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

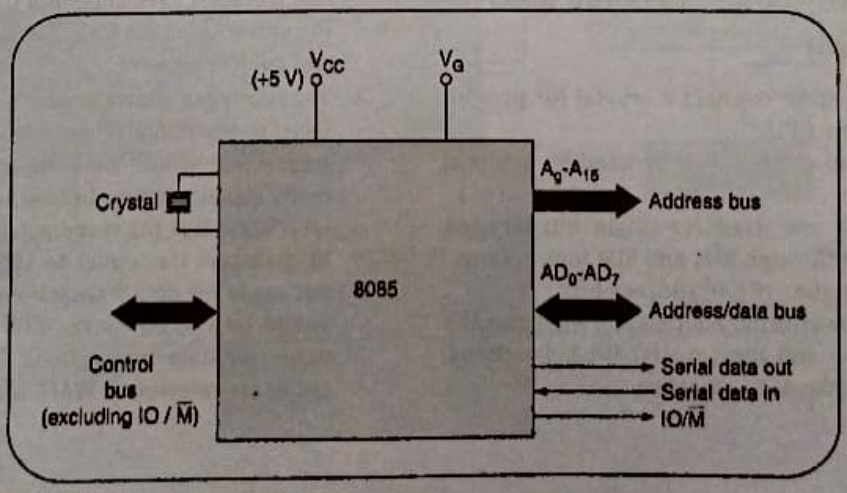

Fig. 2.1 Block diagram of Intel 8085.

- Provides machine cycle status
- 80 machine instructions

The basic block diagram of 8085 is shown in Fig. 2.1.

There are eight address lines which are unidirectional. The lower signals are of multiplexed nature. Together it will give a 16-bit address but the moment the address placing is over, then these eight signals turn into bidirectional data bus. To indicate that it has put the complete 16-bit address bus, there is a control signal, Address Latch Enable (ALE), which will indicate that the address is available and the lower-order 8-bit address has to be externally latched before it becomes the data bus. It has an IO/M/ signal to indicate whether it is I/O mapped I/O or memory mapped I/O. It sends out one bit serially and accepts one bit serially in this line. Read and write are part of the control bus.

The detailed pin diagram of the 8085 is given in Fig. 2.2.

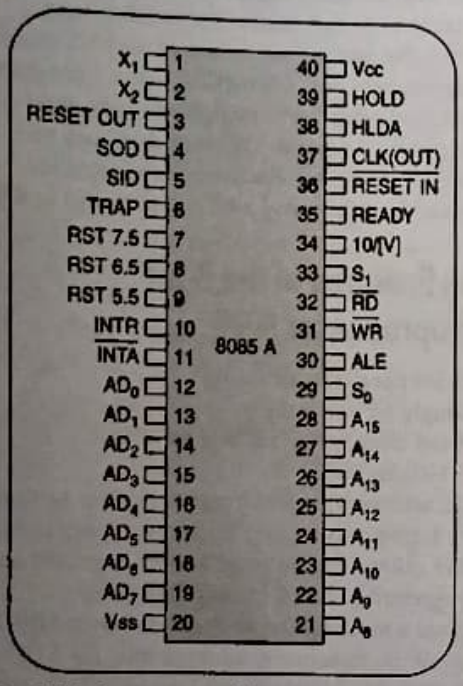

Fig. 2.2 8085 microprocessor chip.

Functions of Various Pins of the 8085 Microprocessor

- **X1, X2** are used to connect a crystal for providing clock to the CPU.
- **8085** provides clock-out to be used by external peripherals.
- **SID** and **SOD** are used for-single bit serial-in and serial-out through RIM and SIM instructions.
- **A15-A8** are higher-order address bits.
- **AD7-AD** are lower-order address bits when the ALE signal is active and they are D7-D0 bidirectional data bus when the ALE is inactive.

Interrupt Signals Available in 8085 Microprocessor

Table 2.1 Interrupt Signals of 8085

Interrupt	*Vector Address*	*Remarks*
TRAP	24	This is the highest priority interrupt, non-maskable, both edge and level sensitive
RST 7. 5	3C	Second priority, maskable, raising edge sensitive
RST 6. 5	34	Third priority, maskable, high-level sensitive
RST 5. 5	2C	Fourth priority, maskable, high-level sensitive
INTR	Vector has to be obtained from interrupt controller in response to INTA	Lowest priority, maskable, level sensitive

The machine status as given by S1, S0 are as follows:

S1	*S0*	*Machine cycle*
0	0	HALT
0	1	WRITE
1	0	READ
1	1	OPCODE FETCH

1. The Hold signal will be used to inform the CPU to release the address, data and control buses to be used by the external requesting unit such as DMA. The Hold acknowledgement will inform the external device that the CPU has relinquished the bus and bus lines have gone into tri-state and requesting unit such as the DMA can use the address, data and control buses.
2. The Ready pin allows slower memories to be interfaced to the 8085. When memory is accessed, the transaction would be completed only when the ready signal is active. In case slower memories are interfaced, it is the responsibility of the memories to de-assert the signal to indicate that they are not ready for data transactions. Some clock cycles would be wasted by the CPU till the memory is ready for data transactions. These wasted clock cycles are referred as WAIT states.

3. The 8085 supports both modes of I/O. To distinguish whether it is memory mapped I/O or I/O mapped I/O, 8085 issues out a signal called IO/M/. Active high of this signal indicates I/O mode access and active low indicates memory mode access.
4. WR/ and RD/ are write and read signals.
5. RESET IN/ is given externally to reset the 8085 microprocessor.
6. RESET OUT is the signal which 8085 generates in response to RESET IN? to reset the other support devices connected to 8085.

The internal details of the 8085 processor are shown in Fig. 2.3.

REGISTERS IN 8085

Accumulator

The accumulator is an 8-bit register used for arithmetic, logic, I/O, and load/store operations.

Flag

The flag is an 8-bit register containing five 1-bit flags:

1. **Sign** — Set if the most significant bit of the result is set.
2. **Zero** — Set if the result zero.

