موجود في مكتبة الكلية
(عند موظف المكتب)

MICROPROCESSORS AND MICROCOMPUTER-BASED SYSTEM DESIGN

1990

Mohamed Rafiquzzaman, Ph.D.
Professor
California State Polytechnic University
Pomona, California
and
Adjunct Professor
University of Southern California
Los Angeles, California

UNIVERSAL BOOK STALL
NEW DELHI

1.jpg

Chapter 3

INTEL 8086

This chapter describes the internal architecture, addressing modes, instruction set, and I/O techniques associated with the 8086 microprocessor. Interfacing capabilities to typical memory and I/O chips such as the 2716, 2142, and 8255 are included.

A design technique is presented showing interconnection of the 8086 to 2716 EPROM, 2142 RAM, and 8255 I/O chips. The memory and I/O maps are then determined.

3.1 INTRODUCTION

The 8086 is Intel's first 16-bit microprocessor. Its design is based on the 8080 but it is not directly compatible with the 8080.

The 8086 is designed using the HMOS technology and contains approximately 29,000 transistors. The 8086 is packaged in a 40-pin DIP (Dual In-line Package) and requires a single 5V power supply. The 8086 can be operated at three different clock speeds. The standard 8086 runs at 5 MHz internal clock frequency, whereas the 8086-2 and 8086-4 run at internal clock frequencies of 8 and 4 MHz, respectively. An external clock generator/driver chip such as the Intel 8284 is needed to generate the 8086 clock input signal. The 8086 divides the external clock connected at the CLK pin internally by three. This means that for a 5-MHz internal clock, the 8284 must generate a 15-MHz clock output which should be connected at the 8086 CLK pin.

The 8086 has a 20-bit address and, hence, it can directly address up to one megabyte (2^{20}) of memory. The 8086 uses a segmented memory. An interesting feature of the 8086 is that it prefetches up to six instruction

Queue

42.jpg

bytes from memory and queues them in order to speed up instruction execution.

The memory of an 8086-based microcomputer is organized as bytes. Each byte can be uniquely addressed with 20-bit addresses of 00000_{16}, 00001_{16}, 00002_{16}, ...$FFFFF_{16}$. An 8086 16-bit word consists of any two consecutive bytes; the low-addressed byte is the low byte of the word and the high-addressed byte contains the high byte as follows:

Low byte of the word	High byte of the word
07_{16}	26_{16}
Address 00520_{16}	Address 00521_{16}

The 16-bit word stored at the even address 00520_{16} is 2607_{16}.

Next consider a word stored at an odd address as follows:

Low byte of the word	High byte of the word
05_{16}	$3F_{16}$
Address 01257_{16}	Address 01258_{16}

The 16-bit word stored at the odd address 01257_{16} is $3F05_{16}$. Note that for word addresses, the programmer uses the low-order address (odd or even) to specify the whole 16-bit word.

The 8086 always accesses a 16-bit word to or from memory. The 8086 can read a 16-bit word in one operation if the first byte of the word is at an even address. On the other hand, the 8086 must perform two memory accesses to two consecutive memory even addresses, if the first byte of the word is at an odd address. In this case, the 8086 discards the unwanted bytes of each. For example, consider MOV BX, ADDR. Note that the X or H (or L) following the 8086 register name in an instruction indicates whether the transfer is 16-bit or 8-bit. For example, MOV ADDR, BX moves the contents of the 20-bit physical memory location addressed by ADDR into the 8086 16-bit register BX. Now, if ADDR is a 20-bit even address such as 30024_{16}, then this MOV instruction loads the low (BL) and high (BH) bytes of the 8086 16-bit register BX with the contents of memory locations 30024_{16} and 30025_{16}, respectively, in a single access. Now, if ADDR is an odd address such as 40005_{16}, then the MOV BX, ADDR instruction loads BL and BH with the contents of memory locations 40005_{16} and 40006_{16}, respectively, in two accesses. Note that the

8086 accesses locations 40004_{16} and 40005_{16} in the first operation but discards the contents of 40004_{16}, and in the second operation accesses 40006_{16} and 40007_{16} but ignores the contents of 40007_{16}.

