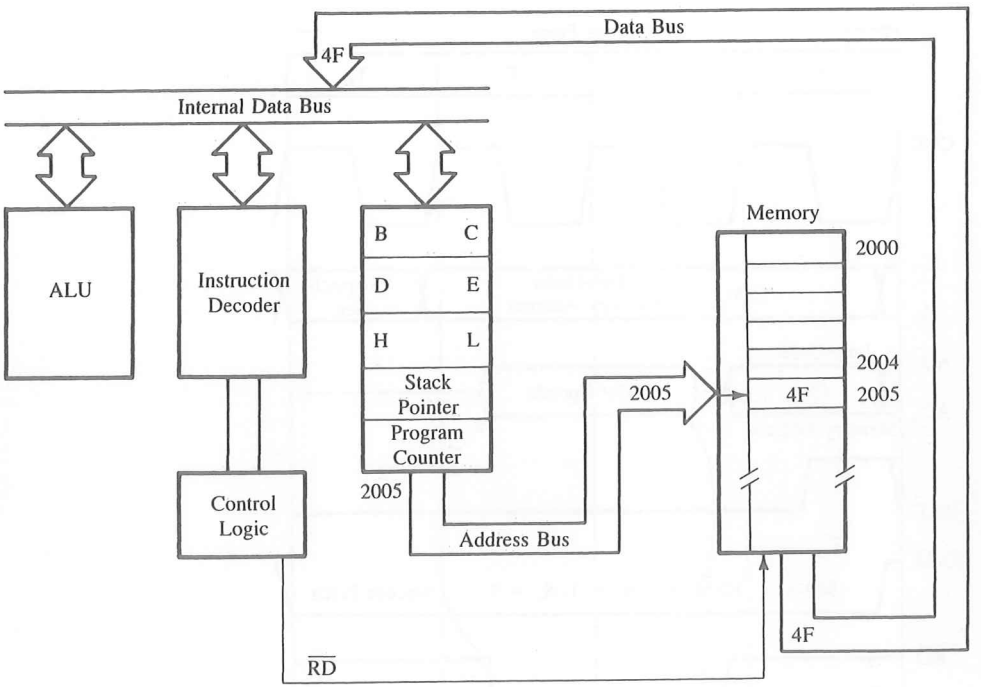



FIGURE 4.2
Data Flow from Memory to the MPU

Figure 4.3 shows the timing of how a data byte is transferred from memory to the MPU; it shows five different groups of signals in relation to the system clock. The address bus and data bus are shown as two parallel lines. This is a commonly used practice to represent logic levels of groups of lines; some lines are high and others are low. The crossover of the lines indicates that a new byte (information) is placed on the bus, and a dashed straight line indicates the high impedance state. To fetch the byte, the MPU performs the following steps:

**Step 1:** The microprocessor places the 16-bit memory address from the program counter (PC) on the address bus (Figure 4.2). In our analogy, this is the equivalent of our courier getting on the road to find the address.

Figure 4.3 shows that at $T_1$ the high-order memory address 20H is placed on the address lines $A_{15}$–$A_8$, the low-order memory address 05H is placed on the bus $AD_7$–$AD_0$, and the ALE signal goes high. Similarly, the status signal IO/$\overline{M}$ goes low, indicating that this is a memory-related operation. (For the sake of clarity, the other two status signals, $S_1$ and $S_0$, are not shown in Figure 4.3; they will be discussed in the next section.)

**Step 2:** The control unit sends the control signal $\overline{RD}$ to enable the memory chip (Figure 4.2). This is similar to ringing the doorbell in our analogy of a package pickup.

The control signal $\overline{RD}$ is sent out during the clock period $T_2$, thus enabling the memory chip (Figure 4.3). The $\overline{RD}$ signal is active during two clock periods.