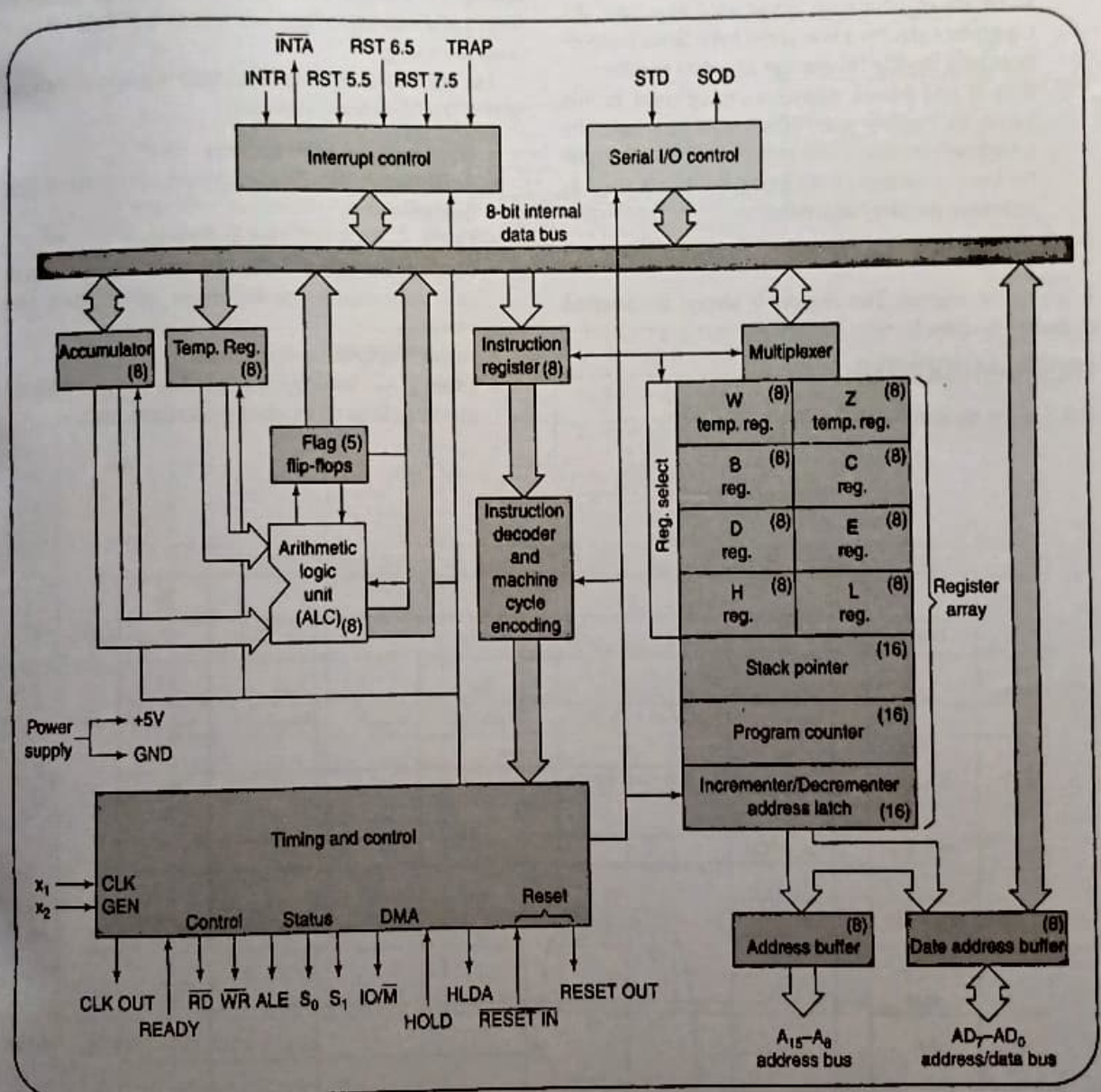

Fig. 2.3 Internal details of 8085.

3. **Auxiliary carry** — Set if there was a carry out from the bit 3 to the bit 4 of the result and this is used in BCD arithmetic.
4. **Parity** — Set if the parity (the number of set bits in the result) is even.
5. **Carry** — Set if there was a carry during addition, or borrow during subtraction/comparison.

General Registers

1. 8-bit B and 8-bit C registers can be used as one 16-bit BC register pair. When used as a pair, the C register contains a low-order byte. Some instructions may use the BC register as a data pointer.
2. 8-bit D and 8-bit E registers can be used as one 16-bit DE register pair. When used as a pair, the E register contains a low-order byte. Some instructions may use the DE register as a data pointer.
3. 8-bit H and 8-bit L registers can be used as one 16-bit HL register pair. When used as a pair, the L register contains a low-order byte. The HL register usually contains a data pointer that is used to reference memory addresses.

STACK POINTER

It is a 16-bit register. This register is always incremented or decremented by 2.

PROGRAM COUNTER

It is a 16-bit register.

The addressing modes available in 8085 are:

- Direct
- Register
- Register indirect
- Implied
- Immediate

Opcode fetch, memory peripheral READ and Write timing diagrams are shown in Figs. 2.4, 2.5 and 2.6.

8085 MACHINE INSTRUCTIONS

8085 machine instruction could be one byte in length or two bytes or three bytes depending on the addressing modes. The opcode will indicate how many bytes are to be fetched to form a machine instruction. The opcode is of fixed type and is one byte long.

The instruction set of the Intel 8085 microprocessor consists of the following instructions:

- **Data moving instructions**
- **Arithmetic** — Add, subtract, increment and decrement
- **Logic** — AND, OR, XOR and rotate
- **Control transfer** — Conditional, unconditional, call subroutine, return from subroutine and restarts
- **Input/Output instructions**
- **Others** — Setting/clearing flag bits, enabling/disabling interrupts, stack operations, etc.

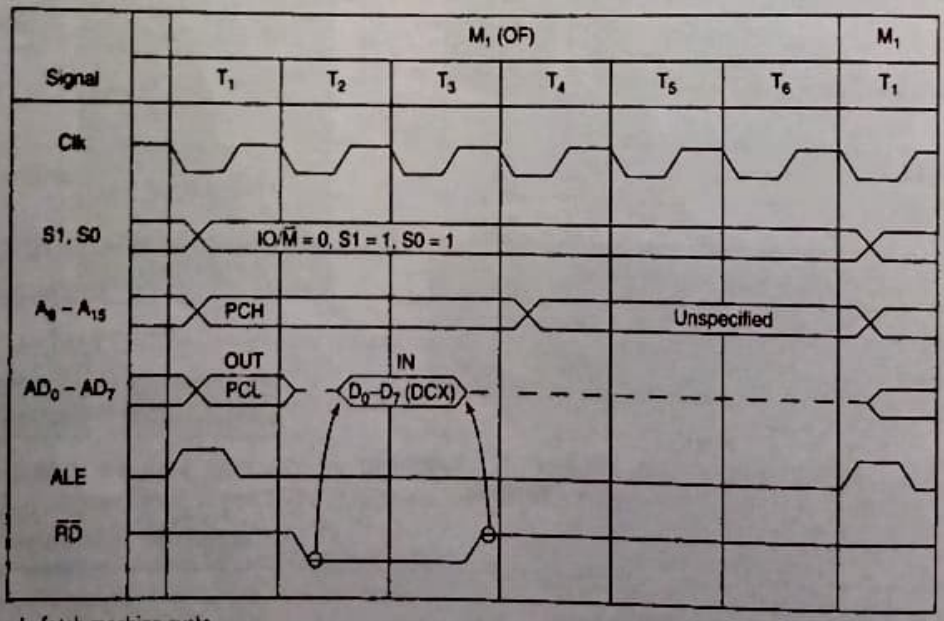

Fig. 2.4 Opcode fetch machine cycle.

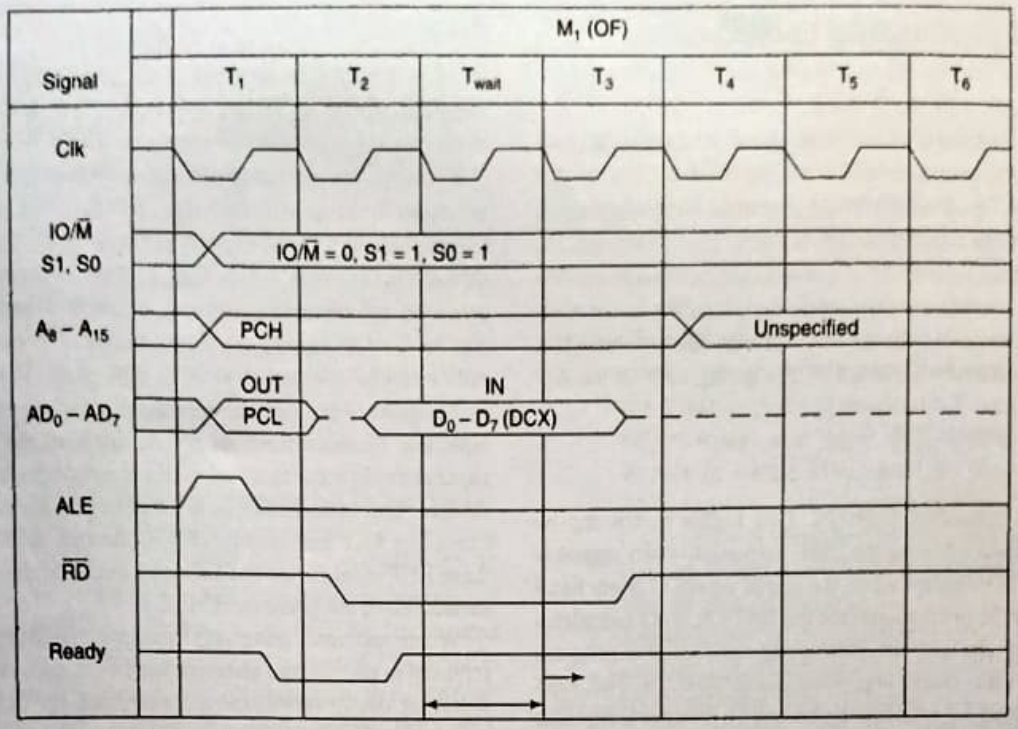

Fig. 2.5 Memory read machine cycle.