Next, consider a byte move such as MOV BH, ADDR. If ADDR is an even address such as 50002_{16}, then this MOV instruction accesses both 50002_{16} and 50003_{16}, but loads BH with the contents of 50002_{16} and ignores the contents of 50003_{16}. However, if ADDR is an odd address such as 50003_{16}, then this MOV loads BH with the contents of 50003_{16} and discards the contents of 50002_{16}.

The 8086 family consists of two types of 16-bit microprocessors — the 8086 and 8088. The main difference is how the processors communicate with the outside world. The 8088 has an 8-bit external data path to memory and I/O, while the 8086 has a 16-bit external data path. This means that the 8088 will have to do two read operations to read a 16-bit word into memory. In most other respects, the processors are identical. Note that the 8088 accesses memory in bytes. No alterations are needed to run software written for one microprocessor on the other. Because of similarities, only the 8086 will be considered here. The 8088 is used in designing the IBM Personal computer.

An 8086 can be configured as a small uniprocessor system (minimum mode if the MN/$\overline{\text{MX}}$ pin is tied to HIGH) or as a multiprocessor system (maximum mode when MN/$\overline{\text{MX}}$ pin is tied to LOW). In a given system, the MN/$\overline{\text{MX}}$ pin is permanently tied to either HIGH or LOW. Some of the 8086 pins have dual functions depending on the selection of the MN/$\overline{\text{MX}}$ pin level. In the minimum mode (MN/$\overline{\text{MX}}$ pin high), these pins transfer control signals directly to memory and input/output devices. In the maximum mode (MN/$\overline{\text{MX}}$ pin low), these same pins have different functions which facilitate multiprocessor systems. In the maximum mode, the control functions normally present in minimum mode are assumed by a support chip, the 8288 bus controller.

Due to technological advances, Intel introduced the high performance 80186 and 80188 which are enhanced versions of the 8086 and 8088, respectively. The 8-MHz 80186/80188 provides two times greater throughput than the standard 5-MHz 8086/8088. Both have integrated several new peripheral functional units such as a DMA controller, a 16-bit timer unit, and an interrupt controller unit into a single chip. Just like the 8086 and 8088, the 80186 has a 16-bit data bus and the 80188 has an 8-bit data bus; otherwise, the architecture and instruction set of the 80186 and 80188 are identical. The 80186/80188 has an on-chip clock generator so that only an external crystal is required to generate the clock. The 80186/80188 can operate at either 6 or 8 MHz internal clock. Like the 8085, the crystal frequency is divided by 2 internally. In other words, external crystals

of 12 or 16 MHz must be connected to generate the 6- or 8-MHz internal clock frequency. The 80186/80188 is fabricated in a 68-pin package. Both processors have on-chip priority interrupt controller circuits to provide five interrupt pins. Like the 8086/8088, the 80186/80188 can directly address one megabyte of memory. The 80186/80188 is provided with 10 new instructions beyond the 8086/8088 instruction set. Examples of these instructions include INs and OUTs for inputting and outputting string byte or string word. The 80286, on the other hand, has added memory protection and management capabilities to the basic 8086 architecture. An 8-MHz 80286 provides up to six times greater throughput than the 5-MHz 8086. The 80286 is fabricated in a 68-pin package. The 80286 can be operated at 4, 6, or 8 MHz clock frequency. An external 82284 clock generator chip is required to generate the clock. The 80286 divides the external clock by two to generate the internal clock. The 80286 is typically used in a multiuser or multitasking system. The 80286 can be operated in two modes, real address and protected virtual address. The real address mode emulates a very high performance 8086. In this mode, the 80286 can directly address one megabyte of memory. The 80286, in the virtual address mode, can directly address 16 megabytes of memory. The virtual address mode provides (in addition to the real address mode capabilities) virtual memory management, task management, and protection. The programmer can select one of these modes by loading appropriate data in the 16-bit machine status word (MSW) register by using load and store instructions. Two examples of these instructions are LMSW (Load MSW register) and SMSW (Store MSW register). The 80286 is used as the CPU of the IBM PC/AT Personal computer. An enhanced version of the 80286 is at the 32-bit 80386 microprocessor which will be covered later. The 80386 is used as the CPU in the IBM's newest PC.

