8085 and 8086 Microprocessor Architectures

2

Objectives

At the conclusion of this chapter, you should be able to:

1. *Understand the organization of 8-bit microprocessor 8085.*
2. *Should be able to build a microcomputer based on 8085.*
3. *Describe how a microprocessor fetches and executes an instruction.*
4. *Familiarize with the features of 16-bit microprocessor 8086.*
5. *List the registers and other parts in 8086 execution unit and bus interface unit.*
6. *Describe functions of the 8086 queue.*
7. *Demonstrate how the 8086 calculates memory addresses.*

This chapter will help us get an overview of the two basic Microprocessor families – viz. 8085 Microprocessor and 8086 Microprocessor.

THE 8085 MICROPROCESSOR FAMILY OVERVIEW

The Intel 4-bit processor introduced in 1971 was used as a simple calculator. Next, Intel introduced the first 8-bit processor 8080 which used two power supplies but was not well received in the market. Then the 8-bit 8085 processor, fully compatible with the 8080 and with a single power supply was released, and this processor was well received and is widely used.

Even today, one can see the use of the 8085 processor in some applications. It is a basic CPU with a fixed-point ALU for binary and decimal arithmetic, hardwired control unit, and some registers. It does not have even the binary MULTIPLY and DIVIDE machine instructions. We will now see the main features of this widely used 8-bit processor, that is, 8085.

Main Features of the 8-bit Microprocessor 8085

- 8-bit parallel processing
- Single +5 volt supply
- Basic clock speed is 3 MHz for 8085A, 5 MHz for 8085A-2
- 12 addressable 8-bit registers, four of them can function only as two 16-bit register pairs
- Six others can be used interchangeably as 8-bit registers or as 16-bit register pairs
- Uses a multiplexed address databus, AD0-AD7
- 8-bit unidirectional address bus, A8-A15
- 4 maskable interrupts and 1 non-maskable interrupt
- Direct Memory addressing (DMA) capability
- Single bit serial-in and parallel-out facility

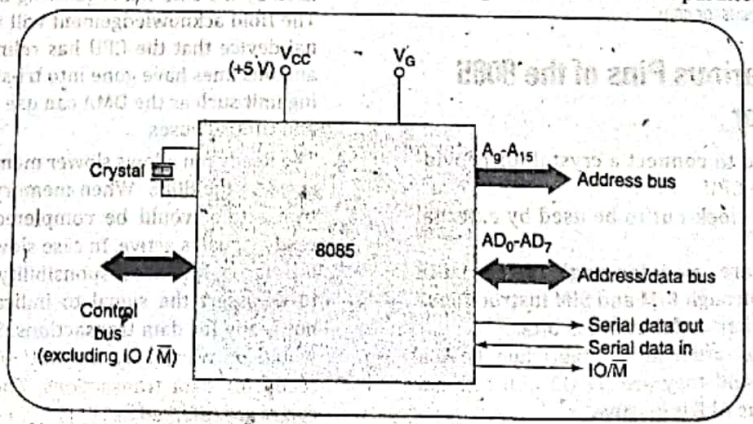

Fig. 2.1 Block diagram of Intel 8085.

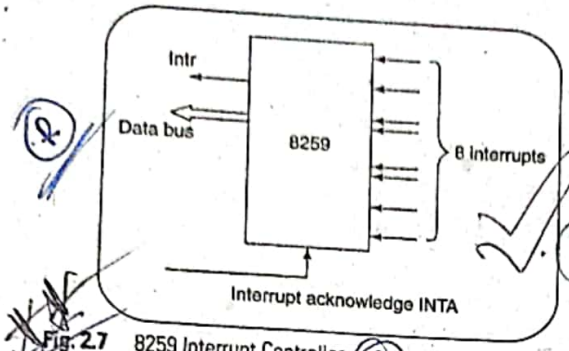

Fig. 2.7 8259 Interrupt Controller

is masked or not. If the current interrupt has higher priority and is unmasked, that interrupt is serviced. For servicing the interrupt, the 8259 will send INTR signal to 8085. In response to this INTR, when 8085 accepts this interrupt, it sends three INTA/ signals one by one to 8259. In response to the first, second and third INTA/ signals, 8259 supplies CALL opcode, low byte of call address and high byte of call address respectively. When the 8085 receives the CALL opcode and 16-bit address, it saves the program counter contents (PC) in stacks, and loads the CALL address into PC. Consequently, 8085 starts executing the corresponding interrupt service routine. This is how external interrupts are handled. 8259 can be cascaded to accept a maximum of 64 interrupts and can also be used with the 8086 microprocessor. Figure 2.7 shows the 8259 diagram with all the input and output signals.

The 8085 processor configuration with 16-bit address bus, 8-bit data bus and control bus is shown in Fig. 2.8. Memories, general-purpose serial (Intel 8251) and parallel (8255) peripheral interfaces, DMA controller (8237) can be connected with peripherals to form a desired computer system. Interfacing details are discussed later.

Interfacing of memories to the system bus is discussed in Chapter 8. Peripheral interfacing to the system bus is explained in Chapter 9.

Having seen briefly the architecture of 8085 microprocessor, we shall now study the architecture of the 16-bit processor 8086.

MAIN FEATURES OF 8086

1. 8086 is a 16-bit processor. Its ALU, internal registers work with 16-bit binary words.
2. 8086 has a 16-bit data bus. It can read or write data to a memory/port, either 16 bits or 8 bits at a time.
3. 8086 has a 20-bit address bus which means it can address up to 220 = 1MB memory location
4. Frequency range of the 8086 is 6–10 MHz.
5. Like 8085, 8086 too can do only fixed-point arithmetic as the integrated circuit technology of that time did not permit to put additional circuitry on 8086 to do floating-point operations. Intel had designed the coprocessor 8087 that can do floating-point arithmetic and other complex mathematical operations. The 8086 can work in conjunction with 8087 to do both fixed-point, floating-point and other complex mathematical functions.
6. The 8086 is designed to operate in two modes, minimum and maximum. In the *minimum mode*, the 8086 processor works in a single processor environment and generates control bus signals shown in parentheses next to pins 24 through 31 in the 8086 pin diagram of Fig. 2.9. The *maximum mode* is designed to be used to work with the coprocessor 8087 and generates signals listed next to pins 24 through 31.
7. The 8086 works in a multiprocessor environment. Control signals for memory and I/O are generated by an external BUS controller.
8. It can pre-fetch up to six instruction bytes from memory and queues them in order to speed up instruction execution.
9. It requires +5 V power supply.
10. It uses a 40-pin dual in line package.
11. 8086 has two blocks— BIU and EU.

The *BIU* performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating addresses of the memory operands, prefetch of up to six bytes of instruction code. The instruction bytes are transferred to the instruction queue. The *EU* executes instructions from the instruction system byte queue.

IMPORTANT 8086 PIN DIAGRAM/ DESCRIPTION

AD15±AD0 (ADDRESS DATA BUS)

These lines constitute the time multiplexed memory/IO address and data bus.

ALE (ADDRESS LATCH ENABLE)

A HIGH on this line causes the lower order 16-bit address bus to be latched, which stores the addresses and then, the lower order 16 bits of the address bus can be used as data bus.

READY

READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer.

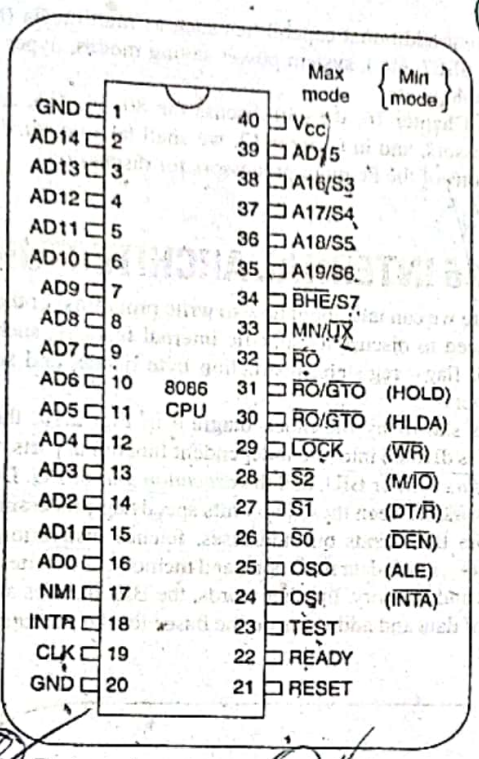

Fig. 2.8 The 8086 pin assignments.

INTR (INTERRUPT REQUEST)

It is a level-triggered input that is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored through an interrupt vector look-up table located in the system memory. It can be internally masked by software resetting the interrupt enable bit. The INTR is internally synchronized. This signal is active HIGH.

INTA

Interrupt Acknowledge from the MP.

NMI NON-MASKABLE INTERRUPT

An edge-triggered input that causes an interrupt request to the MP. A subroutine is vectored using an interrupt vector look-up table located in the system memory. The NMI is not maskable internally by software.

RESET

This causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It then restarts execution.

THE 8086 MICROPROCESSOR FAMILY—AN OVERVIEW

The Intel 8086 is a 16-bit microprocessor that is intended to be used as the CPU in a microcomputer. The term *16-bit* means that its arithmetic logic unit, its internal registers, and most of its instructions are designed to work with 16-bit binary words. The 8086 has a 16-bit data bus, so it can read data from or write data to memory and ports either 16 bits or 8 bits at a time. The 8086 has a 20-bit address bus, so it can address any one of 2^{20}, or 1,048,576, memory locations.

Each of the 1,048,576 memory addresses of the 8086 represents a byte-wide location. Sixteen-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read the entire word in one operation. If the first byte of the word is at an odd address, the 8086 will read the first byte with one bus operation and the second byte with another bus operation. Later we will discuss this in detail. The main point here is that if the first byte of a 16-bit word is at an even address, the 8086 can read the entire word in one operation.

The Intel 8088 has the same ALU, the same registers, and the same instruction set as the 8086. The 8088 also has a 20-bit address bus, so it can address any one of 1,048,576 bytes in memory. The 8088, however, has an 8-bit data bus, so it can only read data from or write data to memory and ports, 8 bits at a time. The 8086, remember, can read or write either 8 or 16 bits at a time. To read a 16-bit word from two successive memory locations, the 8088 will always have to do two read operations. Since the 8086 and the 8088 are almost identical, any reference we make to the 8086 in the rest of the book will also pertain to the 8088 unless we specifically indicate otherwise. This is done to make reading easier. The Intel 8088, incidentally, is used as the CPU in the original IBM Personal Computer, the IBM PC/XT, and several compatible personal computers.

The Intel 80186 is an improved version of the 8086, and the 80188 is an improved version of the 8088. In addition to a 16-bit CPU, the 80186 and 80188 each have programmable peripheral devices integrated in the same package. In a later chapter we will discuss these integrated peripherals. The instruction set of the 80186 and 80188 is a *superset* of the instruction set of the 8086. The term *superset* means that all the 8086 and 8088 instructions will execute properly on an 80186 or an 80188, but the 80186 and the 80188 have a few additional instructions. In other words, a program written for an 8086 or an 8088 is *upward-compatible* to an 80186 or an 80188, but a program written for an 80186 or an 80188 may not execute correctly on an 8086 or an 8088. In the instruction set descriptions in Chapter 6, we specifically indicate which instructions work only with the 80186 or 80188.

The Intel 80286 is a 16 bit, advanced version of the 8086 which was specifically designed for use as the CPU in a multiuser or multitasking microcomputer. When operating in

its *real address mode*, the 80286 functions mostly as a fast 8086. Most programs written for an 8086 can be run on an 80286 operating in its real address mode. When operating in its *virtual address mode*, an 80286 has features which make it easy to keep users' programs separate from one another and to protect the system program from destruction by users' programs. In Chapter 15, we discuss the operation and use of the 80286. The 80286 is the CPU used in the IBM PC/AT personal computer.

With the 80386 processor, Intel started the 32-bit processor architecture, known as the IA-32 architecture. This architecture extended all the address and general purpose registers to 32-bits, which gave the processor the capability to handle 32-bit address (4 GB of memory addressing), with 32-bit data, and yet accommodating all the software designed for the earlier 16-bit processors, 8086,8088, 80186, 80188 and 80286. It contains more sophisticated features for use in multiuser and multitasking environments.

Intel 80486 is the next member of the IA-32 architecture. This processor has the floating point processor (80387) integrated into the CPU chip itself. These processors are then followed by different versions of the Pentium Processors, with different additional capabilities such as multimedia (MMX, SSE, SSE2, etc), system power saving modes, hyper thread technology etc.

In Chapter 16, we will discuss the 80286, 386, and 486 processors, and in Chapter 17, we shall take up some major versions of the Pentium processors for discussion.

8086 INTERNAL ARCHITECTURE

Before we can talk about how to write programs for the 8086, we need to discuss its specific internal features, such as its ALU, flags, registers, instruction byte queue, and segment registers.

As shown by the block diagram in Fig. 2.10, the 8086 CPU is divided into two independent functional parts, the *bus interface unit* or BIU, and the *execution unit* or EU. Dividing the work between these two units speeds up processing.

The BIU sends out addresses, fetches instructions from memory, reads data from ports and memory, and writes data to ports and memory. In other words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

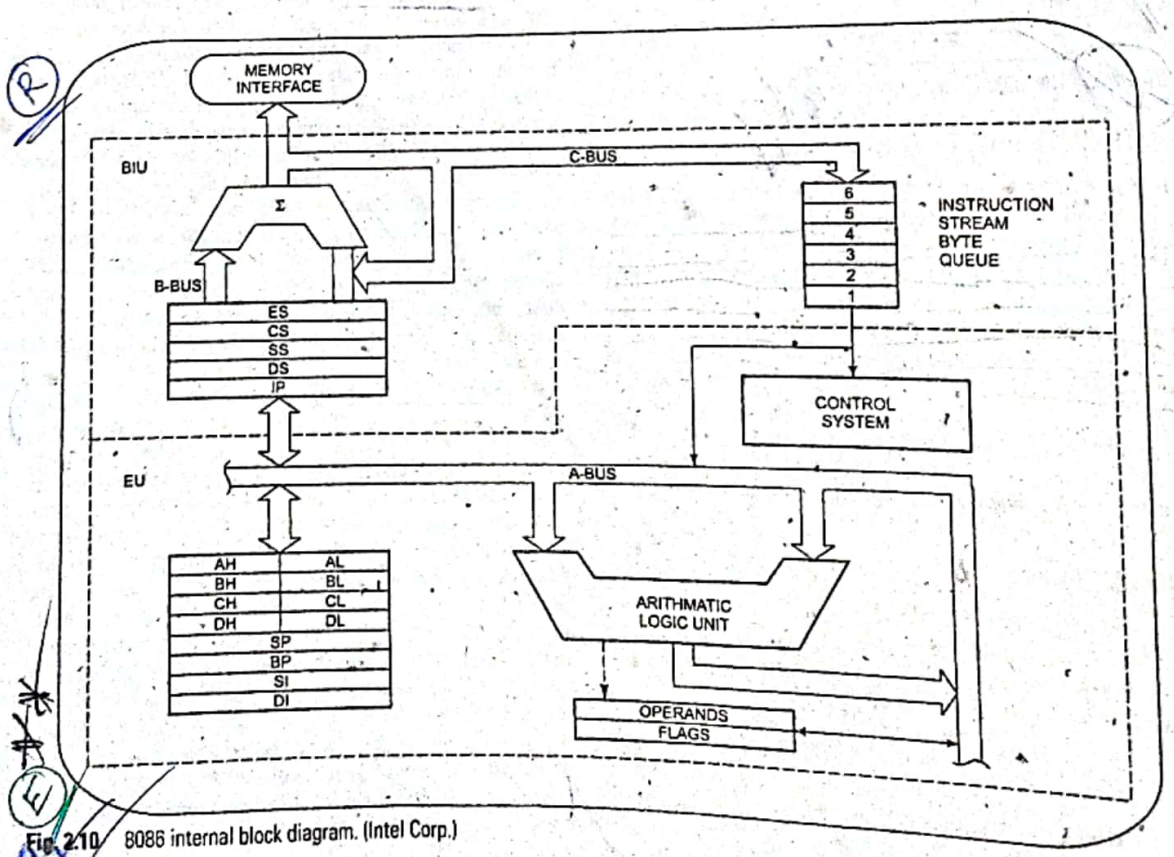

Fig. 2.10 8086 internal block diagram. (Intel Corp.)

The execution unit of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions, and executes instructions. Let's take a look at some of the parts of the execution unit.

The Execution Unit

CONTROL CIRCUITRY, INSTRUCTION DECODER, AND ALU

As shown in Fig. 2.10, the EU contains *control circuitry* which directs internal operations. A *decoder* in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU has a 16-bit *arithmetic logic unit* which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.

FLAG REGISTER

A *flag* is a flip-flop that indicates some condition produced by the execution of an instruction or controls certain operations of the EU. A 16-bit *flag register* in the EU contains nine active flags. Figure 2.11 shows the location of the nine flags in the flag register. Six of the nine flags are used to indicate some *condition* produced by an instruction. For example, a flip-flop called the *carry flag* will be set to a 1 if the addition of two 16-bit binary numbers produces a carry out of the most significant bit position. If no carry out of the MSB is produced by the addition, then the carry flag will be a 0. The EU, thus effectively runs up a "flag" to tell you that a carry was produced.

The six conditional flags in this group are the *carry flag* (CF), the *parity flag* (PF), the *auxiliary carry flag* (AF), the *zero flag* (ZF), the *sign flag* (SF), and the *overflow flag* (OF). The names of these flags should give you hints as to what conditions affect them. Certain 8086 instructions check these flags to determine which of two alternative actions should be done in executing the instruction.

The three remaining flags in the flag register are used to *control* certain operations of the processor. These flags are different from the six conditional flags described above in the way they are set or reset. The six conditional flags are set or reset by the EU on the basis of the results of some arithmetic or logic operation. The *control flags* are deliberately set or reset with specific instructions you put in your program. The three control flags are the *trap flag* (TF), which is used for single stepping through a program; the *interrupt flag* (IF), which is used to allow or prohibit the interruption of a program; and the *direction flag* (DF), which is used with string instructions.

Later we will discuss in detail the operation and use of the nine flags.

GENERAL-PURPOSE REGISTERS

Observe in Fig. 2.10 that the EU has eight *general-purpose registers*, labeled AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually for temporary storage of 8-bit data. The AL register is also called the *accumulator*. It has some features that the other general-purpose registers do not have.

Certain pairs of these general-purpose registers can be used together to store 16-bit data words. The acceptable register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. The AH–AL pair is referred to as the *AX register*, the BH-BL pair is referred to as the *BX register*, the CH–CL pair is referred to as the *CX register*, and the DH–DL pair is referred to as the *DX register*.

The 8086 general-purpose register set is very similar to those of the earlier generation 8080 and 8085 microprocessors. It was designed this way so that the many programs written for the 8080 and 8085 could easily be translated to run on the 8086 or the 8088. The advantage of using internal registers for the temporary storage of data is that, since the data is already in the EU, it can be accessed much more quickly than it could be accessed in external memory. Now let's look at the features of the BIU.