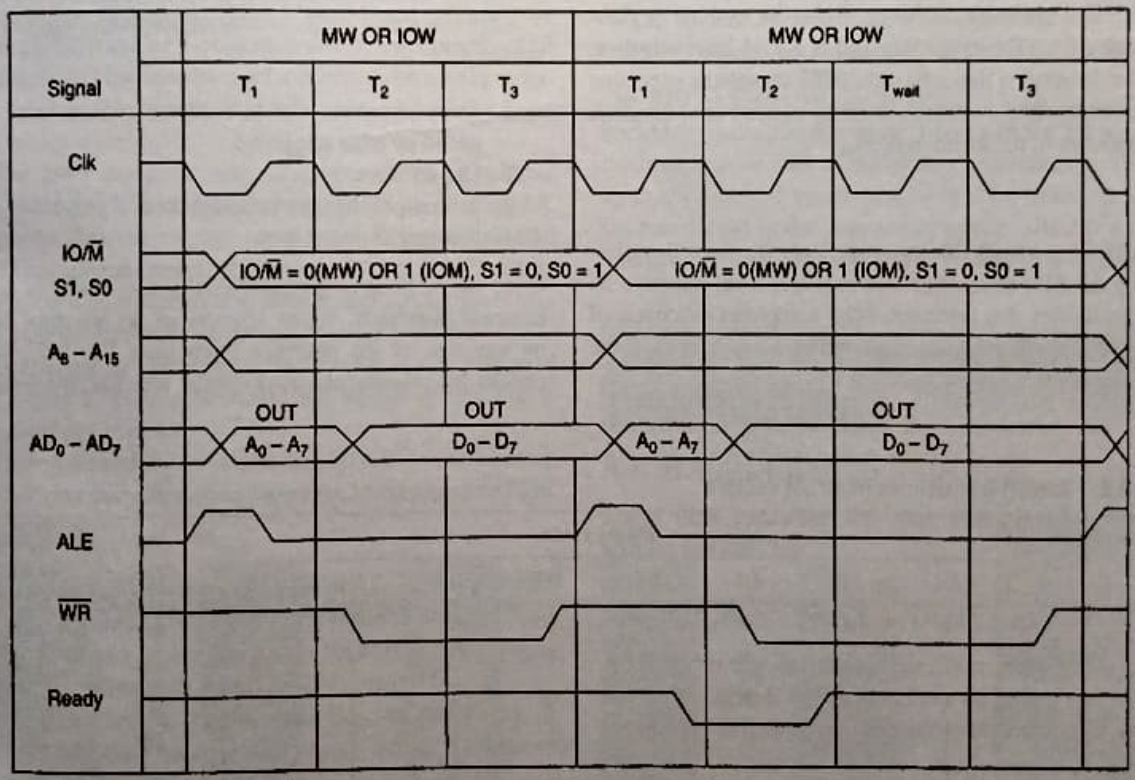

Fig. 2.6 Memory write machine cycle.

Interrupt Processing in 8085

TRAP

TRAP is a non-maskable interrupt, that is, it cannot be disabled by an instruction. In order for the 8085 to service this interrupt, the signal on the TRAP pin must have a sustained HIGH level with a leading edge. If this condition occurs, the 8085 completed execution of the current instruction pushes the program counter onto the stack, and branches to the location 0024_{16} (interrupt address vector for the TRAP). Note that the TRAP interrupt is disabled by the falling edge of the signal on the pin.

RST7.5

RST7.5 is a maskable interrupt. This means that it can be enabled or disabled using the SIM instruction. 8085 responds to the RST7.5 interrupt when the signal on RST7.5 pin has a leading edge. In order to service the RST7.5, 8085 completes execution of the current instruction, pushes the program counter onto the stack, and branches to 003C16. The 8085 remembers RST7.5 interrupt by setting an internal D flip-flop by the leading edge.

RST6.5

RST6.5 is a maskable interrupt. It can be enabled or disabled using the SIM instruction and is HIGH level sensitive. In order to service this interrupt, 8085 completes executing current instruction, saves the program counter onto the stack and branches to the location 0034_{16}.

RST5.5

The RST5.5 is a maskable interrupt. It can be enabled or disabled by the SIM instruction and is HIGH level sensitive. In order to service this interrupt, 8085 completes execution of the current instruction, saves the program counter onto the stack and branches to 0020_{16}.

INTR

INTR is a maskable interrupt. This is also called *handshake interrupt*. INTR is HIGH level sensitive. When no other interrupts are active and the signal on the INTR pin is HIGH, 8085 completes execution of the current instruction, and generates an interrupt acknowledge, INTA, LOW pulse on the control bus. 8085 then expects either a 1-byte CALL (RST0 through RST7) or a 3-byte CALL. This instruction must be provided by external hardware. In other words, the INTA can be used to enable a tri-state buffer. The output of this buffer can be connected to 8085 data lines. The buffer can be designed to provide the appropriate opcode on data lines. Note that the occurrence of INTA, turns off the 8085 interrupt system in order to avoid multiple interrupts from a single device. Also note that there are eight software interrupts by executing RST instructions (RST0 through RST7). Each of these RST instructions has a defined vector address. The vector addresses are given in Table 2.2.

When external interrupts enter, external hardware is required to provide the interrupt vector for each interrupt. On receiving one or more external interrupts, the external hardware generates the INTR signal. INTR is given least priority. If no other interrupts are present and only INTR is present in 8085, it gives out a signal called Interrupt acknowledge-INTA. The external hardware is the Inerrupt Conroller 8259.

8259 can accept eight external interrupts and will be intialized by CPU to

1. Know whether the interrupt signal is level triggered or edge triggered
2. Call address
3. Interrupts that are to be masked
4. Priority of interrupts

Once the 8259 is programmed, it is now ready to accept external interrupts. When it receives an interrupt signal on any one of the interrupt input lines IR0-IR7, it first checks for priority and also checks whether this interrupt

Table 2.2 Restart Instructions Interrupt Vectors

Mnemonics	*Binary Code*								*Hex Code*	*Call Location in Hex*
	D7	*D6*	*D5*	*D4*	*D3*	*D2*	*D1*	*D0*		
RST 0	1	1	0	0	0	1	1	1	C7	0000
RST 1	1	1	0	0	1	1	1	1	CF	0008
RST 2	1	1	0	1	0	1	1	1	D7	0010
RST 3	1	1	0	1	1	1	1	1	DF	0018
RST 4	1	1	1	0	0	1	1	1	E7	0020
RST 5	1	1	1	0	1	1	1	1	EF	0028
RST 6	1	1	1	1	0	1	1	1	F7	0030
RST 7	1	1	1	1	1	1	1	1	FF	0038

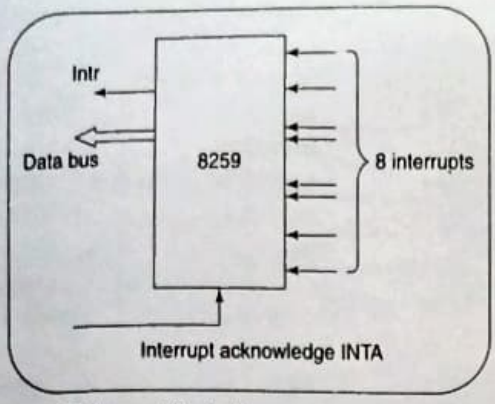

Fig. 2.7 8259 Interrupt Controller

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.