3.2 8086 ARCHITECTURE

Figure 3.1 shows a block diagram of the 8086 internal architecture. As shown in the figure, the 8086 microprocessor is internally divided into two separate functional units. These are the Bus Interface Unit (BIU) and the Execution Unit (EU). The BIU fetches instructions, reads data from memory and ports, and writes data to memory and I/O ports. The EU executes instructions that have already been fetched by the BIU. The BIU and EU function independently. The BIU interfaces the 8086 to the

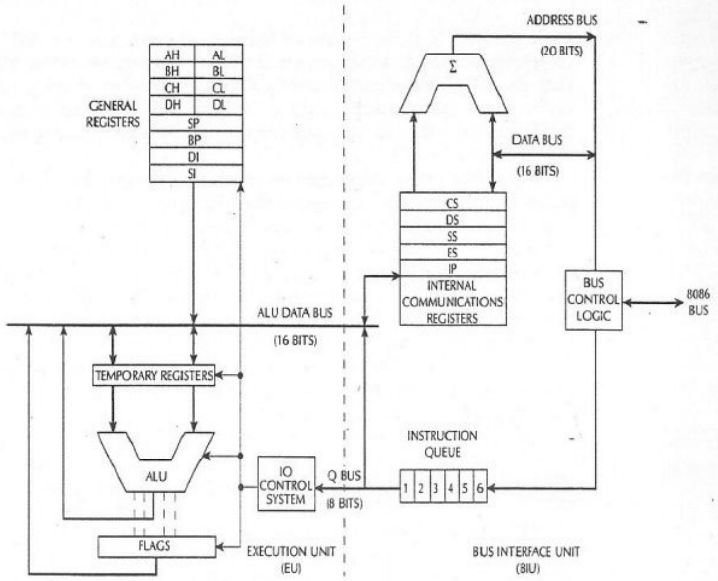


FIGURE 3.1 Internal architecture of the 8086.

outside world. The BIU provides all external bus operations. The BIU contains segment registers, instruction pointer, instruction queue, and address generation/bus control circuitry to provide functions such as fetching and queuing of instructions, and bus control.

The BIU's instruction queue is a First-In First-Out (FIFO) group of registers in which up to six bytes of instruction code are prefetched from memory ahead of time. This is done in order to speed up program execution by overlapping instruction fetch with execution. This mechanism is known as pipelining. If the queue is full and the EU does not request BIU to access memory, the BIU does not perform any bus cycle. On the other hand, if the BIU is not full and if it can store at least two bytes and the EU does not request it to access memory, the BIU may prefetch instructions. However, if BIU is interrupted by EU for memory access

while the BIU is in the process of fetching an instruction, the BIU first completes fetching and then services the EU: the queue allows the BIU to keep the EU supplied with prefetched instructions without tying up the system bus. If an instruction such as Jump or subroutine call is encountered, the BIU will reset the queue and begin refilling after passing the new instruction to the EU.

The BIU contains a dedicated adder which is used to produce the 20-bit address. The bus control logic of the BIU generates all the bus control signals such as read and write signals for memory and I/O.

The BIU has four 16-bit segment registers. These are the Code Segment (CS) register, the Data Segment (DS) register, the Stack Segment (SS) register, and the Extra Segment (ES) register. The 8086's one megabyte memory is divided into segments of up to 64K bytes each. The 8086 can directly address four segments (256K byte within the 1 Mbyte memory) at a particular time. Programs obtain access to code and data in the segments by changing the segment register contents to point to the desired segments. All program instructions must be located in main memory pointed to by the 16-bit CS register with a 16-bit offset in the segment contained in the 16-bit instruction pointer (IP). The BIU computes the 20-bit physical address internally using the programmer-provided logical address (16-bit contents of CS and IP) by logically shifting the contents of CS four bits to left and then adding the 16-bit contents of IP. In other words, the CS is multiplied by 16_{10} by the BIU for computing the 20-bit physical address. This means that all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the 16-bit contents of IP. For example, if [CS] = $456A_{16}$ and [IP] = 1620_{16}, then the 20-bit physical address is generated by the BIU as follows:

Four times logically shifted [CS] to left	= $456A0_{16}$
+ [IP] as offset	= 1620_{16}
20-bit physical address	= $46CC0_{16}$