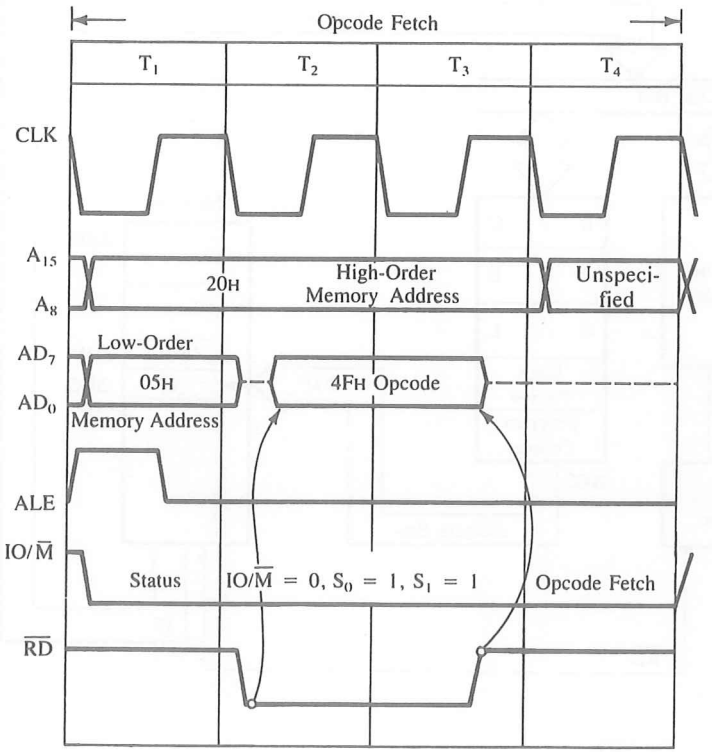



FIGURE 4.3
Timing: Transfer of Byte from Memory to MPU

**Step 3:** The byte from the memory location is placed on the data bus.

When the memory is enabled, the instruction byte (4FH) is placed on the bus $AD_7$–$AD_0$ and transferred to the microprocessor. The $\overline{RD}$ signal causes 4FH to be placed on bus $AD_7$–$AD_0$ (shown by the arrow), and when $\overline{RD}$ goes high, it causes the bus to go into high impedance.

**Step 4:** The byte is placed in the instruction decoder of the microprocessor, and the task is carried out according to the instruction.

The machine code or the byte (4FH) is decoded by the instruction decoder, and the contents of the accumulator are copied into register C. This task is performed during the period $T_4$ in Figure 4.3.

The above four steps are similar to the steps listed in our analogy of the package pickup.

### 4.1.3 Demultiplexing the Bus $AD_7$–$AD_0$

The need for demultiplexing the bus $AD_7$–$AD_0$ becomes easier to understand after examining Figure 4.3. This figure shows that the address on the high-order bus (20H) remains on the bus for three clock periods. However, the low-order address (05H) is lost after the

first clock period. This address needs to be latched and used for identifying the memory address. If the bus $AD_7$–$AD_0$ is used to identify the memory location (2005H), the address will change to 204FH after the first clock period.

Figure 4.4 shows a schematic that uses a latch and the ALE signal to demultiplex the bus. The bus $AD_7$–$AD_0$ is connected as the input to the latch 74LS373. The ALE signal is connected to the Enable (G) pin of the latch, and the Output control ($\overline{OC}$) signal of the latch is grounded.

Figure 4.3 shows that the ALE goes high during $T_1$. When the ALE is high, the latch is transparent; this means that the output changes according to input data. During $T_1$, the output of the latch is 05H. When the ALE goes low, the data byte 05H is latched until the next ALE, and the output of the latch represents the low-order address bus $A_7$–$A_0$ after the latching operation.