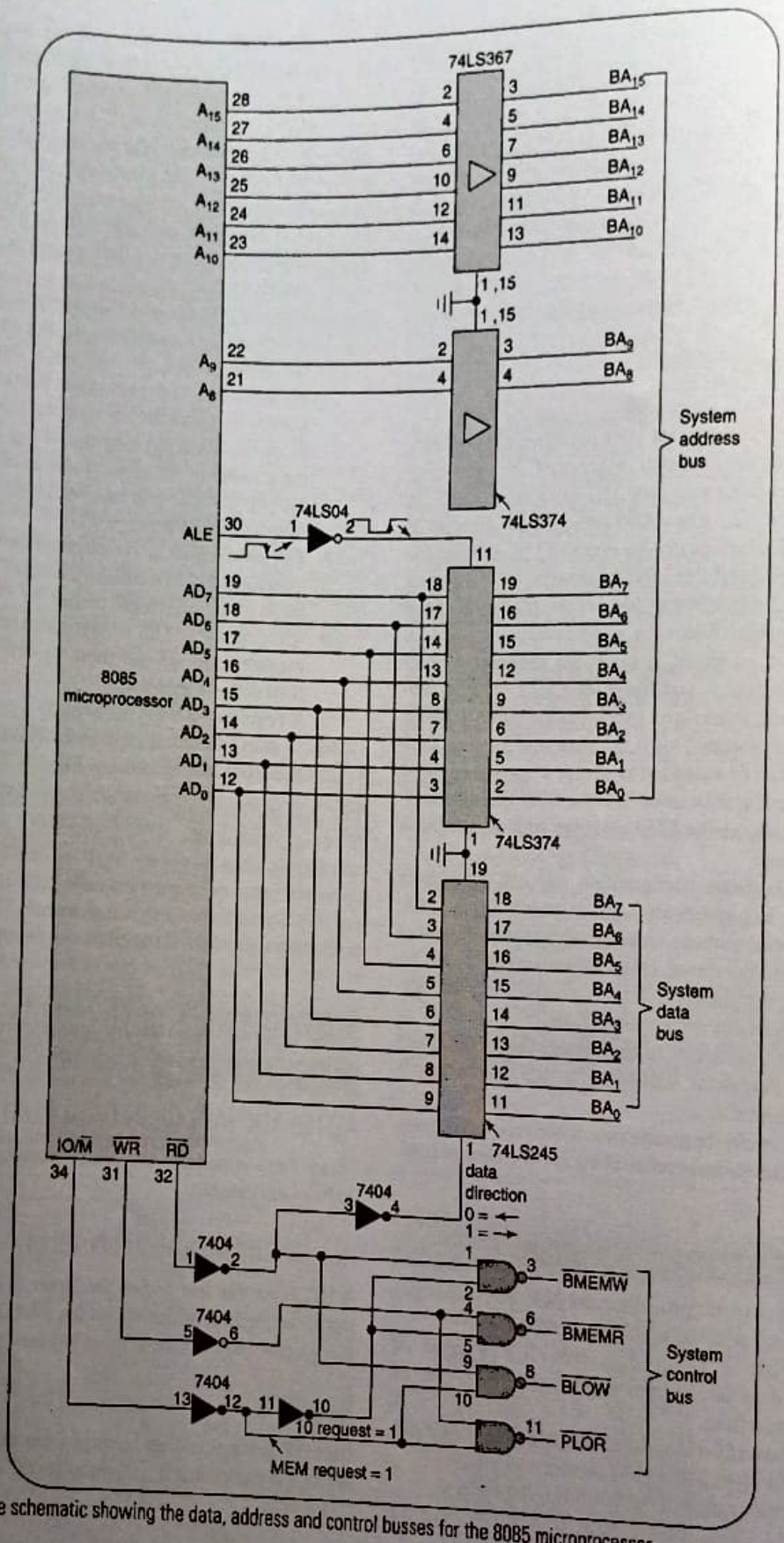

Fig. 2.8 Complete schematic showing the data, address and control busses for the 8085 microprocessor.

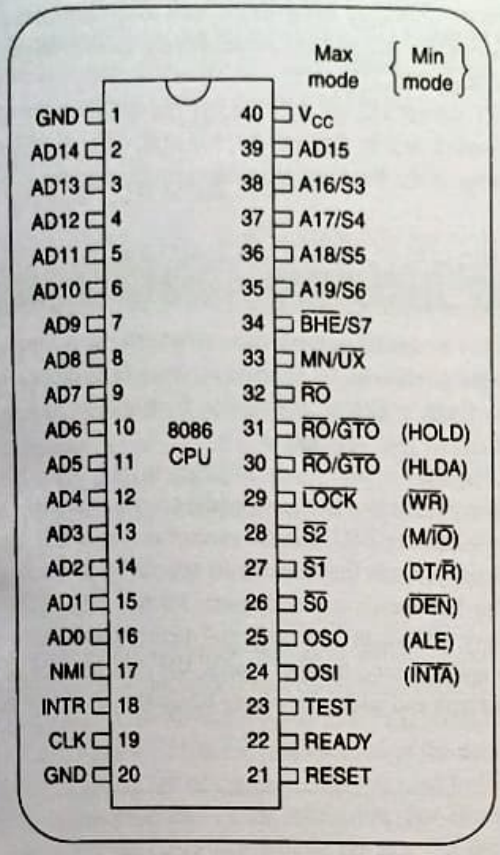

Fig. 2.9 The 8086 pin assignments.

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.

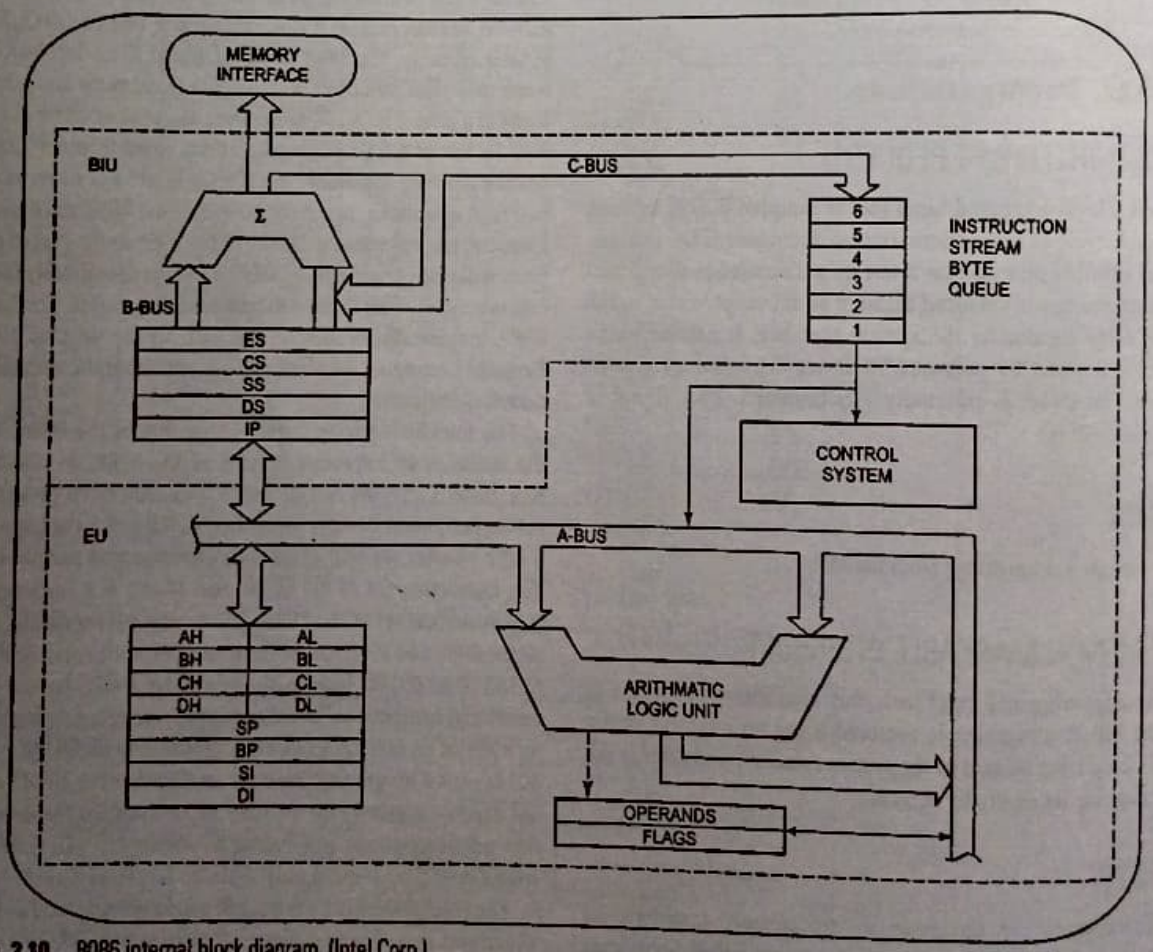

Fig. 2.10 8086 internal block diagram. (Intel Corp.)

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.

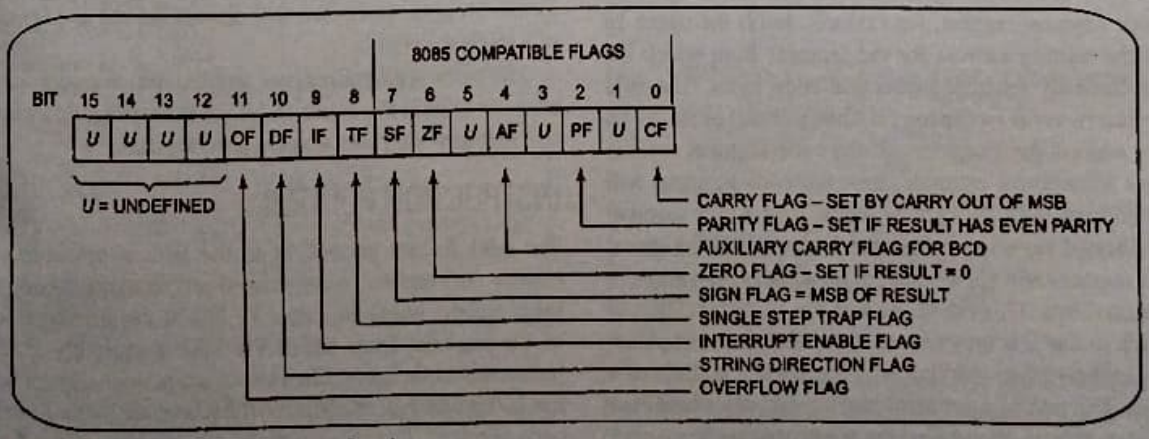

Fig. 2.11 8086 flag register format. (Intel Corp.)