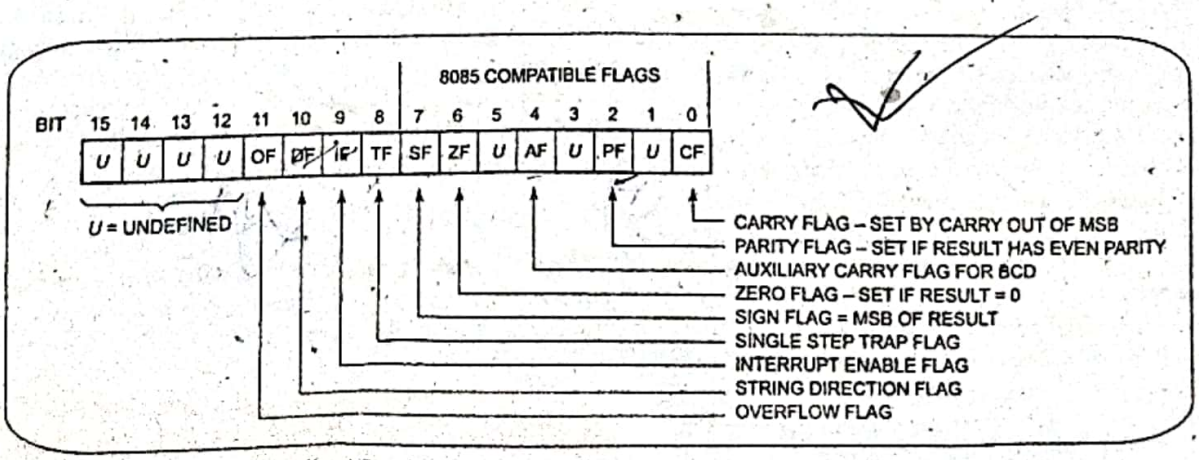

Fig. 2.11 8086 flag register format. (Intel Corp.)

The BIU

THE QUEUE

While the EU is decoding an instruction or executing an instruction which does not require use of the buses, the BIU fetches up to six instruction bytes for the following instructions. The BIU stores these prefetched bytes in a first-in-first-out register set called a *queue*. When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. The process is analogous to the way a bricklayer's assistant fetches bricks ahead of time and keeps a queue of bricks lined up so that the bricklayer can just reach out and grab a brick when necessary. Except in the cases of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called *pipelining*.

SEGMENT REGISTERS

The 8086 BIU sends out 20-bit addresses, so it can address any of 2^{20} or 1,048,576 bytes in memory. However, at any given time the 8086 works with only four 65,536-byte (64-Kbyte) segments within this 1,048,576-byte (1-Mbyte) range. Four *segment registers* in the BIU are used to hold the upper 16 bits of the starting addresses of four memory segments that the 8086 is working with at a particular time. The four segment registers are the *code segment* (CS) register, the *stack segment* (SS) register, the *extra segment* (ES) register, and the *data segment* (DS) register.

Figure 2.12 shows how these four segments might be positioned in memory at a given time. The four segments can be separated as shown, or, for small programs which do not need all 64 Kbytes in each segment, they can overlap.

To repeat, then, a segment register is used to hold the upper 16 bits of the starting address for each of the segments. The code segment register, for example, holds the upper 16 bits of the starting address for the segment from which the BIU is currently fetching instruction code bytes. The BIU always inserts zeros for the lowest 4 bits (nibble) of the 20-bit starting address for a segment. If the code segment register contains 348AH, for example, then the code segment will start at address 348A0H. In other words, a 64-Kbyte segment can be located anywhere within the 1-Mbyte address space, but the segment will always start at an address with zeros in the lowest 4 bits. This constraint was put on the location of segments so that it is only necessary to store and manipulate 16-bit numbers when working with the starting address of a segment. The part of a segment starting address stored in a segment register is often called the *segment base*.

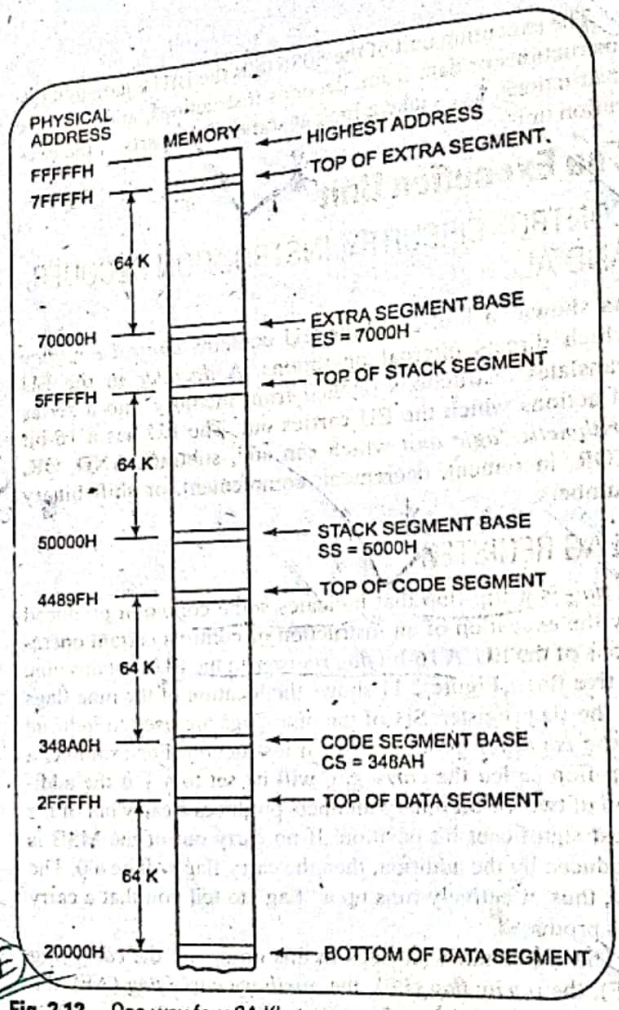

Fig. 2.12 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

A stack is a section of memory set aside to store addresses and data while a *subprogram* executes. The stack segment register is used to hold the upper 16 bits of the starting address for the program stack. We will discuss the use and operation of a stack in detail later.

The extra segment register and the data segment register are used to hold the upper 16 bits of the starting addresses of two memory segments that are used for data.

INSTRUCTION POINTER

The next feature to look at in the BIU is the *instruction pointer* (IP) register. As discussed previously, the code segment register holds the upper 16 bits of the starting address of the segment from which the BIU is currently fetching instruction code bytes. The instruction pointer register holds the 16-bit address, or *offset*, of the next code byte *within* this code segment. The value contained in the IP is referred to as

an *offset* because this value must be offset from (added to) the segment base address in CS to produce the required 20-bit physical address sent out by the BIU. Fig. 2.13*a* shows in diagram form how this works. The CS register points to the *base* or start of the current code segment. The IP contains the distance or offset from this base address to the next instruction byte to be fetched. Fig. 2.13*b* shows how the 16-bit offset in IP is added to the 16-bit segment base address in CS to produce the 20-bit *physical* address. Notice that the two 16-bit numbers are not added directly in line, because the CS register contains only the upper 16 bits of the base address for the code segment. As we said before, the BIU automatically inserts zeros for the lowest 4 bits of the segment base address.

If the CS register, for example, contains 348AH, you know that the starting address for the code segment is 348A0H. When the BIU adds the offset of 4214H in the IP to this segment base address, the result is a 20-bit physical address of 38AB4H.

An alternative way of representing a 20-bit physical address is the *segment base:offset form*. For the address of a code byte, the format for this alternative form will be CS:IP. As an example of this, the address constructed in the preceding paragraph, 38AB4H, can also be represented as 348A:4214.

To summarize, then, the CS register contains the upper 16 bits of the starting address of the code segment in the 1-Mbyte address range of the 8086. The instruction pointer register contains a 16-bit offset which tells, where in that 64-Kbyte code segment the next instruction byte is to be fetched from. The actual physical address sent to memory is produced by adding the offset contained in the IP register to the segment base represented by the upper 16 bits in the CS register.

Any time the 8086 accesses memory, the BIU produces the required 20-bit physical address by adding an offset to a segment base value represented by the contents of one of the segment registers. As another example of this, let's look at how the 8086 uses the contents of the stack segment register and the contents of the stack pointer register to produce a physical address.

STACK SEGMENT REGISTER AND STACK POINTER REGISTER

A stack, remember, is a section of memory set aside to store addresses and data while a subprogram is executing. The 8086 allows you to set aside an entire 64-Kbyte segment as a stack. The upper 16 bits of the starting address for this segment are kept in the stack segment register. The *stack pointer* (SP) register in the execution unit holds the 16-bit offset from the start of the segment to the memory location where a word was most recently stored on the stack. The memory location where a word was most recently stored is called the *top of stack*. Figure 2.14*a* shows this in diagram form.

The physical address for a stack read or a stack write is produced by adding the contents of the stack pointer register to the segment base address represented by the upper 16 bits of the base address in SS. Fig. 2.14*b* shows an example. The 5000H in SS represents a segment base address of 50000H.

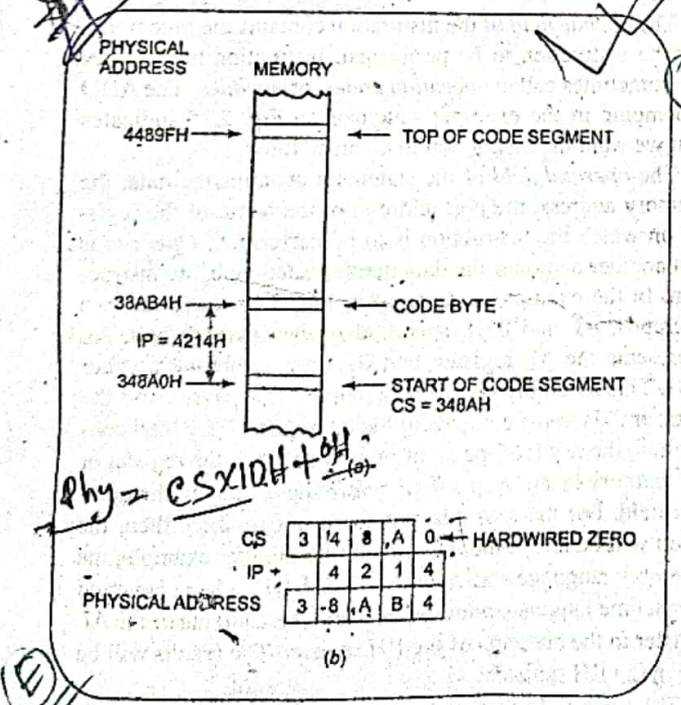

Fig. 2.13 Addition of IP to CS to produce the physical address of the code byte, (a) Diagram, (b) Computation.

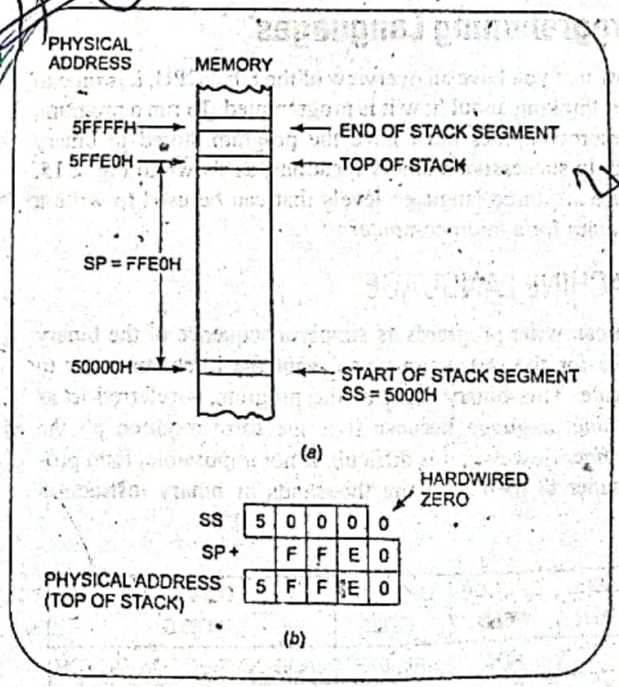

Fig. 2.14 Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram, (b) Computation.

When the FFE0H in the SP is added to this, the resultant physical address for the top of the stack will be 5FFE0H. The physical address can be represented either as a single number, 5FFE0H, or in SS:SP form as 5000:FFE0H.

The operation and use of the stack will be discussed in detail later as need arises.

POINTER AND INDEX REGISTERS IN THE EXECUTION UNIT

In addition to the stack pointer register (SP), the EU contains a 16-bit *base pointer* (BP) register. It also contains a 16-bit *source index* (SI) register and a 16-bit *destination index* (DI) register. These three registers can be used for temporary storage of data just as the general-purpose registers described above. However, their main use is to hold the 16-bit offset of a data word in one of the segments. SI, for example, can be used to hold the offset of a data word in the data segment. The physical address of the data in memory will be generated in this case by adding the contents of SI to the segment base address represented by the 16-bit number in the DS register. After we give you an overview of the different levels of languages used to program a microcomputer, we will show you some examples of how we tell the 8086 to read data from or write data to a desired memory location.

INTRODUCTION TO PROGRAMMING THE 8086

Programming Languages

Now that you have an overview of the 8086 CPU, it is time to start thinking about how it is programmed. To run a program, a microcomputer must have the program stored in binary form in successive memory locations, as shown in Fig. 2.15. There are three language levels that can be used to write a program for a microcomputer.

MACHINE LANGUAGE

You can write programs as simply a sequence of the binary codes for the instructions you want the microcomputer to execute. This binary form of the program is referred to as *machine language* because it is the form required by the machine. However, it is difficult, if not impossible, for a programmer to memorize the thousands of binary instruction codes for a CPU such as the 8086. Also, it is very easy for an error to occur when working with long series of 1's and 0's. Using hexadecimal representation for the binary codes might help some, but there are still thousands of instruction codes to cope with.

LABEL FIELD	OP CODE FIELD	OPERAND FIELD	COMMENT FIELD
NEXT:	ADD	AL, 07H	;ADD CORRECTION FACTOR

Fig. 2.15 Assembly language program statement format.

ASSEMBLY LANGUAGE

To make programming easier, many programmers write programs in *assembly language*. They then translate the assembly language program to machine language so that it can be loaded into memory and run. Assembly language uses two-, three-, or four-letter *mnemonics* to represent each instruction type. A mnemonic is just a device to help you remember something. The letters in an assembly language mnemonic are usually initials or a shortened form of the English word(s) for the operation performed by the instruction. For example, the mnemonic for subtract is SUB, the mnemonic for Exclusive OR is XOR, and the mnemonic for the instruction to copy data from one location to another is MOV.

Assembly language statements are usually written in a standard form that has *four fields*, as shown in Fig. 2.15. The first field in an assembly language statement is the *label field*. *A label* is a symbol or group of symbols used to represent an address which is not specifically known at the time the statement is written. Labels are usually followed by a colon. Labels are not required in a statement, they are just inserted where they are needed. We will show later many uses of labels.

The *opcode field* of the instruction contains the mnemonic for the instruction to be performed. Instruction mnemonics are sometimes called *operation codes*, or *opcodes*. The ADD mnemonic in the example statement in Fig. 2.15 indicates that we want the instruction to do an addition.

The *operand field* of the statement contains the data, the memory address, the port address, or the name of the register on which the instruction is to be performed. *Operand* is just another name for the data item(s) acted on by an instruction. In the example instruction in Fig. 2.15, there are two operands, AL and 07H, specified in the operand field. AL represents the AL register, and 07H represents the number 07H. This assembly language statement thus says, "Add the number 07H to the contents of the AL register." By Intel convention, the result of the addition will be put in the register or the memory location specified *before* the comma in the operand field. For the example statement in Fig. 2.15, then, the result will be left in the AL register. As another example, the assembly language statement ADD BH, AL, when converted to machine language and run, will add the contents of the AL register to the contents of the BH register. The results will be left in the BH register.

The final field in an assembly language statement such as that in Fig. 2.15 is the *commentfield*, which starts with a

semicolon. Comments do not become part of the machine language program, but they are very important. You write *comments* in a program to remind you of the function that an instruction or group of instructions performs in the program.

To summarize why assembly language is easier to use than machine language, let's look a little more closely at the assembly language ADD statement. The general format of the 8086 ADD instruction is

ADD Destination, Source

The *source* can be a number written in the instruction, the contents of a specified register, or the contents of a memory location. The *destination* can be a specified register or a specified memory location. However, the source and the destination in an instruction cannot both be memory locations.

A later section on 8086 addressing modes will show all the ways in which the source of an operand and the destination of the result can be specified. The point here is that the single mnemonic ADD, together with a specified source and a specified destination, can represent a great many 8086 instructions in an easily understandable form.

The question that may occur to you at this point is, "If I write a program in assembly language, how do I get it translated into machine language which can be loaded into the microcomputer and executed?" There are two answers to this question. The first method of doing the translation is to work out the binary code for each instruction, a bit at a time using the templates given in the manufacturer's data books. We will show you how to do this in the next chapter, but it is a tedious and error-prone task. The second method of doing the translation is with an assembler. An assembler is a program which can be run on a personal computer or *microcomputer development system*. It reads the file of assembly language instructions you write and generates the correct binary code for each. For developing all but the simplest assembly language programs, an assembler and other program development tools are essential. We will introduce you to these program development tools in the next chapter and describe their use throughout the rest of this book.

HIGH-LEVEL LANGUAGES

Another way of writing a program for a microcomputer is with a *high-level language*, such as BASIC, Pascal, or C. These languages use program statements which are even more English-like than those of assembly language. Each high-level statement may represent many machine code instructions. An *interpreter program* or a *compiler program* is used to translate higher- level language statements to machine codes which can be loaded into memory and executed. Programs can usually be written faster in high-level languages than in assembly language because a high-level language works with bigger building blocks. However, programs written in a high-level language and interpreted or compiled almost always execute more slowly and require more memory than the same programs written in assembly language. Programs that involve a lot of hardware control, such as robots and factory control systems, or programs that must run as quickly as possible are usually best written in assembly language. Complex data processing programs that manipulate massive amounts of data, such as insurance company records, are usually best written in a high-level language. The decision concerning which language to use has recently been made more difficult by the fact that current assemblers allow the use of many high-level language features, and the fact that some current high-level languages provide assembly language features.

OUR CHOICE

For most of this book we work very closely with hardware, so assembly language is the best choice. In later chapters, however, we do show you how to write programs which contain modules written in assembly language and modules written in the high-level language C. In the next chapter we introduce you to assembly language programming techniques. Before we go on to that, however, we will use a few simple 8086 instructions to show you more about accessing data in registers and memory locations.

How the 8086 Accesses Immediate and Register Data

In a previous discussion of the 8086 BIU, we described how the 8086 accesses code bytes using the contents of the CS and IP registers. We also described how the 8086 accesses the stack using the contents of the SS and SP registers. Before we can teach you assembly language programming techniques, we need to discuss some of the different ways in which an 8086 can access the data that it operates on. The different ways in which a processor can access data are referred to as its *addressing modes*. In assembly language statements, the addressing mode is indicated in the instruction. We will use the 8086 MOV instruction to illustrate some of the 8086 addressing modes.