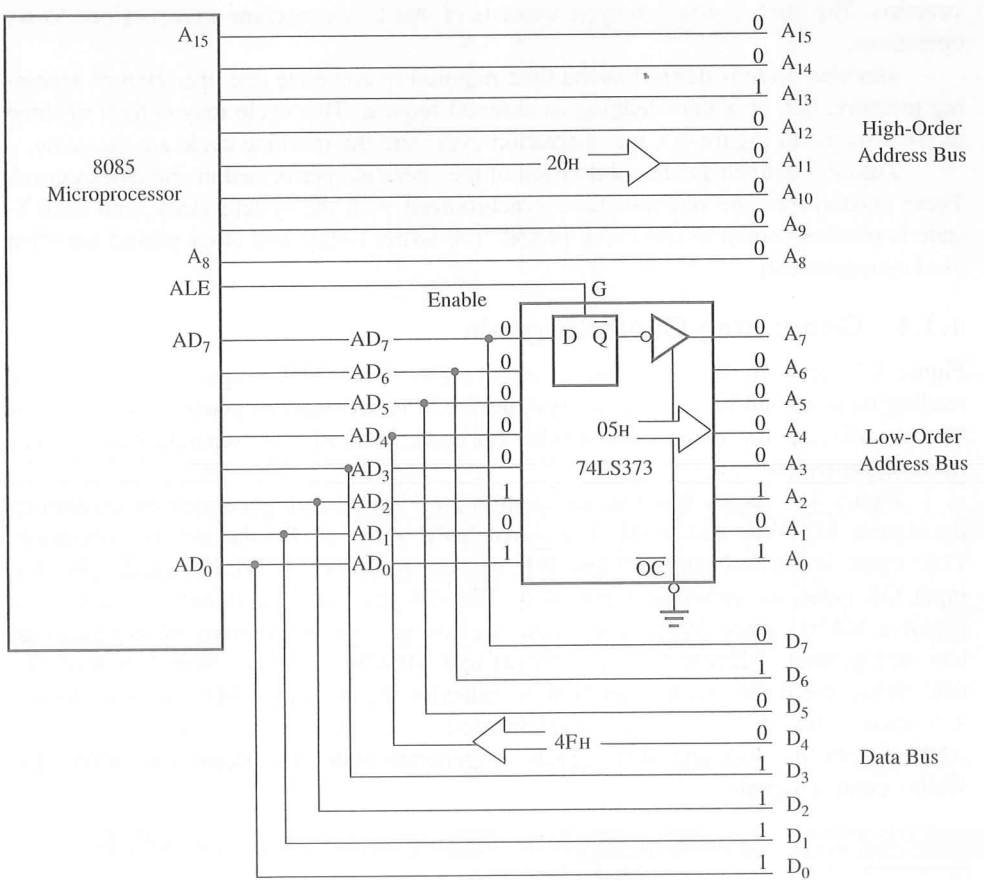



FIGURE 4.4
Schematic of Latching Low-Order Address Bus

Intel has circumvented the problem of demultiplexing the low-order bus by designing special devices such as the 8155 (256 bytes of R/W memory + I/Os), which is compatible with the 8085 multiplexed bus. These devices internally demultiplex the bus using the ALE signal (see Figures 4.18 and 4.19).

After carefully examining Figure 4.3, we can make the following observations:

1. The machine code 4FH (0100 1000) is a one-byte instruction that copies the contents of the accumulator into register C.
2. The 8085 microprocessor requires one external operation—fetching a machine code* from memory location 2005H.
3. The entire operation—fetching, decoding, and executing—requires four clock periods.

Now we can define three terms—instruction cycle, machine cycle, and T-state—and use these terms later for examining timings of various 8085 operations (Section 4.2).

*Instruction cycle* is defined as the time required to complete the execution of an instruction. The 8085 instruction cycle consists of one to six machine cycles or one to six operations.

*Machine cycle* is defined as the time required to complete one operation of accessing memory, I/O, or acknowledging an external request. This cycle may consist of three to six T-states. In Figure 4.3, the instruction cycle and the machine cycle are the same.

*T-state* is defined as one subdivision of the operation performed in one clock period. These subdivisions are internal states synchronized with the system clock, and each T-state is precisely equal to one clock period. The terms T-state and clock period are often used synonymously.