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*.

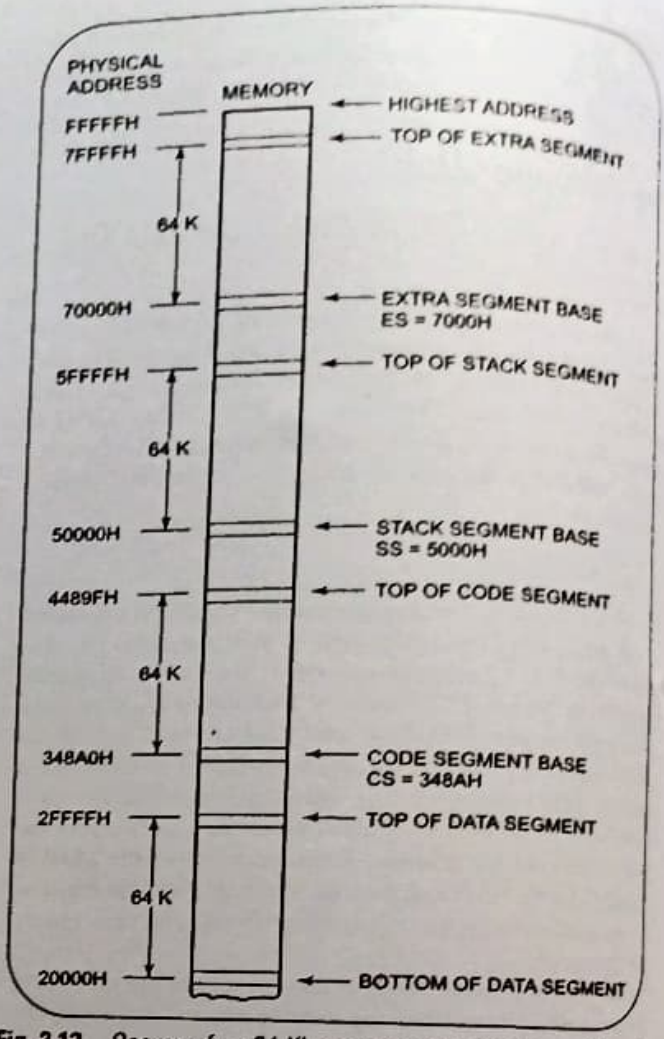

Fig. 2.12 One way four 64-Kbyte segments might be positioned within the 1-Mbyte address space of an 8086.

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.

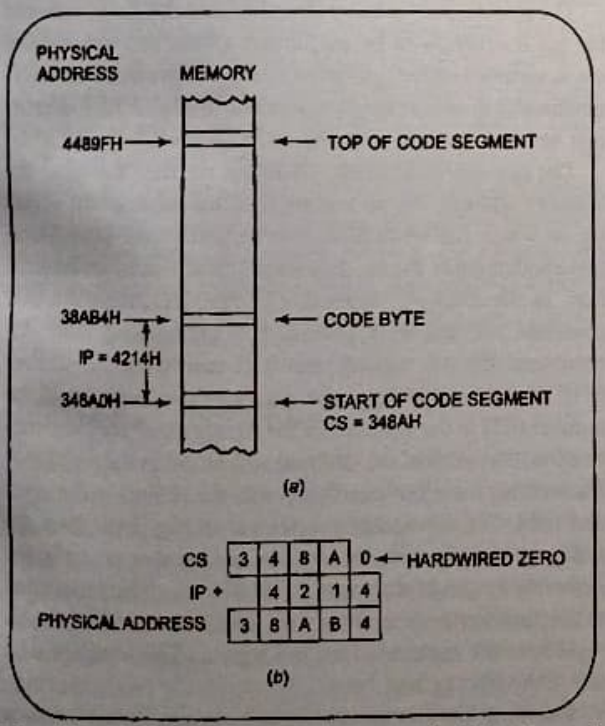

Fig. 2.13 Addition of IP to CS to produce the physical address of the code byte, (a) Diagram, (b) Computation.

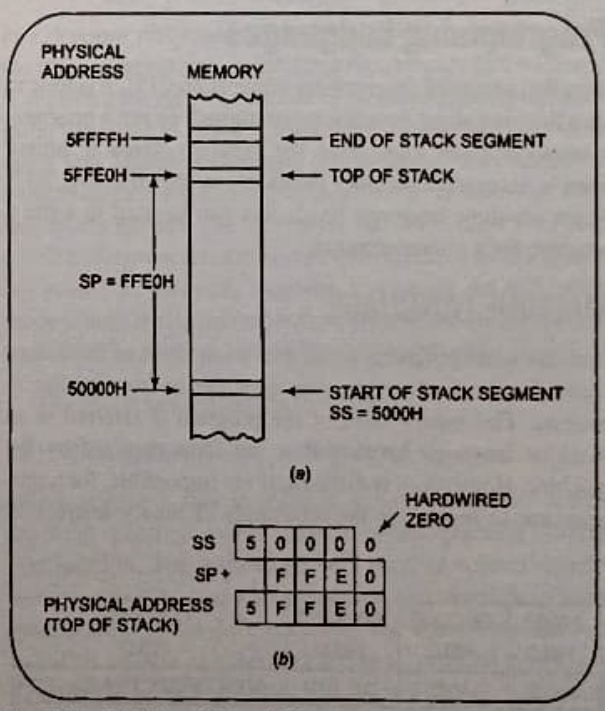

Fig. 2.14 Addition of SS and SP to produce the physical address of the top of the stack. (a) Diagram, (b) Computation.

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.

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.

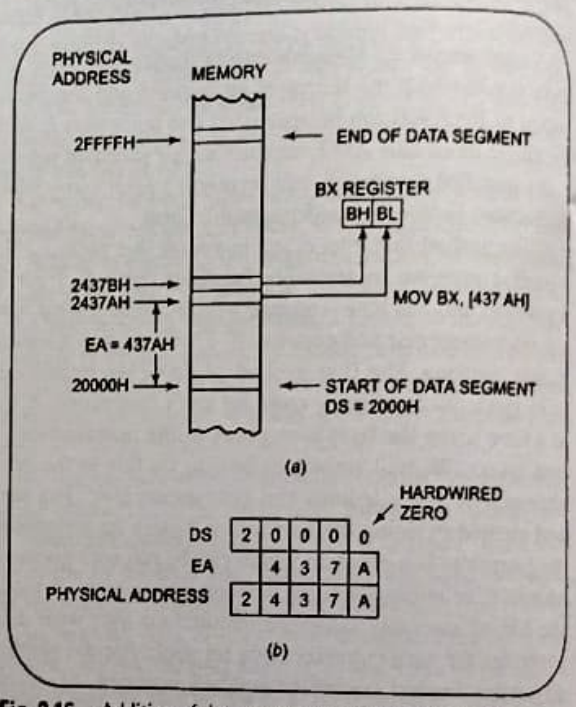

Fig. 2.16 Addition of data segment register and effective address to produce the physical address of the data byte, (a) Diagram, (b) Computation.

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

of data lines, and port addresses, which you need to write programs. Now we will look at the bus signals, timing, and circuit connections of an 8086 and an 8088. In a later chapter we will show the hardware connections for the 80286 and 80386 microprocessors.

System Overview

Figure 2.17*a* shows a block diagram of a simple 8086-based microcomputer. This diagram is a closer look at the generalized microcomputer in Fig. 2.10. First, find the 8086 CPU, the ROM, and the RAM in Fig. 2.17*a*. Next, look for the ports, represented by the block labeled MCS-80 PERIPHERAL. As we discuss in detail later, there is a wide variety of port devices available. Some examples are parallel port devices such as the 8255A, serial port devices, special port devices which interface with CRTs, port devices which interface with keyboards, and port devices which interface with floppy disks.