The MOV instruction has the format

MOV Destination, Source

When executed, this instruction *copies* a word or a byte from the specified source location to the specified destination location. The source can be a number written directly in the instruction, a specified register, or a memory location specified in 1 of 24 different ways. The destination can be a specified register or a memory location specified in any 1 of 24 different ways. The source and the destination cannot both be memory locations in an instruction.

5-A

IMMEDIATE ADDRESSING MODE

Suppose that in a program you need to put the number 437BH in the CX register. The MOV CX, 437BH instruction can be used to do this. When it executes, this instruction will put the *immediate* hexadecimal number 437BH in the 16-bit CX register. This is referred to as *immediate addressing mode* because the number to be loaded into the CX register will be put in the two memory locations immediately following the code for the MOV instruction.

A similar instruction, MOV CL, 48H, could be used to load the 8-bit immediate number 48H into the 8-bit CL register. You can also write instructions to load an 8-bit immediate number into an 8-bit memory location or to load a 16-bit number into two consecutive memory locations, but we are not yet ready to show you how to specify these.

REGISTER ADDRESSING MODE

Register addressing mode means that a register is the source of an operand for an instruction. The instruction MOV CX, AX, for example, copies the contents of the 16- bit AX register into the 16-bit CX register. Remember that the destination location is specified in the instruction before the comma, and the source is specified after the comma. Also note that the contents of AX are just *copied* to CX, not actually moved. In other words, the previous contents of CX are written over, but the contents of AX are not changed. For example, if CX contains 2A84H and AX contains 4971H before the MOV CX, AX instruction executes, then after the instruction executes, CX will contain 4971H and AX will still contain 4971H. You can MOV any 16-bit register to any 16-bit register, or you can MOV any 8-bit register to any 8-bit register. However, you cannot use an instruction such as MOV CX, AL because this is an attempt to copy a *byte-type* operand (AL) into a *word-type* destination (CX). The byte in AL would fit in CX, but the 8086 would not know which half of CX to put it in. If you try to write an instruction like this and you are using a good assembler, the assembler will tell you that the instruction contains a *type error*. To copy the byte from AL to the high byte of CX, you can use the instruction MOV CH, AL. To copy the byte from AL to the low byte of CX, you can use the instruction MOV CL, AL.

Accessing Data in Memory

OVERVIEW OF MEMORY ADDRESSING MODES

The addressing modes described in the following sections are used to specify the location of an operand in memory. To access data in memory, the 8086 must also produce a 20-bit physical address. It does this by adding a 16-bit value called the *effective address* to a segment base address represented by the 16-bit number in one of the four segment registers. The effective address (EA) represents the *displacement* or *offset* of the desired operand from the segment base. In most cases, any of the segment bases can be specified, but the data segment is the one most often used. Fig. 2.16*a* shows in graphic form how the EA is added to the data segment base to point to an operand in memory. Fig. 2.16*b* shows how the 20-bit physical address is generated by the BIU. The starting address for the data segment in Fig. 2.16*b* is 20000H, so the data segment register will contain 2000H. The BIU adds the effective address, 437AH, to the data segment base address of 20000H to produce the physical address sent out to memory. The 20-bit physical address sent out to memory by the BIU will then be 2437AH. The physical address can be represented either as a single number 2437AH or in the segment base:offset form as 2000:437AH.

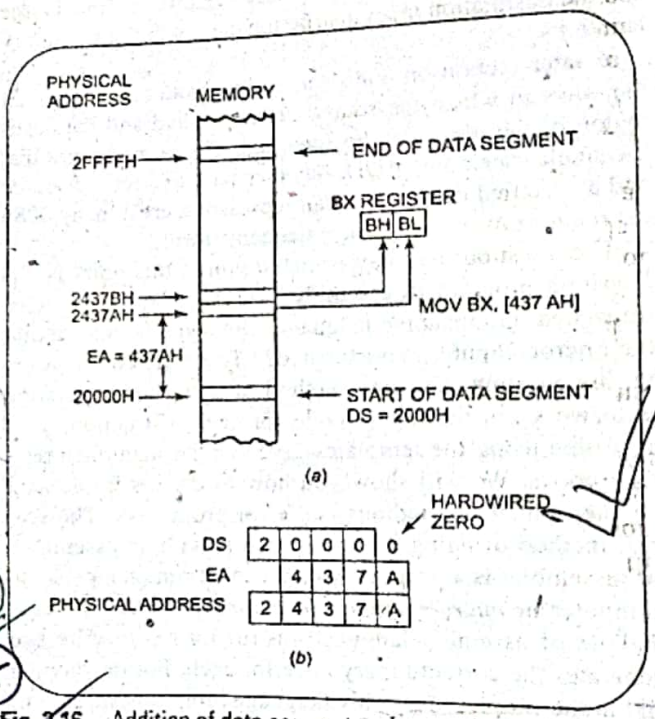

Fig. 2.16 Addition of data segment register and effective address to produce the physical address of the data byte, (a) Diagram, (b) Computation.

The execution unit calculates the effective address for an operand using information you specify in the instruction. You can tell the EU to use a number in the instruction as the effective address, to use the contents of a specified register as the effective address, or to compute the effective address by adding a number in the instruction to the contents of one or two specified registers. The following section describes one way you can tell the execution unit to calculate an effective address. In later chapters we show other ways of specifying the effective address. Later we also show how the addressing modes this provides, are used to solve some common programming problems.

DIRECT ADDRESSING MODE

For the simplest memory addressing mode, the effective address is just a 16-bit number written directly in the instruction. The instruction MOV BL, [437AH] is an example. The square brackets around the 437AH are shorthand for "the contents of the memory location(s) at a displacement from the segment base of." When executed, this instruction will copy "the contents of the memory location at a displacement from the data segment base of" 437AH into the BL register, as shown by the rightmost arrow in Fig. 2.16*a*. The BIU calculates the 20-bit physical memory address by adding the effective address 437AH to the data segment base, as shown in Fig. 2.16*b*. This addressing mode is called *direct* because the displacement of the operand from the segment base is specified directly in the instruction. The displacement in the instruction will be added to the data segment base in DS unless you tell the BIU to add it to some other segment base. Later we will show you how to do this.

Another example of the direct addressing mode is the instruction MOV BX, [437AH]. When executed, this instruction copies a 16-bit word from memory into the BX register. Since each memory address of the 8086 represents a byte of storage, the word must come from two memory locations. The byte at a displacement of 437AH from the data segment base will be copied into BL, as shown by the right arrow in Fig. 2.16*a*. The contents of the next higher address, displacement 437BH, will be copied into the BH register, as shown by the left arrow in Fig. 2.16*a*. From the instruction coding, the 8086 will automatically determine the number of bytes that it must access in memory.

An important point here is that an 8086 always stores the low byte of a word in the lower of the two addresses and stores the high byte of a word in the higher address. To stick this in your mind, remember:

Low byte—Low address, High byte—High address

The previous two examples showed how the direct addressing mode can be used to specify the source of an operand. Direct addressing can also be used to specify the destination of an operand in memory. The instruction MOV [437AH], BX, for example, will copy the contents of the BX register to two memory locations in the data segment. The contents of BL will be copied to the memory location at a displacement of 437AH. The contents of BH will be copied to the memory location at a displacement of 437BH. This operation is represented by simply reversing the direction of the arrows in Fig. 2.16*a*.

NOTE: When you are hand-coding programs using direct addressing of the form shown above, make sure to put in the square brackets to remind you how to code the instruction. If you leave the brackets out of an instruction such as MOV BX, [437AH], you will code it as if it were the instruction MOV BX, 437AH. This second instruction will load the immediate number 437AH into BX, rather than loading a word from memory at a displacement of 437AH into BX. Also note that if you are writing an instruction using direct addressing such as this for an assembler, you must write the instruction in the form MOV BL, DS:BYTE PTR [437AH] to give the assembler all the information it needs. As we will show you in the next chapter, when you are using an assembler, you usually use a name to represent the direct address rather than the actual numerical value.

A FEW WORDS ABOUT SEGMENTATION

At this point you may be wondering why Intel designed the 8086 family devices to access memory using the segment:offset approach rather than accessing memory directly with 20-bit addresses. The segment:offset scheme requires only a 16-bit number to represent the base address for a segment, and only a 16-bit offset to access any location in a segment. This means that the 8086 has to manipulate and store only 16-bit quantities instead of 20-bit quantities. This makes for an easier interface with 8- and 16-bit-wide memory boards and with the 16-bit registers in the 8086.

The second reason for segmentation has to do with the type of microcomputer in which an 8086-family CPU is likely to be used. A previous section of this chapter described briefly the operation of a timesharing microcomputer system. In a timesharing system, several users share a CPU. The CPU works on one user's program for perhaps 20 ms, then works on the next user's program for 20 ms. After working 20 ms for each of the other users, the CPU comes back to the first user's program again. Each time the CPU switches from one user's program to the next, it must access a new section of code and new sections of data. Segmentation makes this switching quite easy. Each user's program can be assigned a separate set of logical segments for its code and data. The user's program will contain offsets or displacements from these segment bases. To change from one user's program to a second user's program, all that the CPU has to do is to reload the four segment registers with the segment base addresses assigned to the second user's program. In other words, segmentation makes it easy to keep users' programs and data separate from one another, and segmentation makes it easy to switch from one user's program to another user's program. In Chapter 16 we tell you much more about the use of segmentation in multiuser systems.

A BASIC 8086 MICROCOMPUTER SYSTEM

Introduction

In previous chapters we worked with what is often called the *programmer's model* of the 8086. This model shows features such as internal registers, number of address lines, number

8086 Family Assembly Language Programming—Introduction

3

Objectives

At the conclusion of this chapter, you should be able to:

1. *Write a task list, flowchart, or pseudocode for a simple programming problem.*
2. *Write, code or assemble, and run a very simple assembly language program.*
3. *Describe the use of program development tools such as editors, assemblers, linkers, locators, debuggers, and emulators.*
4. *Properly document assembly language programs.*

PROGRAM DEVELOPMENT STEPS

Defining the Problem

The last chapter showed you the format for assembly language instructions and introduced you to a few 8086 instructions. Developing a program, however, requires more than just writing down a series of instructions. When you want to build a house, it is a good idea to first develop a complete set of plans for the house. From the plans you can see whether the house has the rooms you need, whether the rooms are efficiently placed, and whether the house is structured so that you can easily add on to it if you have more kids. You have probably seen examples of what happens when someone attempts to build a house by just putting pieces together without a plan.

Likewise, when you write a computer program, it is a good idea to start by developing a detailed plan or outline for the entire program. A good outline helps you to break down a large and seemingly overwhelming programming job into small modules which can easily be written, tested, and debugged. The more time you spend organizing your programs, the less time it will take you to write and debug them. You should *never* start writing an assembly language program by just writing down instructions! In this chapter we show you how to develop assembly language programs in a systematic way.

The first step in writing a program is to think very carefully about the problem that you want the program to solve. In other words, ask yourself many times, "What do I really want this program to do?" If you don't do this, you may write a program that works great but does not do what you need it to do. As you think about the problem, it is a good idea to write down exactly what you want the program to do and the order in which you want the program to do it. A good outline helps you to break down a large and seemingly overwhelming programming job into small modules which can easily be written, tested, and debugged. The more time you spend organizing your programs, the less time it will take you to write and debug them. You should *never* start writing an assembly language program by just writing down instructions! In this chapter we show you how to develop assembly language pro-grams in a systematic way. At this point you do not write down program statements, you just write the operations you want in general terms. An example for a simple programming problem might be

1. Read temperature from sensor.
2. Add correction factor of + 7.
3. Save result in a memory location.

For a program as simple as this, the three actions desired are very close to the eventual assembly language statements. For more complex problems, however, we develop a more extensive outline before writing the assembly language statements. The next section shows you some of the common ways of representing program operations in a program outline.

Representing Program Operations

The formula or sequence of operations used to solve a programming problem is often called the *algorithm* of the program. The following sections show you two common ways of representing the algorithm for a program or program segment.

One way of representing program operations is with owcharts. Flowcharts are a very graphic representation, and ey are useful for short program segments, especially those at deal directly with hardware. However, flowcharts use great deal of space. Consequently, the flowchart for even moderately complex program may take up several pages. often becomes difficult to follow program flow back and orth between pages. Also, since there are no agreed-upon ructures, a poor programmer can write a flowchart which amps all over the place and is even more difficult to follow. he term "logical spaghetti" comes to mind here.

A second way of representing the operations you want in program is with a top-down design approach and standard rogram structures. The overall program problem is first roken down into major functional modules. Each of these nodules is broken down into smaller and smaller modules ntil the steps in each module are obvious. The algorithms for he whole program and for each module are expressed with a tandard structure. Only three basic structures, SEQUENCE, F-THEN-ELSE, and WHILE-DO, are needed to represent ny needed program action or series of actions. However, ther useful structures such as IF-THEN, REPEAT-UNTIL, OR-DO, and CASE can be derived from these basic three. A structure can contain another structure of the same type or one of the other types. Each structure has only one entry point and one exit point. These programming structures may seem restrictive, but using them usually results in algorithms which are easy to follow. Also, as we will show you soon, if you write the algorithm for a program carefully with these standard structures, it is relatively easy to translate the algorithm to the equivalent assembly language instructions.

Finding the Right Instruction

After you get the structure of a program worked out and written down, the next step is to determine the instruction statements required to do each part of the program. Since the examples in this book are based on the 8086 family of microprocessors, now is a good time to give you an overview of the instructions the 8086 has for you to use. First, however, is a hint about how to approach these instructions.

You do not usually learn a new language by memorizing an entire dictionary of the language. A better way is to learn a few useful words and practice putting these words together in simple sentences. You can then learn more words as you need them to express more complex thoughts. Likewise, you should not try to memorize all the instructions for a microprocessor at once.

For future reference, Chapter 6 contains a dictionary of all the 8086 instructions with detailed descriptions and examples of each. As an introduction, however, the few pages here contain a list of all the 8086 instructions with a short explanation of each. Skim through the list and pick out a dozen or so instructions that seem useful and understandable. As a start, look for move, input, output, logical, and arithmetic instructions. Then look through the list again to see if you can find the instructions that you might use to do the "read temperature sensor value from a port, add + 7, and store result in memory" example program.

You can use Chapter 6 as a reference as you write programs. Here we simply list the 8086 instructions in *functional* groups with single-sentence descriptions so that you can see the types of instructions that are available to you. As you read through this section, do not expect to understand all the instructions. When you start writing programs, you will probably use this section to determine the type of instruction and Chapter 6 to get the instruction details as you need them. After you have written a few programs, you will remember most of the basic instruction types and will be able to simply look up an instruction in Chapter 6 to get any additional details you need. Chapter 4 shows you in detail how to use the move, arithmetic, logical, jump, and string instructions. Chapter 5 shows how to use the call instructions and the stack.

DATA TRANSFER INSTRUCTIONS

General-purpose byte or word transfer instructions:

MNEMONIC	DESCRIPTION
MOV	Copy byte or word from specified source to specified destination.
PUSH	Copy specified word to top of stack.
POP	Copy word from top of stack to specified location.
PUSHA	(80186/80188 only) Copy all registers to stack.
POPA	(80186/80188 only) Copy words from stack to all registers.
XCHG	Exchange bytes or exchange words.
XLAT	Translate a byte in AL using a table in memory.

Simple input and output port transfer instructions:

IN	Copy a byte or word from specified port to accumulator.
OUT	Copy a byte or word from accumulator to specified port.

Special address transfer instructions:

LEA	Load effective address of operand into specified register.
LDS	Load DS register and other specified register from memory.
LES	Load ES register and other specified register from memory.

Flag transfer instructions:

LAHF	Load (copy to) AH with the low byte of the flag register.
SAHF	Store (copy) AH register to low byte of flag register.
PUSHF	Copy flag register to top of stack.
POPF	Copy word at top of stack to flag register.

ARITHMETIC INSTRUCTIONS

Addition instructions:

ADD	Add specified byte to byte or specified word to word.
ADC	Add byte + byte + carry flag or word + word + carry flag.
INC	Increment specified byte or specified word by 1.
AAA	ASCII adjust after addition.
DAA	Decimal (BCD) adjust after addition.

Subtraction instructions:

SUB	Subtract byte from byte or word from word.
SBB	Subtract byte and carry flag from byte or word and carry flag from word.
DEC	Decrement specified byte or specified word by 1.
NEG	Negate — invert each bit of a specified byte or word and add 1 (form 2's complement).
CMP	Compare two specified bytes or two specified words.
AAS	ASCII adjust after subtraction.
DAS	Decimal (BCD) adjust after subtraction.

Multiplication instructions:

MUL	Multiply unsigned byte by byte or unsigned word by word.
IMUL	Multiply signed byte by byte or signed word by word.
AAM	ASCII adjust after multiplication.

Division instructions:

DIV	Divide unsigned word by byte or unsigned double word by word.
IDIV	Divide signed word by byte or signed double word by word.
AAD	ASCII adjust before division.
CBW	Fill upper byte of word with copies of sign bit of lower byte.
CWD	Fill upper word of double word with sign bit of lower word.

BIT MANIPULATION INSTRUCTIONS

Logical instructions:

NOT	Invert each bit of a byte or word.
AND	AND each bit in a byte or word with the corresponding bit in another byte or word.
OR	OR each bit in a byte or word with the corresponding bit in another byte or word.
XOR	Exclusive OR each bit in a byte or word with the corresponding bit in another byte or word.
TEST	AND operands to update flags, but don't change operands.

Shift instructions:

SHL/SAL	Shift bits of word or byte left, put zero(s) in LSB(s).
SHR	Shift bits of word or byte right, put zero(s) in MSB(s).
SAR	Shift bits of word or byte right, copy old MSB into new MSB.

Rotate instructions:

ROL	Rotate bits of byte or word left, MSB to LSB and to CF.
ROR	Rotate bits of byte or word right, LSB to MSB and to CF.
RCL	Rotate bits of byte or word left, MSB to CF and CF to LSB.
RCR	Rotate bits of byte or word right, LSB to CF and CF to MSB.

STRING INSTRUCTIONS

A *string* is a series of bytes or a series of words in sequential memory locations. A string often consists of ASCII character codes. In the list, a "/" is used to separate different mnemonics for the same instruction. Use the mnemonic which most clearly describes the function of the instruction in a specific application. A "B" in a mnemonic is used to specifically indicate that a string of bytes is to be acted upon. A "W" in the mnemonic is used to indicate that a string of words is to be acted upon.

REP	An instruction prefix. Repeat following instruction until CX = 0.
REPE/REPZ	An instruction prefix. Repeat instruction until CX = 0 or zero flag ZF 1.
REPNE/REPNZ	An instruction prefix. Repeat until CX = 0 or ZF = 1.
MOVS/MOVSB/MOVSW	Move byte or word from one string to another.
COMPS/COMPSB/COMPSW	Compare two string bytes or two string words.
INS/INSB/INSW	(80186/80188) Input string byte or word from port.
OUTS/OUTSB/OUTSW	(80186/80188) Output string byte or word to port.
SCAS/SCASB/SCASW	Scan a string. Compare a string byte with a byte in AL or a string word with a word in AX.

LODS/LODSB/LODSW	Load string byte into AL or string word into AX.
STOS/STOSB/STOSW	Store byte from AL or word from AX into string.