### 4.1.4 Generating Control Signals

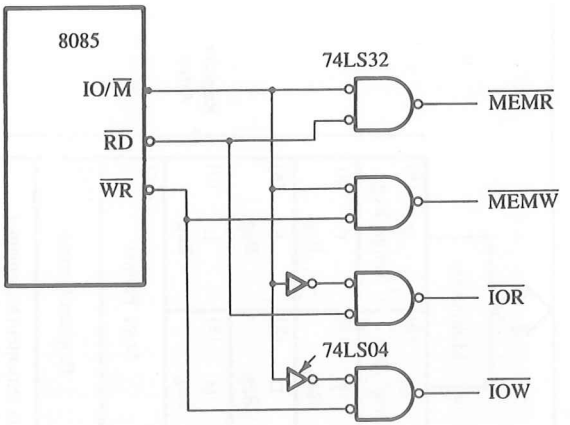
Figure 4.3 shows the $\overline{RD}$ (Read) as a control signal. Because this signal is used both for reading memory and for reading an input device, it is necessary to generate two different Read signals: one for memory and another for input. Similarly, two separate Write signals must be generated.

Figure 4.5 shows that four different control signals are generated by combining the signals $\overline{RD}$, $\overline{WR}$, and IO/$\overline{M}$. The signal IO/$\overline{M}$ goes low for the memory operation. This signal is ANDed with $\overline{RD}$ and $\overline{WR}$ signals by using the 74LS32 quadruple two-input OR gates, as shown in Figure 4.5. The OR gates are functionally connected as negative NAND gates. When both input signals go low, the outputs of the gates go low and generate $\overline{MEMR}$ (Memory Read) and $\overline{MEMW}$ (Memory Write) control signals. When the IO/$\overline{M}$ signal goes high, it indicates the peripheral I/O operation. Figure 4.5 shows that this signal is complemented using the Hex inverter 74LS04 and ANDed with the $\overline{RD}$ and $\overline{WR}$ signals to generate $\overline{IOR}$ (I/O Read) and $\overline{IOW}$ (I/O Write) control signals.

---

*This code is an operation code (opcode) that instructs the microprocessor to perform the specified task. The term *opcode* was explained in Chapter 2 (Section 2.3).

FIGURE 4.5
Schematic to Generate Read/Write Control Signals for Memory and I/O



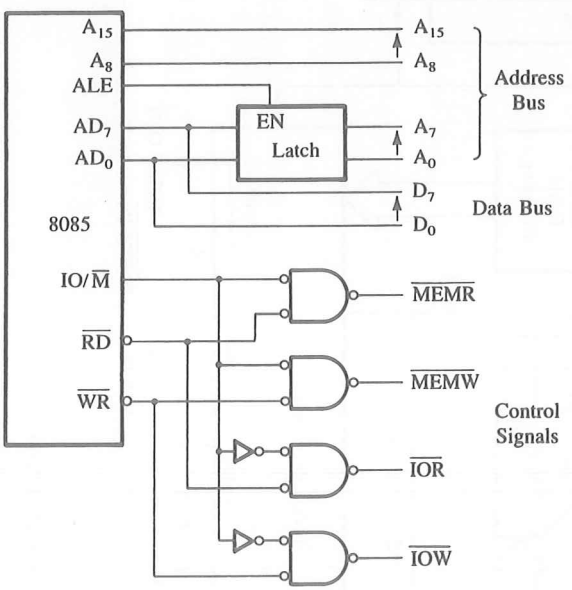




FIGURE 4.6
8085 Demultiplexed Address and Data Bus with Control Signals

To demultiplex the bus and to generate the necessary control signals, the 8085 microprocessor requires a latch and logic gates to build the MPU, as shown in Figure 4.6. This MPU can be interfaced with any memory or I/O.