Next, find the control bus, address bus, and data bus in Fig. 2.17*a*. The basic control bus consists of the signals labeled M/$\overline{IO}$, $\overline{RD}$, and $\overline{WR}$ at the top of the figure. If the 8086 is doing a read from memory or from a port, the $\overline{RD}$ signal will be asserted. If the 8086 is doing a write to memory or to a port, the $\overline{WR}$ signal will be asserted. During a read from memory or a write to memory, the M/$\overline{IO}$ signal will be high, and during port operations the M/$\overline{IO}$ signal will be low. As we show you in detail later, the $\overline{RD}$, $\overline{WR}$, and M/$\overline{IO}$ signals are used to enable addressed devices.

The address bus and the data bus are shown separately on the right side of Fig. 2.17*a*, but where they leave the 8086, the two buses are shown as a single bus labeled ADDR/DATA. The reason for this is that, in order to save pins, the lower 16 bits of addresses are multiplexed on the data bus. Here's an overview of how this works.

As a first step in any operation where it accesses memory or a port, the 8086 sends out the lower 16 bits of the address on the data bus. External latches such as the 74LS373 octal devices shown in Fig. 2.17*a* are used to "grab" this address and hold it during the rest of the operation. To strobe these latches at the proper time, the 8086 outputs a signal called Address Latch Enable or ALE. Once the address is stored on the outputs of the latches, the 8086 removes the address from the address/data bus and uses the bus for reading or writing data.

Another section of Fig. 2.17*a* to look at briefly is the block labeled 8286 Transceiver. This block represents bidirectional three-state buffers. For a very small system, these buffers are

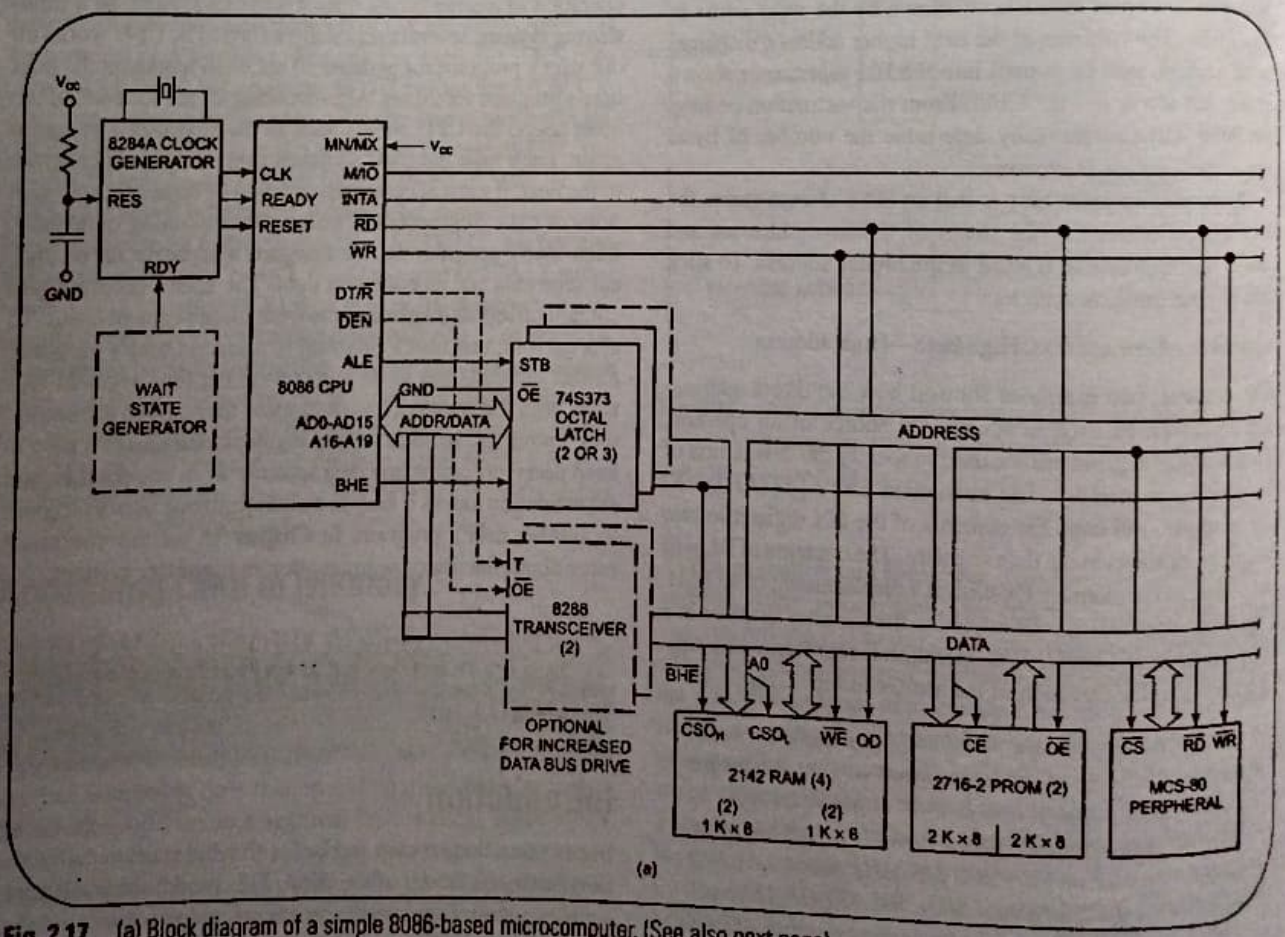

Fig. 2.17 (a) Block diagram of a simple 8086-based microcomputer. (See also next page)

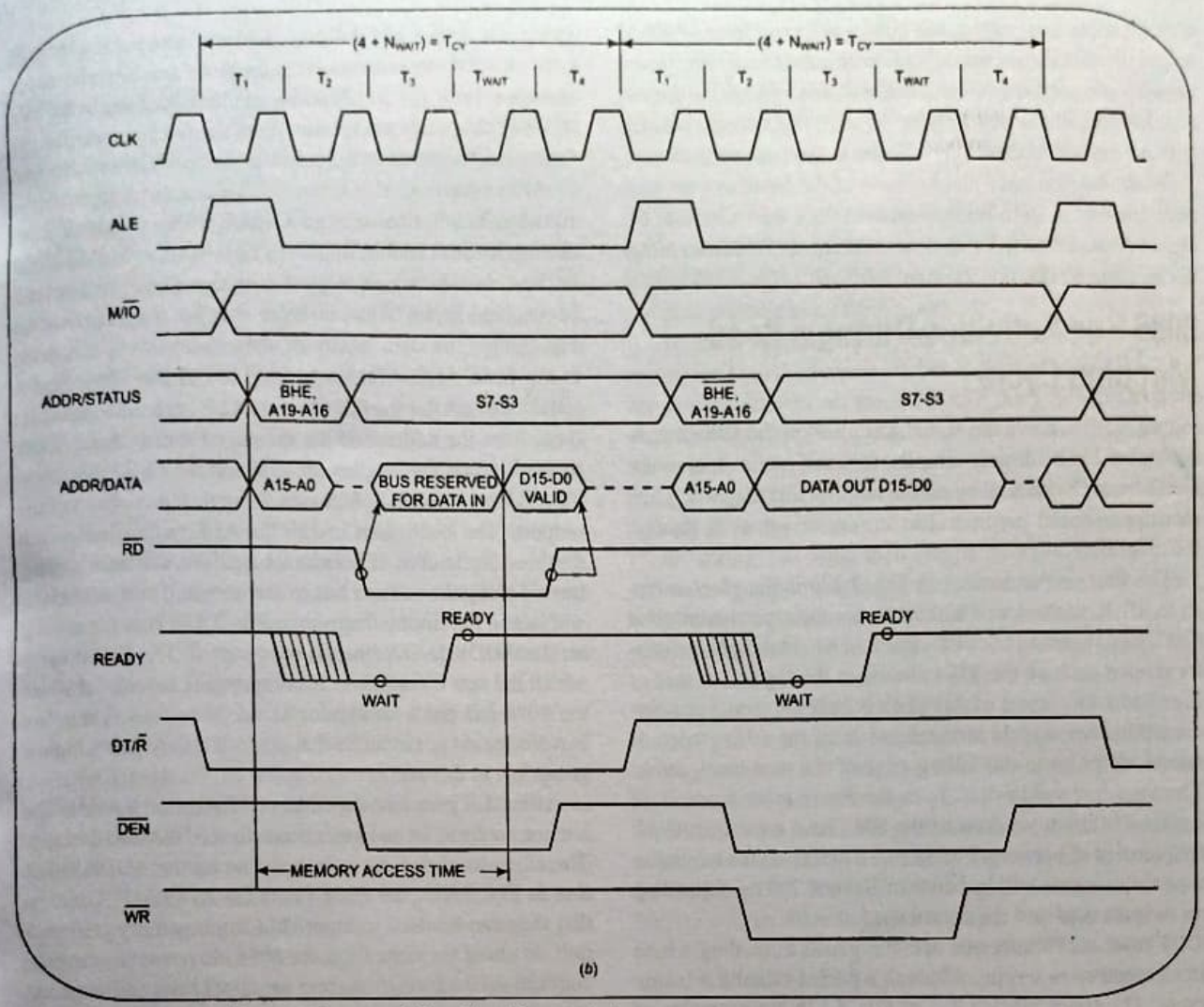

Fig. 2.17 (continued) (b) Basic 8086 system timing. (Intel Corporation)