PROGRAM EXECUTION TRANSFER INSTRUCTIONS

These instructions are used to tell the 8086 to start fetching instructions from some new address, rather than continuing in sequence.

Unconditional transfer instructions:

CALL	Call a procedure (subprogram), save return address on stack.
RET	Return from procedure to calling program.
JMP	Go to specified address to get next instruction.

Conditional transfer instructions:

A "/" is used to separate two mnemonics which represent the same instruction. Use the mnemonic which most clearly describes the decision condition in a specific program. These instructions are often used after a compare instruction. The terms *below* and *above* refer to unsigned binary numbers. *Above* means larger in magnitude. The terms *greater than* or *less than* refer to signed binary numbers. *Greater than* means more positive.

JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JC	Jump if carry flag CF = 1.
JE/JZ	Jump if equal/Jump if zero flag ZF = 1.
JG/JNLE	Jump if greater/Jump if not less than or equal.
JGE/JNL	Jump if greater than or equal/ Jump if not less than.
JL/JNGE	Jump if less than/Jump if not greater than or equal.
JLE/JNG	Jump if less than or equal/Jump if not greater than.
JNC	Jump if no carry (CF = 0).
JNE/JNZ	Jump if not equal/Jump if not zero (ZF = 0).
JNO	Jump if no overflow (overflow flag OF = 0).
JNP/JPO	Jump if not parity/Jump if parity odd (PF = 0).
JNS	Jump if not sign (sign flag SF = 0).
JO	Jump if overflow flag OF = 1.
JP/JPE	Jump if parity/Jump if parity even (PF = 1).
JS	Jump if sign (SF = 1).

Iteration control instructions:

These instructions can be used to execute a series of instructions some number of times. Here mnemonics separated by a "/" represent the same instruction. Use the one that best fits the specific application.

LOOP	Loop through a sequence of instructions until CX = 0.
LOOPE/LOOPZ	Loop through a sequence of instructions while ZF = 1 and CX ≠ 0.
LOOPNE/LOOPNZ	Loop through a sequence of instructions while ZF = 0 and CX ≠ 0.
JCXZ	Jump to specified address if CX = 0.

If you aren't tired of instructions, continue skimming through the rest of the list. Don't worry if the explanation is not clear to you because we will explain these instructions in detail in later chapters.

Interrupt instructions:

INT	Interrupt program execution, call service procedure.
INTO	Interrupt program execution if OF = 1.
IRET	Return from interrupt service procedure to main program.

High-level language interface instructions:

ENTER	(80186/80188 only) Enter procedure.
LEAVE	(80186/80188 only) Leave procedure.
BOUND	(80186/80188 only) Check if effective address within specified array bounds.

PROCESSOR CONTROL INSTRUCTIONS

Flag set/clear instructions:

STC	Set carry flag CF to 1.
CLC	Clear carry flag CF to 0.
CMC	Complement the state of the carry flag CF.
STD	Set direction flag DF to 1 (decrement string pointers).
CLD	Clear direction flag DF to 0.
STI	Set interrupt enable flag to 1 (enable INTR input).
CLI	Clear interrupt enable flag to 0 (disable INTR input).

External hardware synchronization instructions:

HLT	Halt (do nothing) until interrupt or reset.
WAIT	Wait (do nothing) until signal on the TEST pin is low.
ESC	Escape to external coprocessor such as 8087 or 8089.
LOCK	An instruction prefix. Prevents another processor from taking the bus while the adjacent instruction executes.

The first line at the top of the coding form in Fig. 3.4 does not represent an instruction. It simply indicates that we want to set aside a memory location to store the result. This location must be in available RAM so that we can write to it. Address 00100H is an available RAM location on an SDK-86 prototyping board, so we chose it for this example. Next, we decide where in memory we want to start putting the code bytes for the instructions of the program. Again, on an SDK-86 prototyping board, address 00200H and above is available RAM, so we chose to start the program at address 00200H.

The first operation we want to do in the program is to initialize the data segment register. As discussed previously, two MOV instructions are used to do this. The MOV AX, 0010H instruction, when executed, will load the upper 16 bits of the address we chose for data storage into the AX register. The MOV DS, AX instruction will copy this number from the AX register to the data segment register. Now we get to the instructions that do the input, add, and store operations. The IN AL, 05H instruction will copy a data byte from the port 05H to the AL register. The ADD AL, 07 instruction will add 07H to the AL register and leave the result in the AL register. The MOV [0000], AL instruction will copy the byte in AL to a memory location at a displacement of 0000H from the data segment base. In other words, AL will be copied to a physical address computed by adding 0000 to the segment base address represented by the 0010H in the DS register. The result of this addition is a physical address of 00100H so the result in AL will be copied to physical address 00100H in memory. This is an example of the direct addressing mode described near the end of the previous chapter.

The INT 3 instruction at the end of the program functions as a *breakpoint.* When the 8086 on an SDK- 86 board executes this instruction, it will cause the 8086 to stop executing the instructions of your program and return control to the *monitor* or *system program.* You can then use *system commands* to look at the contents of registers and memory locations, or you can run another program. Without an instruction such as this at the end of the program, the 8086 would fetch and execute the code bytes for your program, then go on fetching meaningless bytes from memory and trying to execute them as if they were code bytes.

The next major section of this chapter will show you how to construct the binary codes for these and other 8086 instructions so that you can assemble and run the programs on a development board such as the SDK-86. First, however, we want to use Fig. 3.4 to make an important point about writing assembly language programs.

DOCUMENTATION

In a previous section of this chapter, we stressed the point that you should do a lot of thinking and carefully write down the algorithm for a program before you start writing instruction statements. You should also document the program itself so that its operation is clear to you and to anyone else who needs to understand it.

Each page of the program should contain the name of the program, the page number, the name of the programmer, and perhaps a version number. Each program or procedure should have a heading block containing an *abstract* describing what the program is supposed to do, which procedures it calls, which registers it uses, which ports it uses, which flags it affects, the memory used, and any other information which will make it easier for another programmer to interface with the program.

Comments should be used generously to describe the specific *function* of an instruction or group of instructions in this particular program. Comments should not be just an expansion of the instruction mnemonic. A comment of ";add 7 to AL" after the instruction ADD AL, 07H, for example, would not tell you much about the function of the instruction in a particular program. A more enlightening comment might be ";Add altitude correction factor to temperature." Incidentally, not every statement needs an individual comment. It is often more useful to write a comment which explains the function of a group of instructions.

We cannot overemphasize the importance of clear, concise documentation in your programs. Experience has shown that even a short program you wrote without comments a month ago may not be at all understandable to you now.

CONSTRUCTING THE MACHINE CODES FOR 8086 INSTRUCTIONS

This section shows you how to construct the binary codes for 8086 instructions. Most of the time you will probably use an assembler program to do this for you, but it is useful to understand how the codes are constructed. If you have an 8086-based prototyping board such as the Intel SDK-86 available, knowing how to hand code instructions will enable you to code, enter, and run simple programs.

Instruction Templates

To code the instructions for 8-bit processors such as the 8085, all you have to do is look up the hexadecimal code for each instruction on a one-page chart. For the 8086, the process is not quite as simple. Here's why. There are 32 ways to specify the source of the operand in an instruction such as MOV CX, source. The source of the operand can be any one of eight 16-bit registers, or a memory location specified by any one of 24 memory addressing modes. Each of the 32 possible instructions requires a different binary code. If CX is made the source rather than the destination, then there are 32 ways of specifying the destination. Each of these 32 possible instructions requires a different binary code. There are thus 64 different codes for MOV instructions using CX as a source or as a destination. Likewise, another 64 codes are required to specify all the possible MOVs using CL as a

source or a destination, and 64 more are required to specify all the possible MOVs using CH as a source or a destination. The point here is that, because there is such a large number of possible codes for the 8086 instructions, it is impractical to list them all in a simple table. Instead, we use a *template* for each basic instruction type and fill in bits within this template to indicate the desired addressing mode, data type, etc. In other words, we build up the instruction codes on a bit-by-bit basis.

Different Intel literature shows two slightly different formats for coding 8086 instructions. One format is shown at the end of the 8086 data sheet in Appendix A. The second format is shown along with the 8086 instruction timings in Appendix B. We will start by showing you how to use the templates shown in the 8086 data sheet.

As a first example of how to use these templates, we will build the code for the IN AL, 05H instruction from our example program. To start, look at the template for this instruction in Fig. 3.5*a*. Note that two bytes are required for the instruction. The upper 7 bits of the first byte tell the 8086 that this is an "input from a fixed port" instruction. The bit labeled "W" in the template is used to tell the 8086 whether it should input a byte to AL or a word to AX. If you want the 8086 to input a byte from an 8-bit port to AL, then make the W bit a 0. If you want the 8086 to input a word from a 16-bit port to the AX register, then make the W bit a 1. The 8-bit port address, 05H or 00000101 binary, is put in the second byte of the instruction. When the program is loaded into memory to be run, the first instruction byte will be put in one memory location, and the second instruction byte will be put in the next. Fig. 3.5*c* shows this in hexadecimal form as E4H, 05H.

To further illustrate how these templates are used, we will show here several examples with the simple MOV instruction. We will then show you how to construct the rest of the codes for the example program in Fig. 3.4. Other examples will be shown as needed in the following chapters.

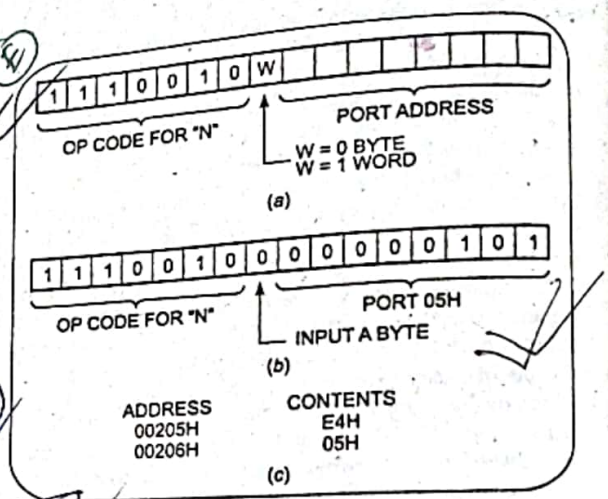

Fig. 3.5 Coding template for 8086 IN (fixed port) instruction. (a) Template (b) Example1 (c) Hex codes in sequential memory locations

MOV Instruction Coding Format and Examples

FORMAT

Figure 3.6 shows the coding template or format for 8086 instructions which MOV data from a register to a register, from a register to a memory location, or from a memory location to a register. Note that at least two code bytes are required for the instruction.

The upper 6 bits of the first byte are an opcode which indicates the general type of instruction. Look in the table in Appendix A to find the 6-bit opcode for this MOV register/memory to/from register instruction. You should find it to be 100010.

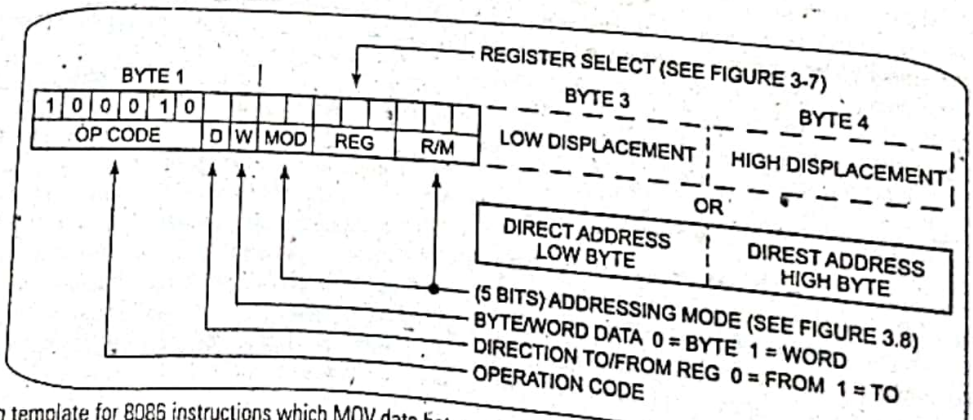

Fig. 3.6 Coding template for 8086 instructions which MOV data between registers or between a register and a memory location.

The W bit in the first word is used to indicate whether a byte or a word is being moved. If you are moving a byte, make W = 0. If you are moving a word, make W = 1.

In this instruction, one operand must always be a register, so 3 bits in the second byte are used to indicate which register is involved. The 3-bit codes for each register are shown in the table at the end of Appendix A and in Fig. 3.7. Look in one of these places to find the code for the CL register. You should get 001.

The D bit in the first byte of the instruction code is used to indicate whether the data is being moved to the register identified in the REG field of the second byte or *from* that register. If the instruction is moving data to the register identified in the REG field, make D = 1. If the instruction is moving data *from* that register, make D = 0.

Now remember that in a MOV instruction, one operand must be a register and the other operand may be a register or a memory location. The 2-bit field labeled MOD and the 3-bit field labeled R/M in the second byte of the instruction code are used to specify the desired addressing mode for the other operand. Fig. 3.8 shows the MOD and R/M bit patterns for each of the 32 possible addressing modes. Here's an overview of how you use this table.

1. If the other operand in the instruction is also one of the eight registers, then put in 11 for the MOD bits in the instruction code. In the R/M bit positions in the instruction code, put the 3-bit code for the other register.
2. If the other operand is a memory location, there are 24 ways of specifying how the execution unit should compute the effective address of the operand in memory. The effective address can be specified directly in the instruction, it can be contained in a register, or it can be the sum of one or two registers and a displacement. The MOD bits are used to indicate whether the address specification in the instruction contains a displacement. The R/M code indicates which register(s) contain part(s) of the effective address. Here's how it works:

   If the specified effective address contains no displacement, as in the instruction MOV CX, [BX] or in the instruction MOV [BX][SI], DX, then make the MOD bits 00 and choose the R/M bits which correspond to the register(s) containing the effective address. For example, if an instruction contains just [BX], the 3-bit R/M code is 111. For an instruction which contains [BX[SI]. the R/M code is 000. Note that for direct addressing, where the displacement of

REGISTER	W=1	CODE W=0
AL	AX	000
BL	BX	011
CL	CX	001
DL	DX	010
AH	SP	100
BH	DI	111
CH	BP	101
DH	SI	110

SEGREG	CODE
CS	01
DS	11
ES	00
SS	10

Fig. 3.7 Instruction codes for 8086 registers.

MOD / R/M	00	01	10	11	
				W = 0	W = 1
000	[BX] + [SI]	[BX] + [SI] + d8	[BX] + [SI] + d16	AL	AX
001	[BX] + [DI]	[BX] + [DI] + d8	[BX] + [DI] + d16	CL	CX
010	[BP] + [SI]	[BP] + [SI] + d8	[BP] + [SI] + d16	DL	DX
011	[BP] + [DI]	[BP] + [DI] + d8	[BP] + [DI] + d16	BL	BX
100	[SI]	[SI] + d8	[SI] + d16	AH	SP
101	[DI]	[DI] + d8	[DI] + d16	CH	BP
110	d16 (direct address)	[BP] + d8	[BP] + d16	DH	SI
111	[BX]	[BX] + d8	[BX] + d16	BH	DI
		MEMORY MODE		REGISTER MODE	

d8 = 8-bit displacement d16 = 16-bit displacement

Fig. 3.8 MOD and R/M bit patterns for 8086 instructions. The effective address (EA) produced by these addressing modes will be added to the data segment base to form the physical address, except for those cases where BP is used as part of the EA. In that case the EA will be added to the stack segment base to form the physical address. You can use a segment-override prefix to indicate that you want the EA to be added to some other segment base.

the operand from the segment base is specified directly in the instruction, MOD is 00 and R/M is 110. For an instruction using direct addressing, the low byte of the direct address is put in as a third instruction code byte of the instruction, and the high byte of the direct address is put in as a fourth instruction code byte.

3. If the effective address specified in the instruction contains a displacement less than 256 along with a reference to the contents of a register, as in the instruction MOV CX, 43H[BX], then code in MOD as 01 and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV CX, 43H[BX], MOD will be 01 and R/M will be 111. Put the 8-bit value of the displacement in as the third byte of the instruction.

4. If the expression for the effective address contains a displacement which is too large to fit in 8 bits, as in the instruction MOV DX, 4527H[BX], then put in 10 for MOD and choose the R/M bits which correspond to the register(s) which contain the part(s) for the effective address. For the instruction MOV DX, 4527H[BX], the R/M bits are 111. The low byte of the displacement is put in as a third byte of the instruction. The high byte of the displacement is put in as a fourth byte of the instruction. The examples which follow should help clarify all this for you.

MOV Instruction Coding Examples

All the examples in this section use the MOV instruction template in Fig. 3.6. As you read through these examples, it is a good idea to keep track of the bit-by- bit development on a separate piece of paper for practice.

CODING MOV SP, BX

This instruction will copy a word from the BX register to the SP register. Consulting the table in Appendix A, you find that the 6-bit opcode for this instruction is 100010. Because you are moving a word, W = 1. The D bit for this instruction may be somewhat confusing, however. Since two registers are involved, you can think of the move as either to SP or *from* BX. It actually does not matter which you assume as long as you are consistent in coding the rest of the instruction. If you think of the instruction as moving a word *to* SP, then make D = 1 and put 100 in the REG field to represent the SP register. The MOD field will be 11 to represent register addressing mode. Make the R/M field 011 to represent the other register, BX. The resultant code for the instruction MOV SP, BX will be 10001011 11100011. Fig. 3.9a shows the meaning of all these bits.

If you change the D bit to a 0 and swap the codes in the REG and R/M fields, you will get 10001001 11011100, which is another equally valid code for the instruction. Fig. 3.9b shows the meaning of the bits in this form. This second form, incidentally, is the form that the Intel 8086 Macroassembler produces.

CODING MOV CL, [BX]

This instruction will copy a byte to CL from the memory location whose effective address is contained in BX. The effective address will be added to the data segment base in DS to produce the physical address.

To find the 6-bit opcode for byte 1 of the instruction, consult the table in Appendix A. You should find that this code is 100010. Make D = 1 because data is being moved to register CL. Make W = 0 because the instruction is moving a byte into CL. Next you need to put the 3-bit code which represents register CL in the REG field of the second byte of the instruction code. The codes for each register are shown in Fig. 3.7. In this figure you should find that the code for CL is 001. Now, all you need to determine is the bit patterns for the MOD and R/M fields. Again use the table in Fig. 3.8 to do this. In the table, first find the box containing the desired addressing mode. The box containing [BX], for example, is in the lower left corner of the table. Read the required MOD-bit pattern from the top of the column. In this case, MOD is 00. Then read the required R/M- bit pattern at the left of the box. For this instruction you should find R/M to be 111. Assembling all these bits together should give you 10001010 00001111 as the binary code for the instruction MOV CL, [BX]. Fig. 3.10 summarizes the meaning of all the bits in this result.