### 4.1.5 A Detailed Look at the 8085 MPU and Its Architecture

Figure 4.7 shows the internal architecture of the 8085 beyond the programmable registers we discussed previously. It includes the ALU (Arithmetic/Logic Unit), Timing and

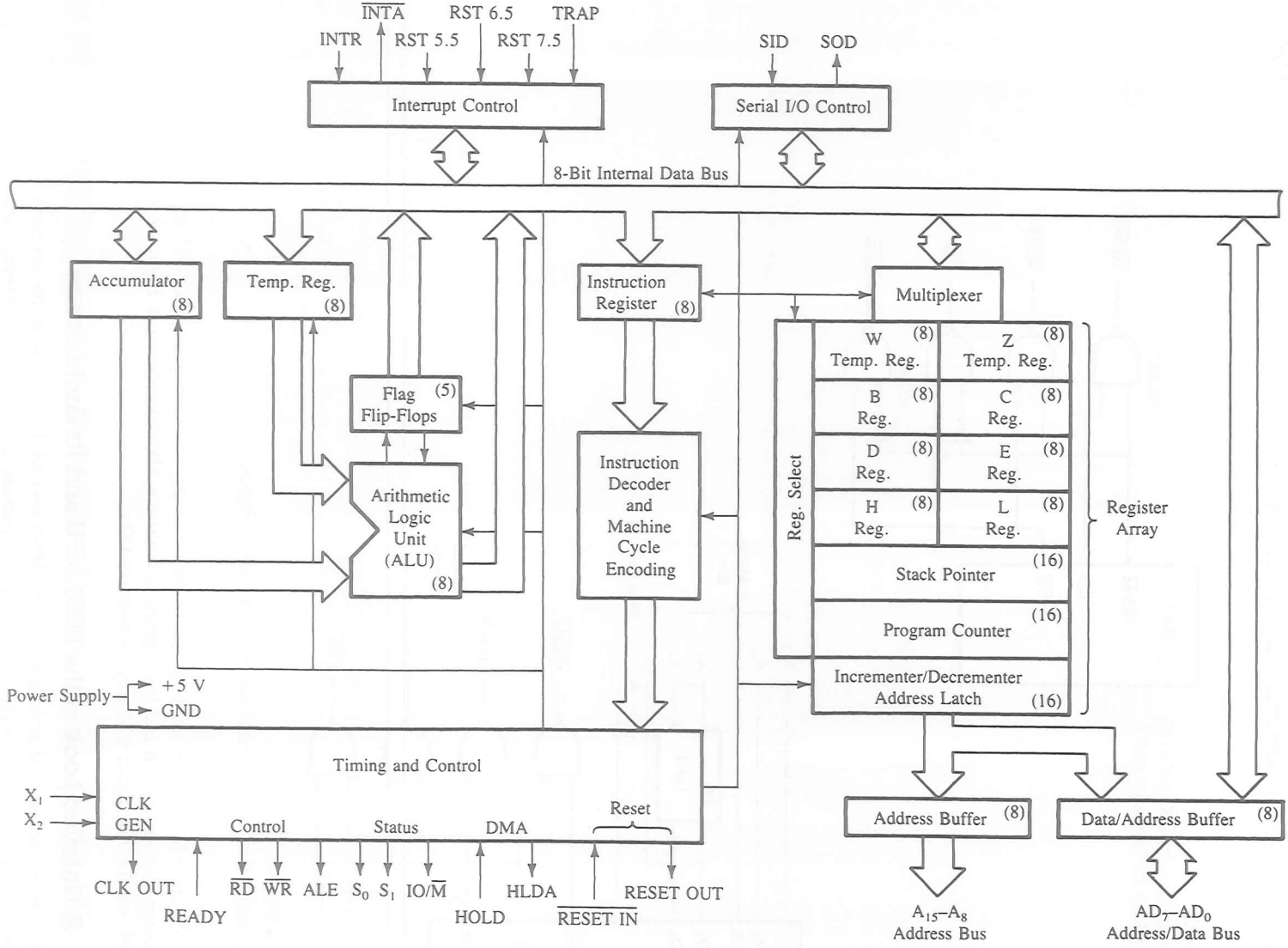



FIGURE 4.7
The 8085A Microprocessor: Functional Block Diagram

NOTE: The 8085A microprocessor is commonly known as the 8085.