not needed, but as more devices are added to a system, they become necessary. Here's why. Most of the devices—such as ROMs, RAMs, and ports— connected on microprocessor buses have MOS inputs, so on a dc basis they don't require much current. However, each input or output added to the system data bus, for example, acts like a capacitor of a few picofarads connected to ground. In order to change the logic state on these signal lines from low to high, all this added capacitance must be charged. To change the logic state to a low, the capacitance must be discharged. If we connect more than a few devices on the data bus lines, the 8086 outputs cannot supply enough current drive to charge and discharge the circuit capacitance fast enough. Therefore, we add external high-current drive buffers to do the job.

Buffers used on the data bus must be bidirectional because the 8086 sends data out on the data bus and also reads data in on the data bus. The *Data Transmit/ Receive signal*, DT/$\overline{R}$, from the 8086 sets the direction in which data will pass through the buffers. When DT/$\overline{R}$ is asserted high, the buffers will be set up to transmitdata from the 8086 to ROM, RAM, or ports. When DT/$\overline{R}$ is asserted low, the buffers will be set up to allow data to come into the 8086 from ROM, RAM, or ports.

The buffers used on the data bus must have three-state outputs so the outputs can be floated when the bus is being used for other operations. For example, you certainly do not want data bus buffer outputs enabled onto the data bus while the 8086 is putting out the lower 16 bits of an address on these lines. The 8086 asserts the $\overline{DEN}$ signal to enable the three-state outputs on data bus buffers at the appropriate time in an operation.

The final section of Fig. 2.17*a* to look at is the 8284A clock generator in the upper left corner. This device uses a crystal to produce the stable-frequency clock signal which steps the 8086 through execution of its instructions in an orderly manner. The 8284A also synchronizes the RESET signal and the READY signal with the clock so that these signals are applied to the 8086 at the proper times. When the

RESET input is asserted, the 8086 goes to address FFFF0H to get its next instruction. The first instruction of the system start-up program is usually located at this address, so asserting this signal is a way to *boot*, or start, the system. We will discuss the use of the READY input in the next section.

Now that you have an overview of the basic system connections for an 8086 microcomputer, let's take a look at the signal present on the buses as an 8086 reads data from memory or from a port.

8086 Bus Activities During a Read Machine Cycle

Figure. 2.17*b* shows the signal activities on the 8086 microcomputer buses during simple read and write operations. Don't be overwhelmed by all the lines on this diagram. Their meanings should become clear to you as we work through the diagram.

The first line to look at in Fig. 2.17*b* is the *clock waveform*, CLK, at the top. This represents the crystal-controlled clock signal sent to the 8086 from an external clock generator device such as the 8284 shown in the top left corner of Fig. 2.17*a*. One cycle of this clock is called a *state*. For reference purposes, a state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse. The time interval labeled T_1 in the figure is an example of a state. Different versions of the 8086 have maximum clock frequencies of between 5 MHz and 10 MHz, so the minimum time for one state will be between 100 and 200 ns, depending on the part used and the crystal used.

A basic microprocessor operation such as reading a byte from memory or writing a byte to a port is called a *machine cycle*. The times labeled T_{CY} in Fig. 2.17*b* are examples of machine cycles. As you can see in the figure, a machine cycle consists of several states.

The time a microprocessor requires to fetch and execute an entire instruction is referred to as an *instruction cycle*. An instruction cycle consists of one or more machine cycles.

To summarize this, an instruction cycle is made up of machine cycles, and a machine cycle is made up of states. The time for a state is determined by the frequency of the clock signal. In this section, we will discuss the activities that occur on the 8086 microcomputer buses during a read machine cycle.

The best way to analyze a timing diagram such as the one in Fig. 2.17b is to think of time as a vertical line moving from left to right across the diagram. With this technique, you can easily see the sequence of activities on the signal lines as you move your imaginary time line across the waveforms.

During T_1, of a read machine cycle the 8086 first asserts the M/IO signal. It will assert this signal high if it is going to do a read from memory during this cycle, and it will assert $\overline{M/IO}$ low if it is going to do a read from a port during this cycle. The timing diagram in Fig. 2.17*b* shows two crossed waveforms for the M/IO signal because the signal may be going low or going high for a read cycle. The point where the two waveforms cross indicates the time at which the signal becomes valid for this machine cycle. Likewise, in the rest of the timing diagram, crossed lines are used to represent the time when information on a line or group of lines is changed.

After asserting $\overline{M/IO}$, the 8086 sends out a high on the Address Latch Enable signal (ALE). This signal is connected to the enable input (STB) of the 74S373 octal latches, as shown in Fig. 2.17*a*, so these latches will be enabled when ALE is high. As you can also see in Fig. 2.17*a*, the data inputs of these latches are connected to the 8086 AD0–AD15, A16–A19, and *Bus High Enable* ($\overline{BHE}$) lines. After the 8086 asserts ALE high, it sends out on these lines the address of the memory location that it wants to read. Since the latches are enabled by ALE being high, this address information passes through the latches to their outputs. The 8086 then makes the ALE output low, which disables the latches. The address held on the latch outputs travels along the address bus to memory and port devices.

Note in the timing diagram in Fig. 2.17*b* how the activity on the ADDR/DATA lines is represented. The first point at which the two waveforms cross represents the time at which the 8086 has put a valid address on these lines. These two waveforms *do not* indicate that all 16 lines are going high or going low at this point.

After ALE goes low, the address information is held on the latches, so the 8086 no longer needs to send out the addresses. Therefore, as shown by a dashed line on the ADDR/DATA line in Fig. 2.17*b*, the 8086 floats the AD0–AD15 lines so that they can be used to input data from memory or from a port. At about the same time, the 8086 also removes the $\overline{BHE}$ and A16-A19 information from the upper lines and sends out some status information on those lines.

The 8086 is now ready to read data from the addressed memory location or port, so near the end of state T_2 the 8086 asserts its $\overline{RD}$ signal low. If you trace the connection of the $\overline{RD}$ signal in Fig. 2.17*a*, you should see that this signal is used to enable the addressed memory device or port device. When enabled, the addressed device will put a byte or word of data on the data bus. In other words, asserting the $\overline{RD}$ signal low causes the addressed device to put data on the data bus. This cause-and-effect relationship is shown on the timing diagram in Fig. 2.17*b* by an arrow going from the falling edge of $\overline{RD}$ to the "bus reserved for data in" section of the ADDR/DATA waveforms. The bubble on the tail of the arrow is always put on the signal transition or level that causes some action, and the point of the arrow always indicates the action caused. Arrows of this sort are only used to show the effect a signal from one device will have on another device. They are not usually used to indicate signal cause and effect within a device.

Now, referring to Fig. 2.17*b* again, find the section of the AD0–AD15 waveform marked off as memory access time near the bottom of the diagram. This time represents the time

it takes for the memory to output valid data after it receives an address and an RD signal. If the access time for a memory device is too long, the memory will not have valid data on its outputs soon enough in the machine cycle for the 8086 to receive it correctly. The 8086 will then treat whatever garbage happens to be on the data bus as valid data and go on with the next machine cycle. As long as Murphy's law is still in force, the garbage read in will probably cause the entire program to crash. A section later in the chapter shows you how to calculate whether a particular ROM, RAM, or port device has a short-enough access time to work properly in a given 8086 system. For now, however, we just need you to understand the concept so we can show you one way that an 8086 can accommodate a slow device.