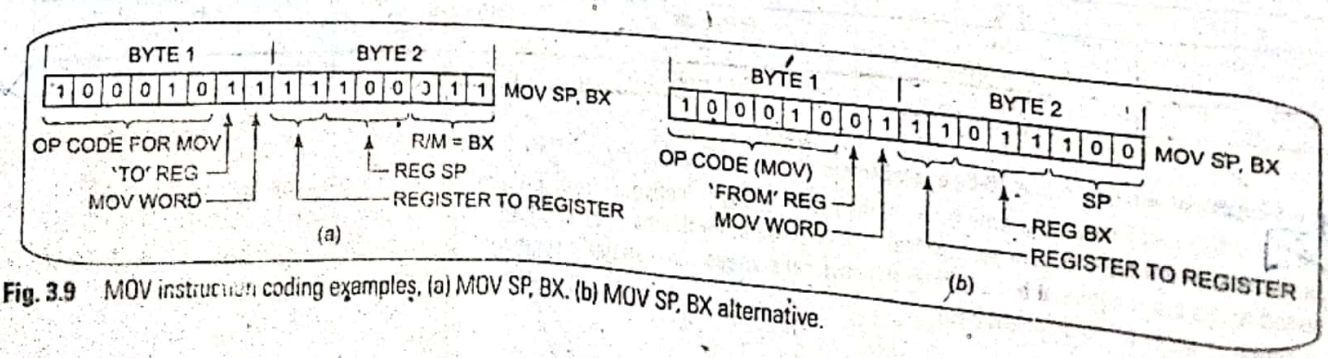

Fig. 3.9 MOV instruction coding examples, (a) MOV SP, BX. (b) MOV SP, BX alternative.

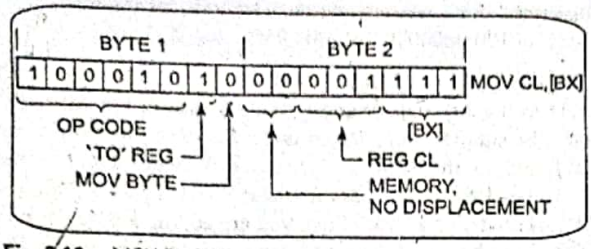

Fig. 3.10 MOV CL, [BX].

CODING MOV 43H[SI], DH

This instruction will copy a byte from the DH register to a memory location. The BIU will compute the effective address of the memory location by adding the indicated displacement of 43H to the contents of the SI register. As we showed you in the last chapter, the BIU then produces the actual physical address by adding this effective address to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. Put 110 in the REG field to represent the DH register. D = 0 because you are moving data *from* the DH register. W = 0 because you are moving a byte. The R/M field will be 100 because SI contains part of the effective address. The MOD field will be 01 because the displacement contained in the instruction, 43H, will fit in 1 byte. If the specified displacement had been a number larger than FFH, then MOD would be 10. Putting all these pieces together gives 10001000 01110100 for the first two bytes of the instruction code. The specified displacement, 43H or 01000011 binary, is put after these two as a third instruction byte. Fig. 3.11 shows this. If an instruction specifies a 16-bit displacement, then the low byte of the displacement is put in as byte 3 of the instruction code, and the high byte of the displacement is put in as byte 4 of the instruction code.

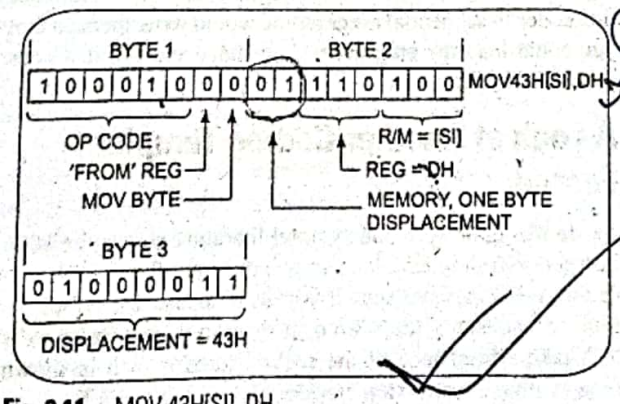

Fig. 3.11 MOV 43H[SI], DH.

CODING MOV CX, [437AH]

This instruction copies the contents of two memory locations into the CX register. The direct address or displacement of the first memory location from the start of the data segment is 437AH. As we showed you in the last chapter, the BIU will produce the physical memory address by adding this displacement to the data segment base represented by the 16-bit number in the DS register.

The 6-bit opcode for this instruction is again 100010. Make D = 1 because you are moving data to the CX register, and make W = 1 because the data being moved is a word. Put 001 in the REG field to represent the CX register, then consult Fig. 3.8 to find the MOD and R/M codes. In the first column of the figure you should find a box labeled "direct address," which is the name given to the addressing mode used in this instruction. For direct addressing, you should find MOD to be 00 and R/M to be 110. The first two code bytes for the instruction, then, are 10001011 00001110. These two bytes will be followed by the low byte of the direct address, 7AH (01111010 binary), and the high byte of the direct address, 43H (01000011 binary). The instruction will be coded into four successive memory addresses as 8BH, 0EH, 7AH, and 43H. Fig. 3.12 spells this out in detail.

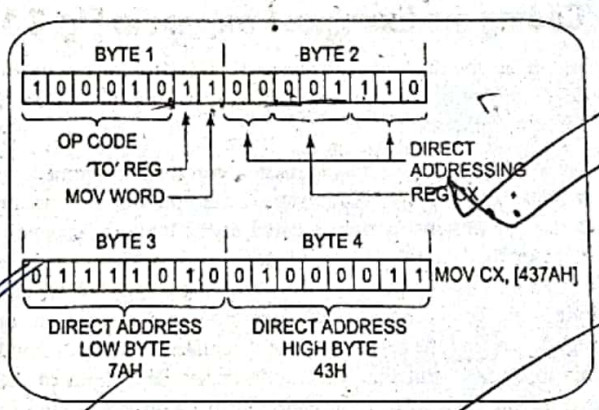

Fig. 3.12 MOV CX, [437AH].

CODING MOV CS:[BX], DL

This instruction copies a byte from the DL register to a memory location. The effective address for the memory location is contained in the BX register. Normally an effective address in BX will be added to the data segment base in DS to produce the physical memory address. In this instruction, the CS: in front of [BX] indicates that we want the BIU to add the effective address to the code segment base in CS to produce the physical address. The CS: is called a *segment override prefix*.

When an instruction containing a segment override prefix is coded, an 8-bit code for the segment override prefix is put in memory *before* the code for the rest of the instruction. The code byte for the segment override prefix has the format 001XX110. You insert a 2-bit code in place of the X's to indicate which segment base you want the effective address to be added to. As shown in Fig. 3.7, the codes for these 2 bits are as follows: ES = 00, CS = 01, SS = 10, and DS = 11.

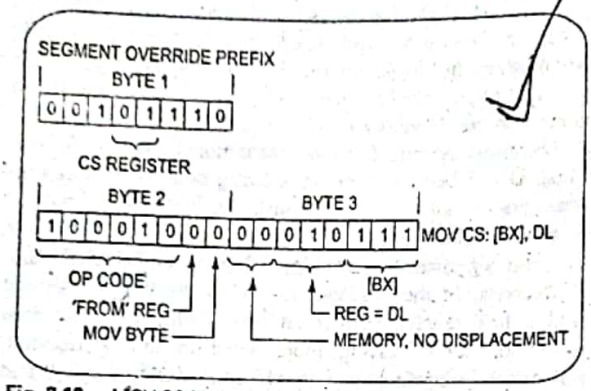

Fig. 3.13 MOV CS:[BX], DL.

The segment override prefix byte for CS, then, is 00101110. For practice, code out the rest of this instruction. Fig. 3.13 shows the result you should get and how the code for the segment override prefix is put before the other code bytes for the instruction.

Coding the Example Program in Fig. 3.4

Again, as you read through this section, follow the bit- by-bit development of the instruction codes on a separate piece of paper for practice.

MOV AX, 0010H This instruction will load the immediate word 0010H into the AX register. The simplest code template to use for this instruction is listed in the table in Appendix A under the "MOV — Immediate to register" heading. The format for this instruction is 1011 W REG, data byte low, data byte high. W = 1 because you are moving a word. Consult Fig. 3.7 to find the code for the AX register. You should find this to be 000. Put this 3-bit code in the REG field of the instruction code. The completed instruction code byte is 10111000. Put the low byte of the immediate number, 10H, in as the second code byte. Then put the high byte of the immediate data, 00H, in as the third code byte. The resultant sequence of code bytes, then, will be B8H, 10H, 00H.

MOV DS, AX This instruction copies the contents of the AX register into the data segment register. The template to use for coding this instruction is found in the table in Appendix A under the heading "MOV — Register/memory to segment register." The format for this template is 10001110 MOD 0 segreg R/M. Segreg represents the 2-bit code for the desired segment register, as shown in Fig. 3.7. These codes are also found in the table at the end of Appendix A. The segreg code for the DS register is 11. Since the other operand is a register, MOD should be 11. Put the 3-bit code for the AX register, 000, in the R/M field. The resultant codes for the two code bytes should then be 10001110 11011000, or 8EH D8H.

IN AL, 05H This instruction copies a byte of data from port 05H to the AL register. The coding for this instruction was described in a previous section. The code for the instruction is 11100100 00000101 or E4H 05H.

ADD AL, 07H This instruction adds the immediate number 07H to the AL register and puts the result in the AL register. The simplest template to use for coding this instruction is found in the table in Appendix A under the heading "ADD — Immediate to accumulator." The format is 0000010 W, data byte, data byte. Since you are adding a byte, W = 0. The immediate data byte you are adding will be put in the second code byte. The third code byte will not be needed because you are adding only a byte. The resultant codes, then, are 00000100 00000111 or 04H 07H.

MOV [0000], AL This instruction copies the contents of the AL register to a memory location. The direct address or displacement of the memory location from the start of the data segment is 0000H. The code template for this instruction is found in the table in Appendix A under the heading "MOV — Accumulator to memory." The format for the instruction is 1010001 W, address low byte, address high byte. Since the instruction moves a byte, W = 0. The low byte of the direct address is written in as the second instruction code byte, and the high byte of the direct address is written in as the third instruction code byte. The codes for these 3 bytes, then, will be 10100010 00000000 00000000 or A2H 00H 00H.

INT 3 In some 8086 systems this instruction causes the 8086 to stop executing your program instructions, return to the monitor program, and wait for your next command. According to the format table in Appendix A, the code for a type 3 interrupt is the single byte 11001100 or CCH.

SUMMARY OF HAND CODING THE EXAMPLE PROGRAM

Figure 3.4 shows the example program with all the instruction codes in sequential order as you would write them so that you could load the program into memory and run it. Codes are in HEX to save space.

A Look at Another Coding Template Format

As we mentioned previously, Intel literature shows the 8086 instruction coding templates in two different forms. The preceding sections have shown you how to use the templates found at the end of the 8086 data sheet in Appendix A. Now let's take a brief look at the second form, which is shown along with the instruction clock cycles in Appendix B.

The only difference between the second form for the templates and the form we discussed previously is that the D and W bits are not individually identified. Instead, the complete opcode bytes are shown for each version of an in... For example, in Appendix B, the ... memory 8 ...

Implementing Standard Program Structures in 8086 Assembly Language

4

Objectives

At the conclusion of this chapter, you should be able to:

1. *Write flowcharts or pseudocode for simple programming problems.*
2. *Implement SEQUENCE, IF-THEN-ELSE, WHILE-DO, and REPEAT-UNTIL program structures in 8086 assembly language.*
3. *Describe the operation of selected data transfer, arithmetic, logical, jump, and loop instructions.*
4. *Use based and indexed addressing modes to access data in your programs.*
5. *Describe a systematic approach to debugging a simple assembly language program using debugger, monitor, or emulator tools.*
6. *Write a delay loop which produces a desired amount of delay on a specific 8086 system.*

In Chapter 3 we worked very hard to convince you that you should not try to write programs directly in assembly language. The analogy of building a house without a plan should come to mind here. When faced with a programming problem, you should solve the problem and write the algorithm for the solution using the standard program structures we described. Then you simply translate each step in the flowchart or pseudocode to a group of one to four assembly language instructions which will implement that step. The comments in the assembly language program should describe the functions of each instruction or group of instructions, so you essentially write the comments for the program, then write the assembly language instructions which implement those comments. Once you learn how to implement each of the standard programming structures, you should find it quite easy to translate algorithms to assembly language. Also, as we will show you, the standard structure approach makes debugging relatively easy.

The purposes of this chapter are to show you how to write the algorithms for some common programming problems, how to implement these algorithms in 8086 assembly language, and how to systematically debug assembly language programs. In the process you will also learn more about how some of the 8086 instructions work.

SIMPLE SEQUENCE PROGRAMS

Finding the Average of Two Numbers

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

A common need in programming is to find the average of two numbers. Suppose, for example, we know the maximum temperature and the minimum temperature for a given day, and we want to determine the average temperature. The sequence of steps we go through to do this might look something like the following.

Add maximum temperature and minimum temperature.
Divide sum by 2 to get average temperature.

This sequence doesn't look much like an assembly language program, and it shouldn't. The algorithm at this point should be general enough that it could be implemented in any programming language, or on any machine. Once you are reasonably sure of your algorithm, then you can start thinking about the architecture and instructions of the specific microcomputer on which you plan to run the program. Now let's show you how we get from the algorithm to the assembly language program for it.

SETTING UP THE DATA STRUCTURE

One of the first things for you to think about in this process is the data that the program will be working with. You need to ask yourself questions such as:

```
1                                   ; 8086 PROGRAM F4-05.ASM
2                           ;ABSTRACT  : Program produces a packed BCD byte from 2 ASCII-encoded digits
3                                      ; the first ASCII digit (5) is loaded in BL.
4                                      ; the second ASCII digit (9) is loaded in AL.
5                                      ; the result (packed BCD) is left in AL
6                           ;REGISTERS ; Uses CS, AL, BL, CL
7                           ;PORTS     : None used
8
9 0000                      CODE    SEGMENT
10                                  ASSUME CS:CODE
11 0000  B3 35              START:  MOV  BL, '5'   ; Load first ASCII digit into BL
12 0002  B0 39                      MOV  AL, '9'   ; Load second ASCII digit into AL
13 0004  80 E3 0F                   AND  BL, 0FH   ; Mask upper 4 bits of first digit
14 0007  24 0F                      AND  AL, 0FH   ; Mask upper 4 bits of second digit
15 0009  B1 04                      MOV  CL, 04H   ; Load CL for 4 rotates required
16 000B  D2 C3                      ROL  BL, CL    ; Rotate BL 4 bit positions
17 000D  0A C3                      OR   AL, BL    ; Combine nibbles, result in AL
18 000F                     CODE    ENDS
19                                  END START
```

Fig. 4.5 List file of 8086 assembly language program to produce packed BCD form two ASCII characters.

Note that for the 80186 you can write the single instruction ROL BL, 04H to do this job.

Now that we have determined the instructions needed to mask the upper nibbles and the instructions needed to move the first BCD digit into position, the only thing left is to pack the upper nibble from BL and the lower nibble from AL into a single byte.

COMBINING BYTES OR WORDS WITH THE ADD OR THE OR INSTRUCTION

You can't use a standard MOV instruction to combine two bytes into one as we need to do here. The reason is that the MOV instruction copies an operand from a specified source to a specified destination. The previous contents of the destination are lost. You can, however, use an ADD or an OR instruction to pack the two BCD nibbles.

As described in the previous program example, the ADD instruction adds the contents of a specified source to the contents of a specified destination and leaves the result in the specified destination. For the example program here, the instruction ADD AL, BL can be used to combine the two BCD nibbles. Take a look at Fig. 4.2 to help you visualize this addition.

Another way to combine the two nibbles is with the OR instruction. If you look up the OR instruction in Chapter 6, you will find that it has the format OR destination, source. This instruction ORs each bit in the specified source with the corresponding bit in the specified destination. The result of the ORing is left in the specified destination. ORing a bit with a 1 always produces a result of 1. ORing a bit with a 0 leaves the bit unchanged. To set a bit in a word to a 1, then, all you have to do is OR that bit with a word which has a 1 in that bit position and 0's in all the other bit positions. This is similar to the way the AND instruction is used to clear bits in a word to 0's. See the OR instruction description in Chapter 6 for examples of this.

For the example program here, we use the instruction OR AL,BL to pack the two BCD nibbles. Bits ORed with 0s will not be changed. Bits ORed with 0s will become or stay 1's. Again look at Fig. 4.2 to help you visualize this operation.

SUMMARY OF BCD PACKING PROGRAM

If you compare the algorithm for this program with the finished program in Fig. 4.5, you should see that each step in the algorithm translates to one or two assembly language instructions. As we told you before, developing the assembly language program from a good algorithm is really quite easy because you are simply translating one step at a time to its equivalent assembly language instructions. Also, debugging a program developed in this way is quite easy because you simply single-step or breakpoint your way through it and check the results after each step. In the next section we discuss the 8086 JMP instructions and flags so we can show you how you implement some of the other programming structures in assembly language.

JUMPS, FLAGS, AND CONDITIONAL JUMPS

Introduction

The real power of a computer comes from its ability to choose between two or more sequences of actions based on some condition, repeat a sequence of instructions *as long as*

some condition exists, or repeat a sequence of instructions *until* some condition exists. *Flags* indicate whether some condition is present or not. *Jump* instructions are used to tell the computer the address to fetch its next instruction from. Figure. 4.6 shows in diagram form the different ways a Jump instruction can direct the 8086 to fetch its next instruction from some place in memory other than the next sequential location.

The 8086 has two types of Jump instructions, conditional, and unconditional. When the 8086 fetches and decodes an Unconditional Jump instruction, it always goes to the specified jump destination. You might use this type of Jump instruction at the end of a program so that the entire program runs over and over, as shown in Fig. 4.6.

When the 8086 fetches and decodes a Conditional Jump instruction, it evaluates the state of a specified flag to determine whether to fetch its next instruction from the jump destination location or to fetch its next instruction from the next sequential memory location.

Let's start by taking a look at how the 8086 Unconditional Jump instruction works.

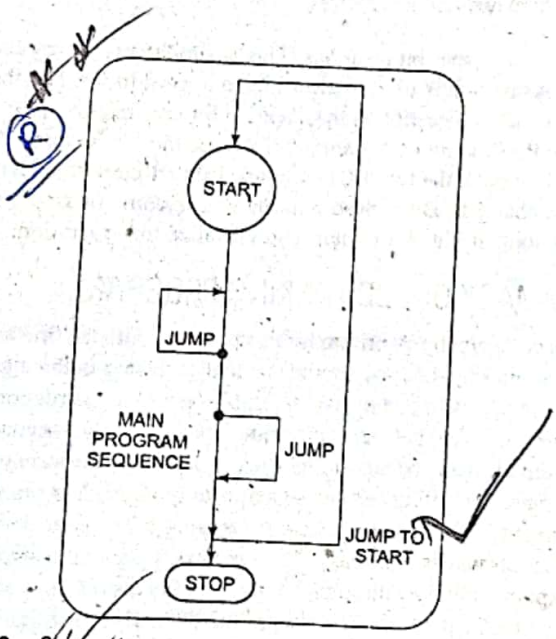

Fig. 4.6 Change in program flow that can be caused by jump instructions

The 8086 Unconditional Jump Instruction

INTRODUCTION

As we said before, Jump instructions can be used to tell the 8086 to start fetching its instructions from some new location rather than from the next sequential location. The 8086 JMP instruction always causes a jump to occur, so this is referred to as an *unconditional* jump.