SOURCE: Intel Corporation, *Embedded Microprocessors* (Santa Clara, Calif.: Author, 1994), pp. 1-11.

Control Unit, Instruction Register and Decoder, Register Array, Interrupt Control, and Serial I/O Control. We will discuss the first four units below; the last two will be discussed later in the book.

### THE ALU

The arithmetic/logic unit performs the computing functions; it includes the accumulator, the temporary register, the arithmetic and logic circuits, and five flags. The temporary register is used to hold data during an arithmetic/logic operation. The result is stored in the accumulator, and the flags (flip-flops) are set or reset according to the result of the operation.

The flags are affected by the arithmetic and logic operations in the ALU. In most of these operations, the result is stored in the accumulator. Therefore, the flags generally reflect data conditions in the accumulator—with some exceptions. The descriptions and conditions of the flags are as follows:

- **S—Sign flag:** After the execution of an arithmetic or logic operation, if bit $D_7$ of the result (usually in the accumulator) is 1, the Sign flag is set. This flag is used with signed numbers. In a given byte, if $D_7$ is 1, the number will be viewed as a negative number; if it is 0, the number will be considered positive. In arithmetic operations with signed numbers, bit $D_7$ is reserved for indicating the sign, and the remaining seven bits are used to represent the magnitude of a number. However, this flag is irrelevant for the operations of unsigned numbers. Therefore, for unsigned numbers, even if bit $D_7$ of a result is 1 and the flag is set, it does not mean the result is negative. (See Appendix A2 for a discussion of signed numbers.)
- **Z—Zero flag:** The Zero flag is set if the ALU operation results in 0, and the flag is reset if the result is not 0. This flag is modified by the results in the accumulator as well as in the other registers.
- **AC—Auxiliary Carry flag:** In an arithmetic operation, when a carry is generated by digit $D_3$ and passed on to digit $D_4$, the AC flag is set. The flag is used only internally for BCD (binary-coded decimal) operations and is not available for the programmer to change the sequence of a program with a jump instruction.
- **P—Parity flag:** After an arithmetic or logical operation, if the result has an even number of 1s, the flag is set. If it has an odd number of 1s, the flag is reset. (For example, the data byte 0000 0011 has even parity even if the magnitude of the number is odd.)
- **CY—Carry flag:** If an arithmetic operation results in a carry, the Carry flag is set; otherwise it is reset. The Carry flag also serves as a borrow flag for subtraction.

The bit positions reserved for these flags in the flag register are as follows:

| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|---|---|
| S | Z | | AC | | P | | CY |

Among the five flags, the AC flag is used internally for BCD arithmetic; the instruction set does not include any conditional jump instructions based on the AC flag. Of the remaining four flags, the Z and CY flags are those most commonly used.

TIMING AND CONTROL UNIT

This unit synchronizes all the microprocessor operations with the clock and generates the control signals necessary for communication between the microprocessor and peripherals. The control signals are similar to a sync pulse in an oscilloscope. The $\overline{RD}$ and $\overline{WR}$ signals are sync pulses indicating the availability of data on the data bus.

INSTRUCTION REGISTER AND DECODER

The instruction register and the decoder are part of the ALU. When an instruction is fetched from memory, it is loaded in the instruction register. The decoder decodes the instruction and establishes the sequence of events to follow. The instruction register is not programmable and cannot be accessed through any instruction.

REGISTER ARRAY

The programmable registers were discussed in the last chapter. Two additional registers, called temporary registers W and Z, are included in the register array. These registers are used to hold 8-bit data during the execution of some instructions. However, because they are used internally, they are not available to the programmer.

### 4.1.6 Decoding and Executing an Instruction

Decoding and executing an instruction after it has been fetched can be illustrated with the example from Section 4.12.