To refresh your memory, look again at the block diagram in Fig. 2.17*a* to find an input on the 8086 CPU labeled READY. When this pin is high, the 8086 is "ready" and operates normally. If the READY input is made low at the right time in a machine cycle, the 8086 will insert one or more WAIT states between T_3 and T_4 in that machine cycle. The read timing diagram in Fig. 2.17*b* shows an example of this. An external hardware device is set up to pulse READY low before the rising edge of the clock in T_2. After the 8086 finishes T_3 of the machine cycle, it enters a WAIT state. During a WAIT state, the signals on the buses remain the same as they were at the start of the WAIT state. The address of the addressed memory location is held on the output of the latches, so it does not change, and as you can see from the timing diagram in Fig. 2.17*b*, the control bus signals, M/$\overline{IO}$ and $\overline{RD}$, also do not change during the WAIT state, T_{WAIT}. The memory or port device then has at least one more clock cycle to get its data output. If the READY input is made high again during T_3 or during the WAIT state, as shown in Fig. 2.17*b*, then after one WAIT state the 8086 will go on with the regular T_4 of the machine cycle.

If the 8086 READY input is still low at the end of a WAIT state, then the 8086 will insert another WAIT state. The 8086 will continue inserting WAIT states until the READY input is made high again.

To summarize, inserting the WAIT state(s) freezes the action on the buses. This gives the addressed device one or more extra clock cycles to put out valid data. As an example of how this is used, we can use slower (cheaper) ROM in a system by adding a simple circuit which pulses the READY input low each time the ROM is addressed. No WAIT states will be inserted in the read machine cycle for reading data from faster devices such as the RAM in the system.

Note in Fig. 2.17a that a READY input signal is usually passed through the 8284A clock generator IC so that the READY signal actually applied to the 8086 is synchronized with the system clock.

Now let us look back at Fig. 2.17*b* to see how $\overline{DEN}$ and DT/$\overline{R}$ function during a read machine cycle. During T_1 of the machine cycle the 8086 asserts DT/$\overline{R}$ low to put the data buffers in the receive mode. Then, after the 8086 finishes using the data bus to send out the lower 16 address bits, it asserts DEN low to enable the data bus buffers. The data put on the data bus by an addressed port or memory will then be able to come in through the buffers to the 8086 on the data bus.

The activities on the 8086 buses during a read machine cycle can be summarized as follows. The 8086 asserts M/$\overline{IO}$ high if the read is to be from memory and asserts M/$\overline{IO}$ low if the read is going to be from a port.

At about the same time, the 8086 asserts ALE high to enable the external address latches. It then sends out $\overline{BHE}$ and the desired address on the AD0-A19 lines. When the 8086 pulls the ALE line low, the address information is latched on the outputs of the external latches. After the 8086 is through using the AD0–AD15 lines for an address, it removes the address from these lines and puts the lines in the input mode (floats them). The 8086 then asserts its $\overline{RD}$ signal low. The $\overline{RD}$ signal going low turns on the addressed memory or port, which then outputs the desired data on the data bus. To complete the cycle, the 8086 brings the $\overline{RD}$ line high again. This causes the addressed memory or port to float its outputs on the data bus. If the 8086 READY input is made low before or during T_2 of a machine cycle, the 8086 will insert WAIT states as long as the READY input is low. When READY is made high, the 8086 will continue with T_4 of the machine cycle. WAIT states can be used to give slow devices additional time to put out valid data. If a system is large enough to need data bus buffers, then the 8086 DT/$\overline{R}$ signal connected to these buffers will set them for input during a read operation or set them for output during a write operation. The 8086 $\overline{DNE}$ signal will enable the buffers at the appropriate time in the machine cycle.

8086 Bus Activities During a Write Machine Cycle

Now that we have analyzed the 8086 bus activities for a read machine cycle, let's take a look at the timing diagram for a write machine cycle in the right-hand side of Fig. 2.17*b*. Most of this diagram should look very familiar to you because it is very similar to that for a read cycle.

During T_1 of a write machine cycle the 8086 asserts M/$\overline{IO}$ low if the write is going to be to a port, and it asserts M/$\overline{IO}$ high if the write is going to be to memory. At about the same time, the 8086 raises ALE high to enable the address latches. The 8086 then outputs $\overline{BHE}$ and the address that it will be writing to on AD0–A19. Incidentally, when writing to a port, lines A16–A19 will always be low, because the 8086 only sends out 16-bit port addresses. After the address has had time to pass through the latches, the 8086 brings ALE low again to latch the address on the outputs of the latches. Besides holding the address, these latches also function as buffers for the address lines. After the address information is latched, the 8086 removes the address

information from AD0–AD15 and outputs the desired data on the data bus. It then asserts its $\overline{WR}$ signal low. The $\overline{WR}$ signal is used to turn on the memory or port that the data is to be written to. After the addressed memory or port has had time to accept the data from the data bus, the 8086 raises the $\overline{WR}$ signal line high again and floats the data bus.

If the memory or port device cannot accept the data word within a normal machine cycle, external hardware can be set up to pulse the READY input low each time that memory or a port device is addressed. If the READY input is pulsed low before or during T_2 of the machine cycle, the 8086 will insert a WAIT state after state T_1. Remember that during WAIT states the signals on the data bus, address bus, and control bus are held constant, so the addressed device has one or more extra clock cycles to accept the data from the data bus. If the READY input is made high before the end of the WAIT state, the 8086 will go on with state T_4 as soon as it finishes the WAIT state. If the READY input is still low just before the end of the WAIT state, the 8086 will insert another WAIT state. It will continue to insert WAIT states until READY is made high. The point here is that the 8086 can be forced to insert as many WAIT states as are necessary for the addressed device to accept the data.

If the system is large enough to need buffers on the data bus, then DT/$\overline{R}$ will be connected to the direction input on the buffers. During a write cycle, the 8086 asserts DT/$\overline{R}$ high to put the buffers in the transmit mode. When the 8086 asserts $\overline{DEN}$ low to enable the buffers, data output from the 8086 will pass through the buffers to the addressed port or memory location.

Work your way across the timing diagrams for the read and write machine cycles in Fig. 2.17*b* until you feel that you understand the sequence of activities that occurs.

Checklist of Important Terms and Concepts in This Chapter

- *8085 microprocessor architecture*
- *TRAP, RST, INTR*
- *Accumulator, flag register, GPRs*
- *Interrupt controller*
- *Machines cycles*
- *8086 architecture*
- *Minimum mode, maximum mode*
- *Segmentation*
- *Bus Interface Unit (BIU)*
- *Instruction byte queue, pipelining, ES, CS, SS, DS registers, IP register*
- *Execution Unit (EU) AX, BX, CX, DX, registers, SP, BP, SI, DI registers*
- *Machine Language, assembly language*
- *Mnemonic, opcode, operand, label, complier, cross assembler*

Review Questions and Problems

*1. What is the reason for giving either manual or power-on RESET to a digital system?

**2. What factors should you consider to estimate the interrupt response time in 8085 microprocessor?

**3. Do you really feel S0 and S! are required to design an 8085 based computer system and in what way can you use them?

**4. Can you give one application where you can use SID and SOD pins in 8085?

**5. To what use are the signals S0/, S!/, S2/ put in 8086?

*6. Describe the sequence of signals that occurs on the address bus, the control bus, and the data bus when a simple microcomputer fetches an instruction.

*7. What determines whether a microprocessor is considered an 8-bit, a 16-bit, or a 32-bit device?

*8. (a) How many address lines does an 8086 have?
(b) How many memory addresses does this number of address lines allow the 8086 to access directly?
(c) At any given time, the 8086 works with four segments in this address space. How many bytes are contained in each segment?

*9. What is the main difference between the 8086 and the 8088?

**10. (a) Describe the function of the 8086 queue.
(b) How does the queue speed up processing?

**11. (a) If the code segment for an 8086 program starts at address 70400H, what number will be in the CS register?
(b) Assuming this same code segment base, what physical address will a code byte be fetched from if the instruction pointer contains 539CH?

***12. What physical address is represented by?
(a) 4370:561EH
(b) 7A32:0028H

**13. What is the advantage of using a CPU register for temporary data storage over using a memory location?

**14. If the stack segment register contains 3000H and the stack pointer register contains 8434H, What is the physical address of the tip of the stack?

**15. (a) What is the advantage of using assembly language instead of writing a program directly in machine language?
(b) Describe the operation an 8086 will perform when it executes ADD AX, BX.

***16. What type of programs are usually written in assembly language?