Remember from previous discussions that the 8086 computes the physical address from which to fetch its next code byte by adding the offset in the instruction pointer register to the code segment base represented by the 16-bit number in the CS register. When the 8086 executes a JMP instruction, it loads a new number into the instruction pointer register, and in some cases it also loads a new number into the code segment register.

If the JMP destination is in the same code segment, the 8086 only has to change the contents of the instruction pointer. This type of jump is referred to as a *near*, or *intrasegment*, jump.

If the JMP destination is in a code segment which has a different name from the segment in which the JMP instruction is located, the 8086 has to change the contents of both CS and IP to make the jump. This type of jump is referred to as *a far*, or *intersegment*, jump.

Near and far jumps are further described as either *direct* or *indirect*. If the destination address for the jump is specified directly as part of the instruction, then the jump is described as *direct*. You can have a direct near jump or a direct far jump. If the destination address for the jump is contained in a register or memory location, the jump is referred to as *indirect*, because the 8086 has to go to the specified register or memory location to get the required destination address. You can have an indirect near jump or an indirect far jump.

Figure 4.7 shows the coding templates for the four basic types of unconditional jumps. As you can see, for the direct types, the destination offset, and, if necessary, the segment base are included directly in the instruction. The indirect types of jumps use the second byte of the instruction to tell the 8086 whether the destination offset (and segment base, if necessary) is contained in a register or in memory locations specified with one of the 24 address modes we introduced you to in the last chapter.

The JMP instruction description in Chapter 6 shows examples of each type of jump instruction, but in most of your programs you will use a direct near-type JMP instruction, so in the next section we will discuss in detail how this type works.

UNCONDITIONAL JUMP INSTRUCTION TYPES—OVERVIEW

The 8086 Unconditional Jump instruction, JMP, has five different types. Figure 4.7 shows the names and instruction coding templates for these five types. We will first summarize how these five types work to give you an overview; then we will describe in detail the two types you need for your programs at this point. The JMP instruction description in Chapter 6 shows examples of each of the five types.

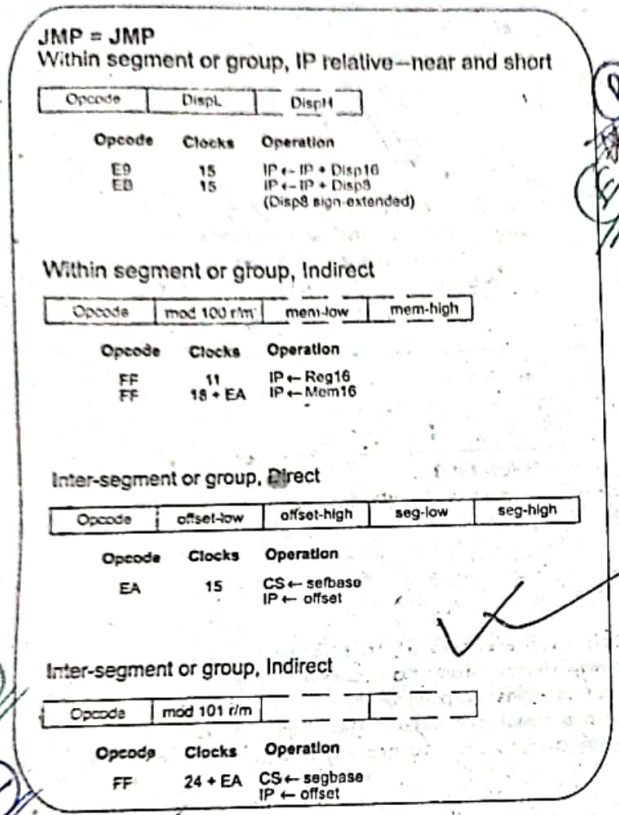

Fig. 4.7 8086 Unconditional Jump instructions. (Intel Corporation)

THE DIRECT NEAR- AND SHORT-TYPE JMP INSTRUCTIONS

As we described previously, a near-type jump instruction can cause the next instruction to be fetched from anywhere in the current code segment. To produce the new instruction fetch address, this instruction adds a 16-bit signed displacement contained in the instruction to the contents of the instruction pointer register. A 16-bit signed displacement means that the jump can be to a location anywhere from +32,767 to –32,768 bytes from the current instruction pointer location. A positive displacement usually means you are jumping ahead in the program, and a negative displacement usually means that you are jumping "backward" in the program.

A special case of the direct near-type jump instruction is the direct short-type jump. If the destination for the jump is within a displacement range of +127 to –128 bytes from the current instruction pointer location, the destination can be reached with just an 8-bit displacement. The coding for this type of jump is shown on the second line of the coding template for the direct near JMP in Fig. 4.7. Only one byte is required for the displacement in this case. Again the 8086 produces the new instruction fetch address by adding the signed 8-bit displacement, contained in the instruction, to the contents of the instruction pointer register. Here are some examples of how you use these JMP instructions in programs.

DIRECT WITHIN-SEGMENT NEAR AND DIRECT WITHIN-SEGMENT SHORT JMP EXAMPLES

Suppose that we want an 8086 to execute the instructions in a program over and over. Figure 4.8 shows how the JMP instruction can be used to do this. In this program, the label BACK followed by a colon is used to give a name to the address we want to jump back to. When the assembler reads this label, it will make an entry in its symbol table indicating where it found the label. Then, when the assembler reads the JMP instruction and finds the name BACK in the instruction, it will be able to calculate the displacement from the jump instruction to the label. This displacement will be inserted as part of the code for the instruction. Even if you are not using an assembler, you should use labels to indicate jump destinations so that you can easily see them. The NOP instructions used in the program in Fig. 4.8 do nothing except fill space. We used them in this example to represent the instructions that we want to loop through over and over. Once the 8086 gets into the JMP-BACK loop, the only ways it can get out are if the power is turned off, an interrupt occurs, or the system is reset.

Now let's see how the binary code for the JMP instruc-tion in Fig. 4.8 is constructed. The jump is to a label in the same segment, so this narrows our choices down to the first three types of JMP instruction shown in Fig. 4.7. For several reasons, it is best to use the direct-type JMP instruction whenever possible. This narrows our choices down to the first two types in Fig. 4.7. The choice between these two is determined by whether you need a 1-byte or a 2-byte displacement to reach the JMP destination address. Since for our example program the destination address is within the range of –128 to +127 bytes from the instruction after the JMP instruction, we can use the direct within-segment short type of JMP. According to Fig. 4.7, the instruction template for this instruction is 11101011 (EBH) followed by a displacement. Here's how you calculate the displacement to put in the instruction.

NOTE: An assembler does this for you automatically, but you should still learn how it is done to help you in troubleshooting.

The numbers in the left column of Fig. 4.8 represent the offset of each code byte from the code segment base. These are the numbers that will be in the instruction pointer as the program executes. After the 8086 fetches an instruction byte, it automatically increments the instruction pointer to point to the next instruction byte. The displacement in the JMP instruction will then be added to the offset of the next in-line instruction after the JMP instruction. For the example program in Fig. 4.8, the displacement in the JMP instruction will be added to offset 0006H, which is in the instruction pointer

```
1                                ; 8086 PROGRAM    F4-08.ASM
2                          ;ABSTRACT  : This program illustrates a "backwards" jump
3                          ;REGISTERS : Uses CS, AL
4                          ;PORTS     : None used
5
6 0000                     CODE   SEGMENT
7                                 ASSUME  CS:CODE
8 0000  04 03              BACK:  ADD AL, 03H  ; Add 3 to total
9 0002  90                        NOP          ; Dummy instructions to represent those
10 0003 90                        NOP          ; instructions jumped back over
11 0004 EB FA                     JMP BACK     ; Jump back over instructions to BACK label
12 0006                    CODE   ENDS
13                                END
```

Fig. 4.8 List file of program demonstrating "backward" JMP.

```
1                                ; 8086 PROGRAM     F4-09.ASM
2                          ;ABSTRACT  : This program illustrates a "forwards" jump
3                          ;REGISTERS : Uses CS, AX
4                          ;PORTS     : None used
5
6 0000                     CODE   SEGMENT
7                                 ASSUME CS:CODE
8 0000  EB 03 90                  JMP THERE     ; Skip over a series of instructions
9 0003  90                        NOP           ; Dummy instructions to represent those
10 0004 90                        NOP           ; Instructions skipped over
11 0005 B8 0000            THERE: MOV AX, 0000H ; Zero accumulator before addition instructions
12 0008 90                        NOP           ; Dummy instruction to represent continuation of execution
13 0009                    CODE   ENDS
14                                END
```

Fig. 4.9 List file of program demonstrating "forward" JMP.

after the JMP instruction executes. What this means is that when you are counting the number of bytes of displacement, you always start counting from the address of the instruction immediately after the JMP instruction. For the example program, we want to jump from offset 0006H back to offset 0000H. This is a displacement of –6H.

You can't, however, write the displacement in the instruction as –6H. Negative displacements must be expressed in 2's complement, sign-and-magnitude form. First, write the number as an 8-bit positive binary number. In this case, that is 00000110. Then, invert each bit of this, including the sign bit, to give 11111001. Finally, add 1 to that result to give 11111010 binary or FAH, which is the correct 2's complement representation for –6H. As shown on line 11 in the assembler listing for the program in Fig. 4.8, the two code bytes for this JMP instruction then are EBH and FAH.

To summarize this example, then, a label is used to give a name to the destination address for the jump. This name is used to refer to the destination address in the JMP instruction. Since the destination in this example is within the range of –128 to +127 bytes from the address after the JMP instruction, the instruction can be coded as a direct within-segment short-type JMP. The displacement is calculated by counting the number of bytes from the next address after the JMP instruction to the destination. If the displacement is negative (backward in the program), then it must be expressed in 2's complement form before it can be written in the instruction code template.

Now let's look at another simple example program, in Fig. 4.9, to see how you can jump ahead over a group of instructions in a program. Here again we use a label to give a name to the address that we want to JMP to. We also use NOP instructions to represent the instructions that we want to skip over and the instructions that continue after the JMP. Let's see how this JMP instruction is coded.

When the assembler reads through the source file for this program, it will find the label "THERE" after the JMP mnemonic. At this point the assembler has no way of knowing whether it will need 1 or 2 bytes to represent the displacement to the destination address. The assembler plays it safe by reserving 2 bytes for the displacement. Then the assembler reads on through the rest of the program. When the assembler finds the specified label, it calculates the displacement from the instruction after the JMP instruction to the label.

THE AUXILIARY CARRY FLAG

This flag has significance in BCD addition or BCD subtraction. If a carry is produced when the least significant nibbles of 2 bytes are added, the auxiliary carry flag will be set. In other words, a carry out of bit 3 sets the auxiliary carry flag. Likewise, if the subtraction of the least significant nibbles requires a borrow, the auxiliary carry/borrow flag will be set. The auxiliary carry/borrow flag is used only by the DAA and DAS instructions. Consult the DAA and DAS instruction descriptions in Chapter 6 for further discussion of addition and subtraction of BCD numbers.

THE ZERO FLAG WITH INCREMENT, DECREMENT, AND COMPARE INSTRUCTIONS

As the name implies, this flag will be set to a 1 if the result of an arithmetic or logic operation is zero. For example, if you subtract two numbers which are equal, the zero flag will be set to indicate that the result of the subtraction is zero. If you AND two words together and the result contains no 1's, the zero flag will be set to indicate that the result is all 0's.

Besides the more obvious arithmetic and logic instructions, there are a few other very useful instructions which also affect the zero flag. One of these is the compare instruction CMP, which we discussed previously with the carry flag. As shown there, the zero flag will be set to a 1 if the two operands compared are equal,

Another important instruction which affects the zero flag is the decrement instruction, DEC. This instruction will decrement (or, in other words, subtract 1 from) a number in a specified register or memory location. If, after decrementing, the contents of the register or memory location are zero, the zero flag will be set. Here's a preview of how this is used. Suppose that we want to repeat a sequence of actions nine times. To do this, we first load a register with the number 09H and execute the sequence of actions. We then decrement the register and look at the zero flag to see if the register is down to zero yet. If the zero flag is not set, then we know that the register is not yet down to zero, so we tell the 8086, with a Jump instruction, to go back and execute the sequence of instructions again. The following sections will show many specific examples of how this is done.

The increment instruction, INC destination, also affects the zero flag. If an 8-bit destination containing FFH or a 16-bit destination containing FFFFH is incremented, the result in the destination will be all 0's. The zero flag will be set to indicate this.

THE SIGN FLAG—POSITIVE AND NEGATIVE NUMBERS

When you need to represent both positive and negative numbers for an 8086, you use 2's complement sign-and-magnitude form. In this form, the most significant bit of the byte or word is used as a sign bit. A 0 in this bit indicates that the number is positive. A 1 in this bit indicates that the number is negative. The remaining 7 bits of a byte or the remaining 15 bits of a word are used to represent the magnitude of the number. For a positive number, the magnitude will be in standard binary form. For a negative number, the magnitude will be in 2's complement form. After an arithmetic or logic instruction executes, the sign flag will be a copy of the most significant bit of the destination byte or the destination word. In addition to its use with signed arithmetic operations, the sign flag can be used to determine whether an operand has been decremented beyond zero. Decrementing 00H, for example, will give FFH. Since the MSB of FFH is a 1, the sign flag will be set.

THE OVERFLOW FLAG

This flag will be set if the result of a signed operation is too large to fit in the number of bits available to represent it. To remind you of what *overflow* means, here is an example. Suppose you add the 8-bit signed number 01110101 (+117 decimal) and the 8-bit signed number 00110111 (+55 decimal). The result will be 10101100 (+172 decimal), which is the correct binary result in this case, but is too large to fit in the 7 bits allowed for the magnitude in an 8-bit signed number. For an 8-bit signed number, a 1 in the most significant bit indicates a negative number. The overflow flag will be set after this operation to indicate that the result of the addition has overflowed into the sign bit.

The 8086 Conditional Jump Instructions

As we stated previously, much of the real power of a computer comes from its ability to choose between two courses of action depending on whether some condition is present or not. In the 8086 the six conditional flags indicate the conditions that are present after an instruction. The 8086 Conditional Jump instructions look at the state of a specified flag(s) to determine whether the jump should be made or not.

Figure 4.10 shows the mnemonics for the 8086 Conditional Jump instructions. Next to each mnemonic is a brief explanation of the mnemonic. Note that the terms *above* and *below* are used when you are working with unsigned binary numbers. The 8-bit unsigned number 11000110 is above the 8-bit unsigned number 00111001, for example. The terms *greater* and *less* are used when you are working with signed binary numbers. The 8-bit signed number 00111001 is greater (more positive) than the 8-bit signed number 11000110, which represents a negative number. Also shown in Fig. 4.10 is an indication of the flag conditions that will cause the 8086 to do the jump. If the specified flag conditions are not present, the 8086 will just continue on to the next instruction in sequence. In other words, if the jump condition is not met, the Conditional Jump instruction will effectively function as a NOP. Suppose, for example, we have the instruction

MNEMONIC	CONDITION TESTED	"JUMP IF . . ."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CFor ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) =1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Note: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

Fig. 4.10 8086 Conditional Jump instructions.

JC SAVE, where SAVE is the label at the destination address. If the carry flag is set, this instruction will cause the 8086 to jump to the instruction at the SAVE: label. If the carry flag is not set, the instruction will have no effect other than taking up a little processor time.

All conditional jumps are *short-type* jumps. This means that the destination label must be in the same code segment as the jump instruction. Also, the destination address must be in the range of −128 bytes to +127 bytes from the address of the instruction after the Jump instruction. As we show in later examples, it is important to be aware of this limit on the range of conditional jumps as you write your programs.

The Conditional Jump instructions are usually used after arithmetic or logic instructions. They are very commonly used after Compare instructions. For this case, the Compare instruction syntax and the Conditional Jump instruction syntax are such that a little trick makes it very easy to see what will cause a jump to occur. Here's the trick. Suppose that you see the instruction sequence.

```
CMP BL, DH
JAE HEATER_OFF
```

in a program, and you want to determine what these instructions do. The CMP instruction compares the byte in the DH register with the byte in the BL register and sets flags according to the result. A previous section showed you how the carry and zero flags are affected by a Compare instruction. According to Fig. 4.10, the JAE instruction says, "Jump if above or equal" to the label HEATER_OFF. The question now is, will it jump if BL is above DH, or will it jump if DH is above BL? You could determine how the flags will be affected by the comparison and use Fig. 4.10 to answer the question, but an easier way is to mentally read parts of the Compare instruction between parts of the Jump instruction. If you read the example sequence as "Jump if BL is above or equal to DH," the meaning of the sequence is immediately clear. As you write your own programs, thinking of a conditional sequence in this way should help you to choose the right Conditional Jump instruction. The next sections show you how we use Conditional and Unconditional Jump instructions to implement some of the standard program structures and solve some common programming problems.

IF-THEN, IF-THEN-ELSE, AND MULTIPLE IF-THEN-ELSE PROGRAMS

IF-THEN Programs

The IF-THEN structure has the format

```
IF condition THEN
    action
    action
```

This structure says that IF the stated condition is found to be true, the series of actions following THEN will be executed. If the condition is false, execution will skip over the actions after the THEN and proceed with the next mainline instruction.

The simple IF-THEN is implemented with a Conditional Jump instruction. In some cases an instruction to set flags is needed before the Conditional Jump instruction. Figure 4.11*a* shows, with a program fragment, one way to implement the simple IF-THEN structure. In this program we first compare BX with AX to set the required flags. If the zero flag is set after the comparison, indicating that AX = BX, the JE instruction will cause execution to jump to the MOV CL, 07H instruction labeled THERE. If AX ≠ BX, then the ADD AX, 0002H instruction after the JE instruction will be executed before the MOV CL,07H instruction.

The implementation in Fig. 4.11*a* will work well for a short sequence of instructions after the Conditional Jump instruction. However, if the sequence of instructions is lengthy, there is a potential problem. Remember from the discussion of conditional jumps in the last section that a conditional jump can only be to a location in the range of −128 bytes to +127 bytes from the address after the Conditional Jump instruction. A long sequence of instructions after the Conditional Jump instruction may put the label out of range of the instruction. If you are absolutely sure that the destination label will not be out of range, then use the instruction sequence shown in Fig. 4.11*a* to implement an IF-THEN structure. If you are not sure whether the destination will be in range, the instruction sequence shown in Fig. 4.11*b* will always work. In this sequence, the Conditional Jump instruction only has to jump over the JMP instruction. The JMP instruction used to get to the label THERE can jump to anywhere in the code segment, or even to another code segment. Note that you have to change the Conditional Jump instruction from JE to JNE for this second version. The price you pay for not having to worry whether the destination is in range is an extra jump instruction. Incidentally, some assemblers now automatically code Conditional Jump instructions in this way if necessary.

```
        CMP AX, BX     ; Compare to set_flags
        JE  THERE      ; If equal then skip correction
        ADD AX, 0002H  ; Add correction factor
THERE:  MOV CL, 07H    ; Load count
```

(*a*)

```
        CMP AX, BX     ; Compare to set flags
        JNE FIX        ; If not equal do correction
        JMP THERE      ; If equal then skip correction
FIX:    ADD AX, 0002H  ; Add correction factor

THERE:  MOV CL, 07H    ; Load count
```

(*b*)

Fig. 4.11 Programming conditional jumps. (*a*) Destinations closer than ±128 bytes. (*b*) Destinations further than ±128 bytes.