---

**Example 4.2** Assume that the accumulator contains data byte 82H, and the instruction MOV C,A (4FH) is fetched. List the steps in decoding and executing the instruction.

**Solution** This example is similar to the example in Section 4.12, except that the contents of the accumulator are specified. To decode and execute the instruction, the following steps are performed.

The microprocessor:

1. Places the contents of the data bus (4FH) in the instruction register (Figure 4.8) and decodes the instruction.
2. Transfers the contents of the accumulator (82H) to the temporary register in the ALU.
3. Transfers the contents of the temporary register to register C.

---

### 4.1.7 Review of Important Concepts

1. The 8085 microprocessor has a multiplexed bus $AD_7$–$AD_0$ used as the lower-order address bus and the data bus.
2. The bus $AD_7$–$AD_0$ can be demultiplexed by using a latch and the ALE signal.
3. The 8085 has a status signal IO/$\overline{M}$ and two control signals $\overline{RD}$ and $\overline{WR}$. By ANDing these signals, four control signals can be generated: $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, and $\overline{IOW}$.

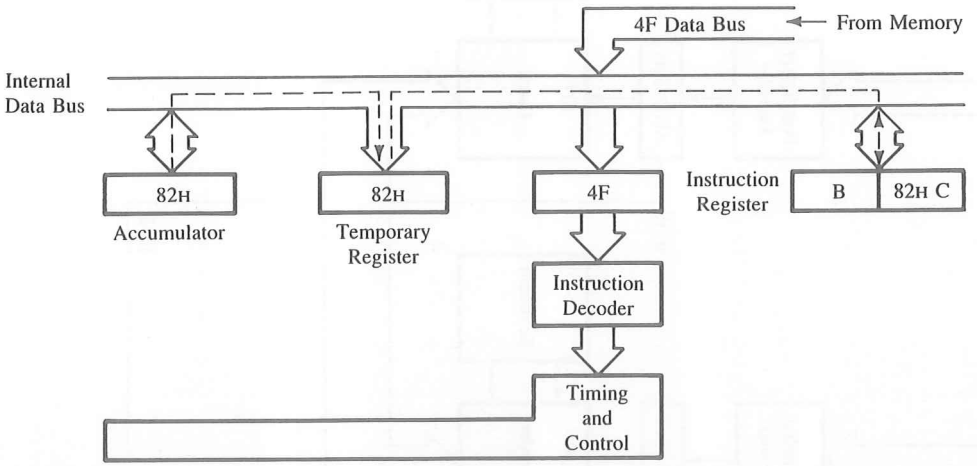



FIGURE 4.8
Instruction Decoding and Execution

**4.** The 8085 MPU

- ☐ transfers data from memory locations to the microprocessor by using the control signal Memory Read ($\overline{\text{MEMR}}$—active low). This is also called **reading from memory.** The term *data* refers to any byte that is placed on the data bus; the byte can be an instruction code, data, or an address.
- ☐ transfers data from the microprocessor to memory by using the control signal Memory Write ($\overline{\text{MEMW}}$—active low). This is also called **writing into memory.**
- ☐ accepts data from input devices by using the control signal I/O Read ($\overline{\text{IOR}}$—active low). This is also known as **reading from an input port.**
- ☐ sends data to output devices by using the control signal I/O Write ($\overline{\text{IOW}}$—active low). This is also known as **writing to an output port.**

**5.** To execute an instruction, the MPU

- ☐ places the memory address of the instruction on the address bus.
- ☐ indicates the operation status on the status lines.
- ☐ sends the $\overline{\text{MEMR}}$ control signal to enable the memory, fetches the instruction byte, and places it in the instruction decoder.
- ☐ executes the instruction.

## 4.2 EXAMPLE OF AN 8085-BASED MICROCOMPUTER

A general microcomputer system was illustrated in Figure 3.15, in the last chapter. After our discussion of the 8085 microprocessor and the interfacing devices, we can expand the system to include more details, as shown in Figure 4.9. The system includes interfacing devices such as buffers, decoders, and latches.