IF-THEN-ELSE Programs

OVERVIEW

The IF-THEN-ELSE structure is used to indicate a choice between two alternative courses of action. Fig. 3.3*b* shows the flowchart and pseudocode for this structure. Basically the structure has the format

```
IF condition THEN
    action
ELSE
    action
```

This is a different situation from the simple IF-THEN, because here either one series of actions or another series of actions is done before the program goes on with the next mainline instruction. An example will show how we implement this structure.

Suppose that we have an 8086 microcomputer which controls a printed-circuit-board-making machine. Part of the job of this 8086 is to check a temperature sensor and turn on a green lamp or a yellow lamp depending on the value of the temperature it reads in. If the temperature is below 30°C, we want to turn on a yellow lamp to tell the operator that the solution is not up to temperature. If the temperature is greater than or equal to 30°C, we want to light a green lamp. With a system such as this, the operator can visually scan all the lamps on the control panel until all the green lamps are lit. When all the lamps are green, the operator can push the GO button to start making boards. The reason that we have the yellow lamp is to let the operator know that this part of the machine is working, but that the temperature is not yet up to 30°C.

Figure 4.12 shows with flowcharts and with pseudocode two ways we can represent the algorithm for this problem. The difference between the two is simply a matter of whether we make the decision based on the temperature being below 30°C or based on the temperature being above or equal to 30°C. The two approaches are equally valid, but your choice determines which Conditional Jump instruction you use to implement the algorithm. Since this program involves reading data in from a port and writing data out to a port, we need to talk briefly about the 8086 IN and OUT instructions before we discuss the details of how these two algorithms can be implemented in assembly language.

THE 8086 IN AND OUT INSTRUCTIONS

The 8086 has two types of input instruction, fixed-*port* and *variable-port*. The fixed-port instruction has the format IN AL,port. The term *port* in these instructions represents an 8-bit port address to be put directly in the instruction. The instruction IN AX,04H, for example, will copy a word from port 04H to the AX register. The 8-bit port

Semiconductor Memories and Interfacing

8

Objectives

At the conclusion of this chapter you should be able to:

1. *Understand types of memories.*
2. *Know about RAM, ROM, PROM, EPROM, EEPROM, FLASH ROM, NVRAM.*
3. *Understand static and dynamic memories.*
4. *Realize the need for DRAM controller.*
5. *Interfacing ROM, SRAM, DRAM to system bus.*
6. *Understand error detecting and correcting in memories*
7. *Know the current trends, North Bridge, South Bridge.*

Memories are devices into which information can be written to and retrieved whenever required. In this chapter, we shall study various categories of semiconductor memory devices used in a computer system and their interfacing to a system bus. There are secondary memory devices such as magnetic disks and optical disks not discussed here. First-generation computers used relays as memories and second-generation computers used magnetic core memories. All these are obsolete now. From the third generation to present times, computers have used semiconductor memories. There are various categories of memories which are explained here.

TYPES OF MEMORIES

RAM—Random Access Memory

The name is given by the manufacturer but actually it should be Read/Write Memory (RWM). It is volatile meaning that information in the memory is lost/unpredictable when power is removed. The types of RAM that are available are bipolar static RAM, static MOS RAM, Dynamic MOS RAM, Pseudo Static MOS RAM, EDO (Extended Data Output) RAM, Synchronous DRAMs, Rhombus DRAMS, and DDRAMs, etc. In future, FRAMs (Ferro Electrical RAMs) may be available.

ROM—Read Only Memory

This is also called One Time Programmable (OTP). Once programmed, it is permanent and not reversible. ROMs are developed by the manufacturers and not programmed by the user. For heavy volume production, ROMs turn out to be very cheap compared to other non-volatile memories like PROMs, EPROMs, EEROMs, and Flash ROMs.

PROM

It is similar to ROM with a difference that this can be one time programmable by the user using a programming device and not by the manufacturer; hence, it will be very useful for prototyping purposes. Examples are 27PC32 (4k × 8), 27PC64 (8k × 8), 27PC128 (16k × 8), 27PC256 (32k × 8), 27PC512 (64k × 8), 27PC010 (128k × 8), etc.

EPROM

It belongs to the ROM family and has a window on the chip. Once it is programmed, the user has to take precautions not to expose it even to sunlight. Then contents will stay for approximately 10 years and in this sense, it can be classified as non-volatile memory. Whenever you want to rewrite, the EPROM has to be taken out of the circuit board, the window on the chip is to be exposed to ultraviolet light for approximately 5 to 10 minutes. This process erases the entire information on the chip and the user have to program the entire information on a separate programmer. After this is done, the window on this chip is to be closed with a tape and the EPROM can be plugged on the circuit board. Examples of EPROM are 2708 (1k × 8), 2716 (2k × 8), 2732 (4k × 8), 2764 (8k × 8), 27128 (16k × 8), etc.

Digital Interfacing

10

Objectives

At the conclusion of this chapter, you should be able to:

1. *Describe simple input and output, strobed input and output, and handshake input and output.*
2. *Initialize a programmable parallel-port device such as the 8255A for simple input or output and for handshake input or output.*
3. *Interpret the timing waveforms for handshake input and output operations.*
4. *Describe how parallel data is sent to a printer on a handshake basis.*
5. *Show the hardware connections and the programs that can be used to interface keyboards to a microcomputer.*
6. *Show the hardware connections and the programs that can be used to interface alphanumeric displays to a microcomputer.*
7. *Describe how an 8279 can be used to refresh a multiplexed LED display and scan a matrix keyboard.*
8. *Initialize an 8279 for a given display and keyboard format.*
9. *Show the circuitry used to interface high-power devices to microcomputer ports.*
10. *Describe the hardware and software needed to control a stepper motor.*
11. *Describe how optical encoders are used to determine the position, direction of rotation, and speed of a motor shaft.*

The major goal of this chapter and the next is to show you the circuitry and software needed to interface a basic microcomputer with a wide variety of digital and analog devices. In each topic we try to show enough detail so that you can build and experiment with these circuits. Perhaps you can use some of them to control appliances around your house or to solve some problems at work.

In this chapter, we concentrate on the devices and techniques used to get digital data into and out of the basic microcomputer. Then, in the next chapter, we concentrate on analog interfacing.

PROGRAMMABLE PARALLEL PORTS AND HANDSHAKE INPUT/OUTPUT

Throughout the program examples in the preceding chapters, we have used port devices to input parallel data to the microprocessor and to output parallel data from the microprocessor. Most of the available port devices, such as the 8255A on the SDK-86 board, contain two or three ports which can be programmed to operate in one of several different modes. The different modes allow you to use the devices for many common types of parallel data transfer. First we will discuss some of these common methods of transferring parallel data, and then we will show how the 8255A is initialized and used in a variety of I/O operations.

Methods of Parallel Data Transfer

SIMPLE INPUT AND OUTPUT

When you need to get digital data from a simple switch, such as a thermostat, into a microprocessor, all you have to do is connect the switch to an input port line and read the port. The thermostat data is always present and ready, so you can read it at any time.

Likewise, when you need to output data to a simple display device such as an LED, all you have to do is connect the input of the LED buffer on an output port pin and output the logic level required to turn on the light. The LED is always there and ready, so you can send data to it at any time. The timing waveform in Fig. 10.1*a*, represents this situation. The crossed lines on the waveform represent the time at which a new data byte becomes valid on the output lines of the port. The absence of other waveforms indicates that this output operation is not directly dependent on any other signals.

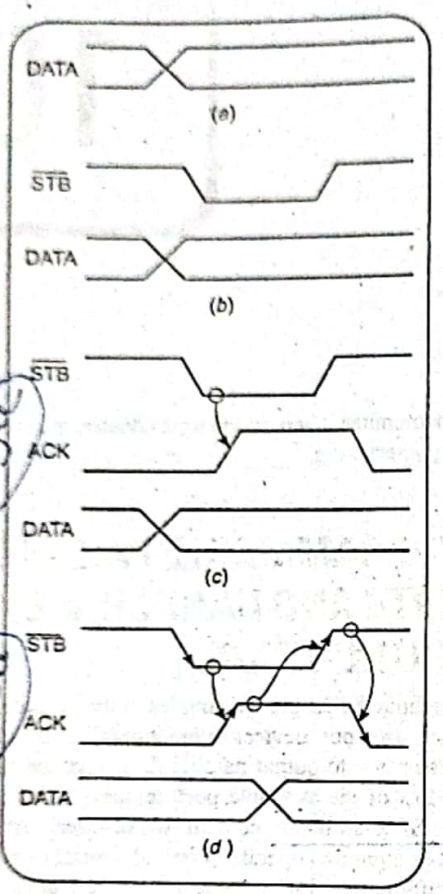

Fig. 10.1 Parallel data transfer. (a) Simple output. (b) Simple strobe I/O. (c) Single handshake I/O. (d) Double handshake I/O.

SIMPLE STROBE I/O

In many applications, valid data is present on an external device only at a certain time, so it must be read in at that time. An example of this is the ASCII-encoded keyboard discussed in Chapter 4. When a key is pressed, circuitry on the keyboard sends out the ASCII code for the pressed key on eight parallel data lines, and then sends out a strobe signal on another line to indicate that valid data is present on the eight data lines. As shown in Fig. 4.19, you can connect this strobe line to an input port line and poll it to determine when you can input valid data from the keyboard. Another alternative, described in Chapter 8, is to connect the strobe line to an interrupt input on the processor and have an interrupt service procedure read in the data when the processor receives an interrupt. The point here is that this transfer is time dependent. You can read in data only when a strobe pulse tells you that the data is valid.

Figure 10.1*b* shows the timing waveforms which represent this type of operation. The sending device, such as a keyboard, outputs parallel data on the data lines, and then outputs an $\overline{\text{STB}}$ signal to let you know that valid data is present.

For low rates of data transfer, such as from a keyboard to a microprocessor, a simple strobe transfer works well. However, for higher speed data transfer this method does not work, because there is no signal which tells the sending device when it is safe to send the next data byte. In other words, the sending system might send data bytes faster than the receiving system could read them. To prevent this problem, a *handshake* data transfer scheme is used.

SINGLE-HANDSHAKE I/O

Figure 10.2 shows the circuit connections and Fig. 10.1*c* shows some example timing waveforms for a *handshake data transfer* from a peripheral device to a microprocessor. The peripheral outputs some parallel data and sends an $\overline{\text{STB}}$ signal to the microprocessor. The microprocessor detects the asserted $\overline{\text{STB}}$ signal on a polled or interrupt basis and reads in the byte of data. Then the microprocessor sends an Acknowledge signal (ACK) to the peripheral to indicate that the data has been read and that the peripheral can send the next byte of data. From the viewpoint of the microprocessor, this operation is referred to as a *handshake* or *strobed input*.

These same waveforms might represent a handshake output from a microprocessor to a parallel printer. In this case, the microprocessor outputs a character to the printer and asserts an $\overline{\text{STB}}$ signal to the printer to tell the printer, "Here is a character for you." When the printer is ready, it answers back with the ACK signal to tell the microprocessor, "I got

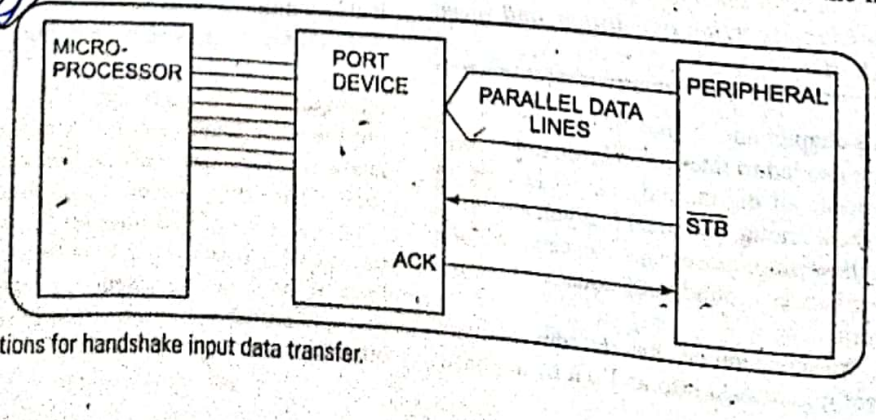

Fig. 10.2 Signal directions for handshake input data transfer.

that one; send me another." We will show you much more about printer interfacing in a later section.

The point of this handshake scheme is that the sending device or system is designed so that it does not send the next data byte until the receiving device or system indicates with an ACK signal that it is ready to receive the next byte.

DOUBLE-HANDSHAKE DATA TRANSFER

For data transfers where even more coordination is required between the sending system and the receiving system, a *double handshake* is used. The circuit connections are the same as those in Fig. 10.2. Figure 10.1*d* shows some example waveforms for a double-handshake input from a peripheral to a microprocessor. Perhaps it will help you to follow these waveforms by thinking of them as a conversation between two people. In these waveforms each signal edge has meaning. The sending device asserts its STB line low to ask, "Are you ready?" The receiving system raises its ACK line high to say, "I'm ready." The peripheral device then sends the byte of data and raises its STB line high to say, "Here is some valid data for you." After it has read in the data, the receiving system drops its ACK line low to say, "I have the data, thank you, and I await your request to send the next byte of data."

For a handshake output of this type, from a microprocessor to a peripheral, the waveforms are the same, but the microprocessor sends the STB signal and the data, and the peripheral sends the ACK signal. In the accompanying laboratory manual we show you how to interface with a speech synthesizer device using this type of handshake system.

Implementing Handshake Data Transfer

For handshake data transfer, a microprocessor can determine when it is time to send the next data byte on a polled or on an interrupt basis. The interrupt approach is usually used, because it makes better use of the processor's time.

The STB or ACK signals for these handshake transfers can be produced on a port pin by instructions in the program. However, this method usually uses too much processor time, so parallel-port devices such as the 8255A have been designed to automatically manage the handshake operation. The 8255A, for example, can be programmed to automatically receive an STB signal from a peripheral, send an interrupt signal to the processor, and send the ACK signal to the peripheral at the proper times. The following sections show you how to connect, initialize, and use an 8255A for a variety of handshake and non handshake applications.

8255A Internal Block Diagram and System Connections

Figure 10.3, shows the internal block diagram of the 8255A. Along the right side of the diagram, you can see that the device has 24 input/output lines. Port A can be used as an 8-bit input port or as an 8-bit output port. Likewise, port B can be used as an 8-bit input port or as an 8-bit output port. Port C can be used as an 8-bit input or output port, as two 4-bit ports, or to produce handshake signals for ports A and B. We will discuss the different modes for these lines in detail a little later.

Along the left side of the diagram, you see the signal lines used to connect the device to the system buses. Eight data lines allow you to write data bytes to a port or the control register and to read bytes from a port or the status register under the control of the RD and WR lines. The address inputs, A0 and A1, allow you to selectively access one of the three ports or the control register. The internal addresses for the device are: port A, 00; port B, 01; port C, 10; control register, 11. Asserting the CS input of the 8255A enables it for reading or writing. The CS input will be connected to the output of the address decoder circuitry to select the device when it is addressed.

The RESET input of the 8255A is connected to the system reset line so that, when the system is reset, all the port lines are initialized as input lines. This is done to prevent destruction of circuitry connected to port lines. If port lines were initialized as outputs after a power-up or reset, the port might try to output to the output of a device connected to the port. The possible argument between the two outputs might destroy one or both of them. Therefore, all the programmable port devices initialize their port lines as inputs when reset.

8255A Operational Modes and Initialization

Figure 10.4, summarizes the different modes in which the ports of the 8255A can be initialized.

MODE 0

When you want to use a port for simple input or output without handshaking, you initialize that port in mode 0. If both port A and port B are initialized in mode 0, then the two halves of port C can be used together as an additional 8-bit port, or they can be used individually as two 4-bit ports. When used as outputs, the port C lines can be individually set or reset by sending a special control word to the control register address. Later we will show you how to do this. The two halves of port C are independent, so one half can be initialized as input, and the other half initialized as output.

MODE 1

When you want to use port A or port B for a handshake (strobed) input or output operation such as we discussed in previous sections, you initialize that port in mode 1.

In this mode, some of the pins of port C function as handshake lines. Pins PC0, PC1, and PC2 function as handshake lines for port B if it is initialized in mode 1. If port A

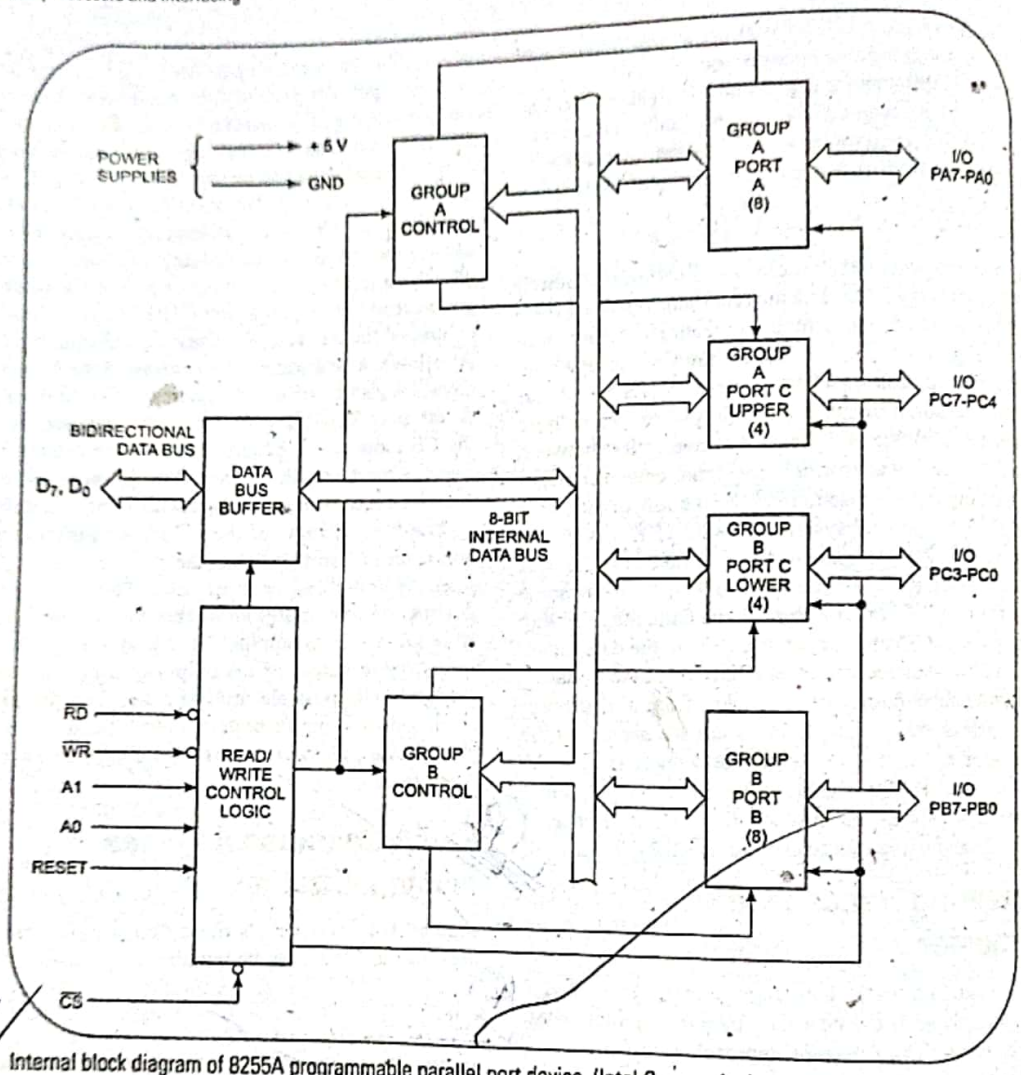

Fig. 10.3 Internal block diagram of 8255A programmable parallel port device. (Intel Corporation)

is initialized as a handshake (mode 1) input port, then pins PC3, PC4, and PC5 function as handshake signals. Pins PC6 and PC7 are available for use as input lines or output lines. If port A is initialized as a handshake output port, then port C pins PC3, PC6, and PC7 function as handshake signals. Port C pins PC4 and PC5 are available for use as input or output lines. Since the 8255A is often used in mode 1, we show several examples in the following sections.

MODE 2

Only port A can be initialized in mode 2. In mode 2, port A can be used for *bidirectional handshake* data transfer. This means that data can be output or input on the same eight lines. The 8255A might be used in this mode to extend the system bus to a slave microprocessor or to transfer data bytes to and from a floppy disk controller board. If port A is initialized in mode 2, then pins PC3 through PC7 are used as handshake lines for port A. The other three pins, PC0 through PC2, can be used for I/O if port B is in mode 0. The three pins will be used for port B handshake lines if port B is initialized in mode 1. After you work your way through the mode 1 examples in the following sections, you should have little difficulty understanding the discussion of mode 2 in the Intel data sheet if you encounter it in a system.

Constructing and Sending 8255A Control Words

Figure 10.5 shows the formats for the two 8255A control words. Note that the MSB of the control word tells the 8255A

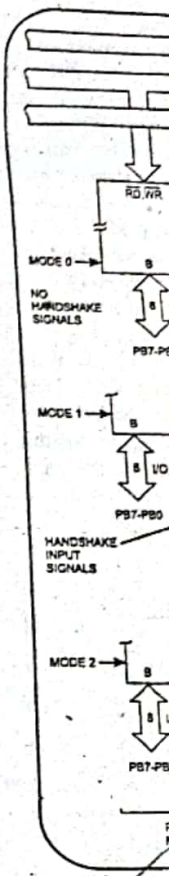

Fig. 10.4 Sur

which control
nition contro
what modes
set/reset con
to set or res
want to enab
transfers. Bo
address of th
As usual,
working you
chapter. Fig
program the
also shows
make up fo
your way th
bit has the
To send
with a MOV
address with

22-A

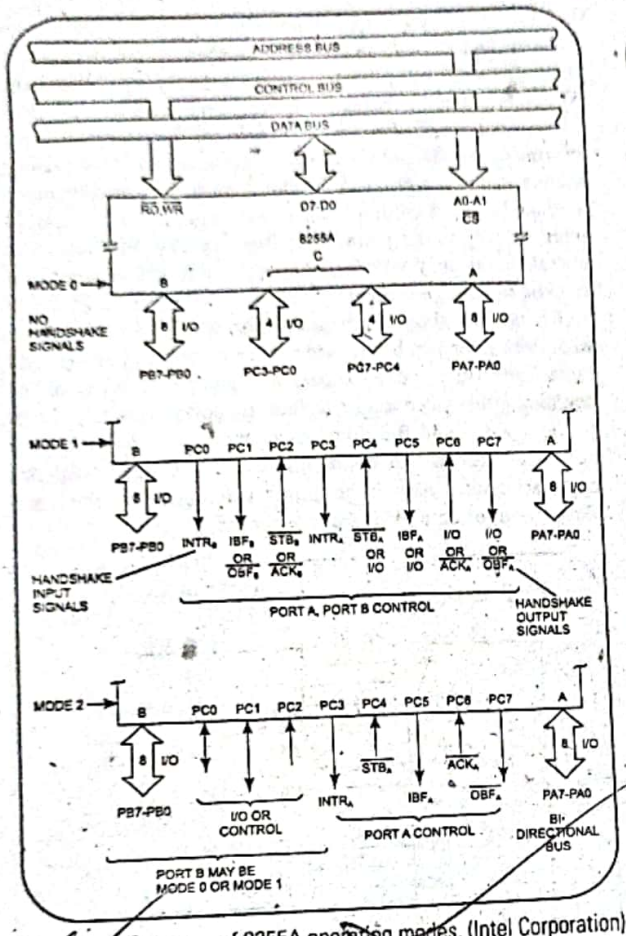

Fig. 10.4 Summary of 8255A operating modes. (Intel Corporation)

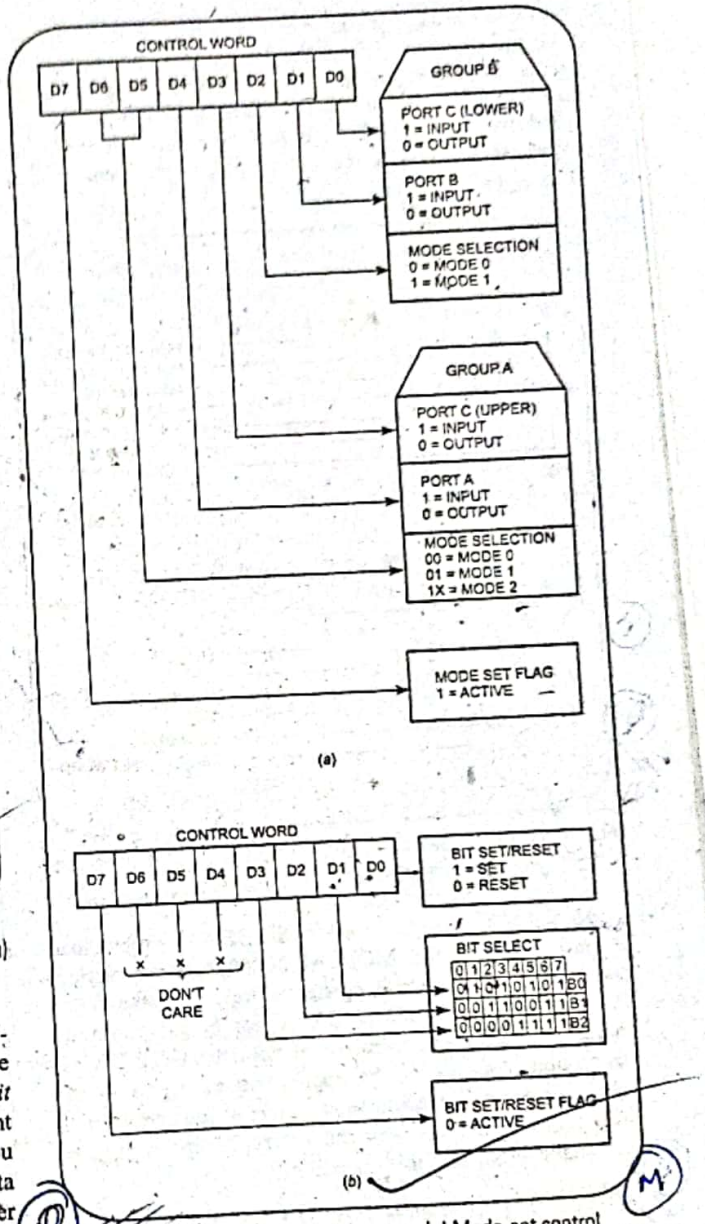

Fig. 10.5 8255A control word formats. (a) Mode-set control word, (b) Port C bit set/reset control word.

which control word you are sending it. You use the *mode definition control word* format in Fig. 10.5*a* to tell the device what modes you want the ports to operate in. You use the *bit set/reset control word* format in Fig. 10.5*b* when you want to set or reset the output on a pin of port C or when you want to enable the interrupt output signals for handshake data transfers. Both control words are sent to the control register address of the 8255A.

As usual, initializing a device such as this consists of working your way through the steps we described in the last chapter. Figure 10.6*a*, shows the control word which will program the 8255A as desired for this example. The figure also shows how you should document any control words you make up for use in your programs. Using Fig. 10.5*a*, work your way through this word to make sure you see why each bit has the value it does.

To send the control word, you load the control word in AL with a MOV AL, 10001110B instruction, point DX at the port address with the MOV DX,0FFFEH instruction, and send the control word to the 8255A control register with the OUT DX, AL instruction.

As an example of how to use the bit set/reset control word, suppose that you want to output a 1 to (set) bit 3 of port C, which was initialized as an output with the mode set control word above. To set or reset a port C output pin, you use the bit set/reset control word shown in Fig. 10.5*b*. Make bit D7 a 0 to identify this as a bit set/reset control word, and put a 1 in

bit D0 to specify that you want to set a bit of port C. Bits D3, D2, and D1 are used to tell the 8255A which bit you want to act on. For this example you want to set bit 3, so you put 011 in these 3 bits. For simplicity and compatibility with future products, make the other 3 bits of the control word 0's. The result, 00000111B, is shown with proper documenta-tion in Fig. 10.6*b*.

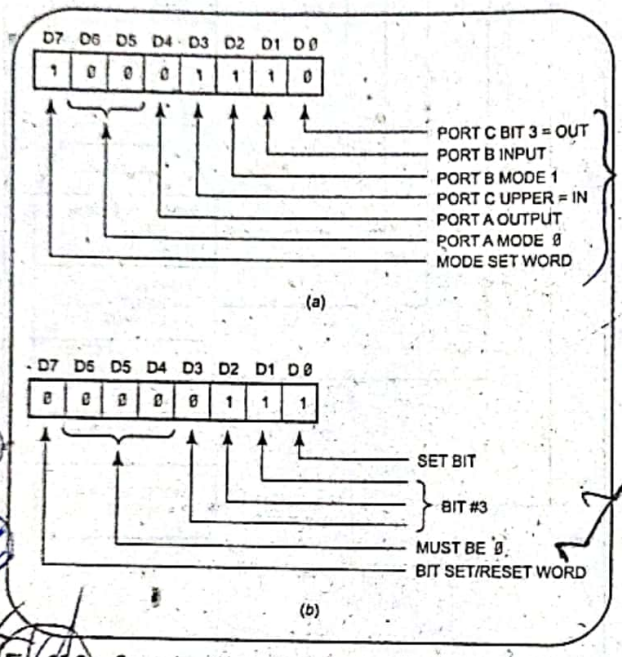

Fig. 10.6 Control word examples for 8255A. (a) Mode-set control word. (b) Port C bit set/reset control word to set bit 3.

To send this control word to the 8255A, simply load it into AL with the MOV AL,00000111B instruction, point DX at the control register address with the MOV DX,0FFFEH instruction if DX is not already pointing there, and send the control word with the OUT DX,AL instruction. As part of the application examples in the following sections, we will show you how you know which bit in port C to set to enable the interrupt output signal for handshake data transfer.

8255A Handshake Application Examples

INTERFACING TO A MICROCOMPUTER-CONTROLLED LATHE

All the machines in the machine shop of our computer-controlled electronics factory operate under microcomputer control. One example of these machines is a lathe which makes bolts from long rods of stainless steel. The cutting instructions for each type of bolt that we need to make are stored on a 3/4-in.-wide teletype-like metal tape. Each instruction is represented by a series of holes in the tape. A tape reader pulls the tape through an optical or mechanical sensor to detect the hole patterns and converts these to an 8-bit parallel code. The microcomputer reads the instruction codes from the tape reader on a handshake basis and sends the appropriate control instructions to the lathe. The microcomputer must also monitor various conditions around the lathe. It must, for example, make sure the lathe has cutting lubricant oil, is not out of material to work on, and is not jammed up in some way. Machines that operate in this way are often referred to as *computer numerical control*, or *CNC, machines.*

Figure 10.7 shows in diagram form how you might use an 8255A to interface a microcomputer to the tape reader and lathe. Later in the chapter, we will show you some of the actual circuitry needed to interface the port pins of the 8255A to the sensors and the high-power motors of the lathe. For now, we want to talk about initializing the 8255A for this application and analyze the timing waveforms for the handshake input of data from the tape reader.

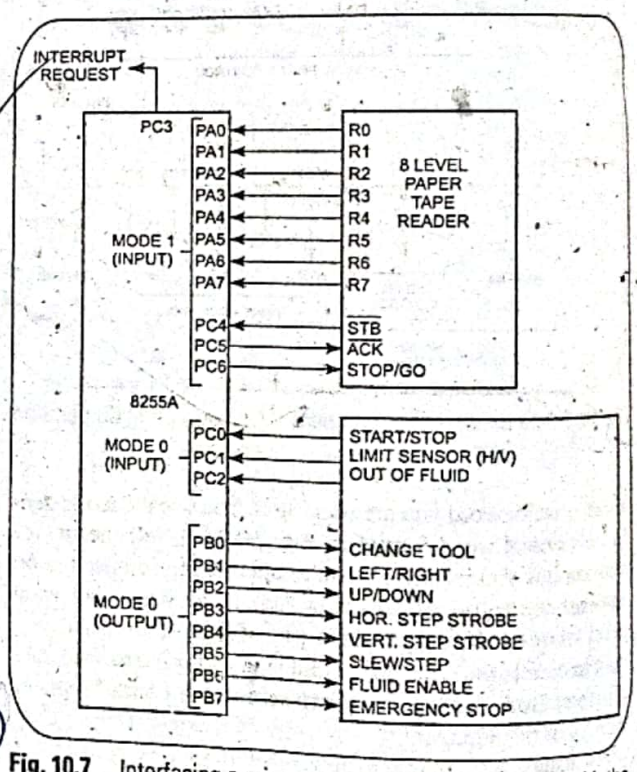

Fig. 10.7 Interfacing a microprocessor to a tape reader and lathe.

Your first task is to make up the control word which will initialize the 8255A in the correct modes for this application. To do this, start by making a list showing how you want each port pin or group of pins to function. Then put in the control word bits that implement those pin functions. For our example here,

Port A needs to be initialized for handshake input (mode 1) because instruction codes have to be read in from the tape reader on a handshake basis.

Port B needs to be initialized for simple output (mode 0). No handshaking is needed here because this port is being used to output simple on or off control signals to the lathe.
Port C, bits PC0, PC1, and PC2 are used for simple input of sensor signals from the lathe.
Port C, bits PC3, PC4, and PC5 function as the handshake signals for the data transfer from the tape reader connected to port A.
Port C, bit PC6 is used for output of the STOP/GO signal to the tape reader.
Port C, bit PC7 is not used for this example.

Figure 10.8 shows the control word to initialize the 8255A for these pin functions. You send this word to the control register address of the 8255A as described above.

Before we go on, there is one more point we have to make about initializing the 8255A for this microcomputer-controlled lathe application. In order for the handshake input data transfer from the tape reader to work correctly, the interrupt request signal from bit PC3 has to be enabled. This is done by sending a bit set/reset control word for the appropriate bit of port C. Figure 10.9 shows the port C bit that must be set to enable the interrupt output signal for each of the 8255A handshake modes. For the example here, port A is being used for handshake input, so according to Fig. 10.9, port C, bit PC4 must be set to enable the interrupt output for this operation. The bit set/reset control word to do this is 00001001B. You send this bit set/reset control word to the control address of the 8255A.

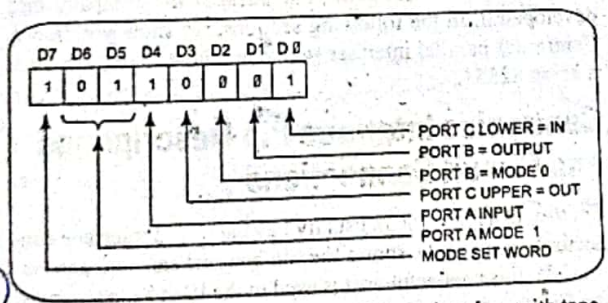

Fig. 10.8 Control word to initialize 8255A for interface with tape reader and lathe.

	Port C Interrupt Signal Pin Number	To Enable Interrupt Request Set Port C bit
MODE 1		
Port A IN	PC3	PC4
Port B IN	PC0	PC2
Port A OUT	PC3	PC6
Port B OUT	PC0	PC2
MODE 2		
Port A IN	PC3	PC4
Port A OUT	PC3	PC6

Fig. 10.9 Port C bits to set to enable interrupt request outputs for handshake modes.

Handshake data transfer from the tape reader to the 8255A can be stopped by disabling the 8255A interrupt output on port C, pin PC3. This is done by resetting bit PC4 with a bit set/reset control word of 00001000. You will later see another example of the use of this interrupt enable/disable process in Fig. 10.16.

As another example of 8255A interrupt output enabling, suppose that you are using port B as a handshake output port. According to Fig. 10.9, you need to set bit PC2 to enable the 8255A interrupt output signal. The bit set/reset control word to do this is 00000101.

Now let's talk about how the program for this machine might operate and how the handshake data transfer actually takes place.

After initializing everything, you would probably read port C, bits PC0, PC1, and PC2 to check if the lathe was ready to operate. For any 8255A mode, you read port C by simply doing an input from the port C address. Then you output a start command to the tape reader on bit PC6. This is done with a bit set/reset command. Assuming that you want to reset bit PC6 to start the tape reader, the bit set/reset control word for this is 00001100. When the tape reader receives the Go command, it will start the handshake data transfer to the 8255A. Let's work our way through the timing waveforms in Fig. 10.10, to see how the data transfer takes place.

The tape reader starts the process by sending out a byte of data to port A on its eight data lines. The tape reader then asserts its $\overline{STB}$ line low to tell the 8255A that a new byte of data has been sent. In response, the 8255A raises its Input Buffer Full (IBF) signal on PC5 high to tell the tape reader that it is ready for the data. When the tape reader detects the IBF signal at a high level, it raises its $\overline{STB}$ signal high again. The rising edge of the $\overline{STB}$ signal has two effects on the 8255A. It first latches the data byte in the input latches of the 8255A. Once the data is latched, the tape reader can remove the data byte in preparation for sending the next data byte. This is shown by the dashed section on the right side of the data waveform in Fig. 10.10. Second, if the interrupt signal output has been enabled, the rising edge of the $\overline{STB}$ signal will cause the 8255A to output an Interrupt Request signal to the microprocessor on bit PC3.

The processor's response to the interrupt request will be to go to an interrupt service procedure which reads in the byte of data latched in port A. When the $\overline{RD}$ signal from the microprocessor goes low for this read of port A, the 8255A will automatically reset its Interrupt Request signal on PC3. This is done so that a second interrupt cannot be caused by the same data byte transfer. When the processor raises its $\overline{RD}$ signal high again at the end of the read operation, the 8255A automatically drops its IBF signal on PC5 low again. IBF going low again is the signal to the tape reader that the data transfer is complete and that it can send the next byte of data. The time between when the 8255A sends the Interrupt Request signal and when the processor reads the data byte