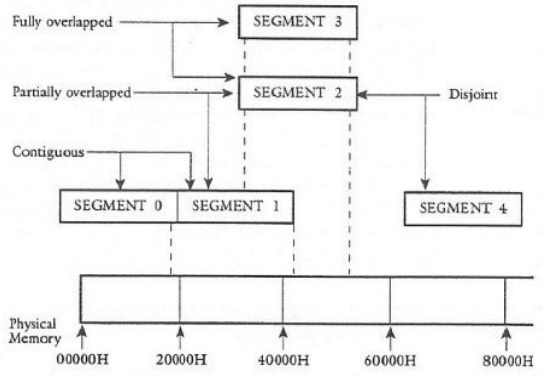
The BIU always inserts four zeros for the lowest 4-bits of the 20-bit starting address (physical) of a segment. In other words, the CS contains the base or start of the current code segment, and IP contains the distance or offset from this address to the next instruction byte to be fetched. Note that immediate data are considered as part of the code segment.

The SS register points to the current stack. The 20-bit physical stack address is calculated from SS and SP for stack instructions such as PUSH and POP. The programmer can use the BP register instead of SP for accessing the stack using the based addressing mode. In this case, the 20-bit physical stack address is calculated from BP and SS.

The DS register points to the current data segment; operands for most instructions are fetched from this segment. The 16-bit contents of Source Index (SI) or Destination Index (DI) are used as offset for computing the 20-bit physical address.

The ES register points to the extra segment in which data (in excess of 64K pointed to by DS) is stored. String instructions always use ES and DI to determine the 20-bit physical address for the destination.

The segments can be continuous, partially overlapped, fully overlapped, or disjoint. An example of how five segments (segment 0 through segment 4) may be stored in physical memory are shown below:



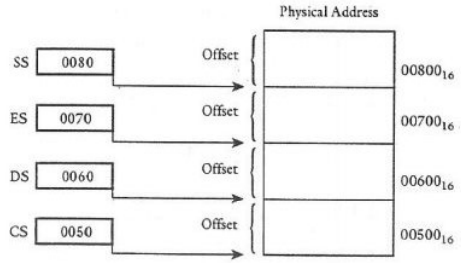
In the above SEGMENTs 0 and 1 are contiguous (adjacent), SEGMENTs 1 and 2 are partially overlapped, SEGMENTs 2 and 3 are fully overlapped, and SEGMENTs 2 and 4 are disjoint. Every segment must start on 16-byte memory boundaries.

Typical examples of values of segments should then be selected based on physical addresses starting at 00000_{16}, 00010_{16}, 00020_{16}, 00030_{16}, ... $FFFF0_{16}$. A physical memory location may be mapped into (contained in) one or more logical segments. Many applications can be written to simply initialize the segment registers and then forget them.

45.jpg

A segment can be pointed to by more than one segment register. For example, DS and ES may point to the same segment in memory if a string located in that segment is used as a source segment in one string instruction and as a destination segment in another string instruction. Note that for string instructions, a destination segment must be pointed to by ES.

It should be pointed out that codes should not be written within 6 bytes of the end of physical memory. Failure to comply with this guideline may result in an attempted op code fetch from nonexistent memory, hanging the CPU if READY is not returned. One example of four currently addressable segments is shown below:



The EU decodes and executes instructions. A decoder in the EU control system translates instructions. The EU has a 16-bit ALU for performing arithmetic and logic operations.

The EU has eight 16-bit general registers. These are AX, BX, CX, DX, SP, BP, SI, and DI. The 16-bit registers AX, BX, CX, and DX can be used as two 8-bit registers (AH, AL, BH, BL, CH, CL, DH, DL). For example, the 16-bit register DX can be considered as two 8-bit registers DH (high byte of DX) and DL (low byte of DX). The general-purpose registers AX, BX, CX, and DX are named after special functions carried out by each one of them. For example, the AX is called the 16-bit accumulator while the AL is the 8-bit accumulator. The use of accumulator registers is assumed by some instructions. The Input/Output (IN or OUT) instructions always use AX or AL for inputting/outputting 16- or 8-bit data to or from an I/O port.

Multiplication and division instructions also use AX or AL. The AL register is the same as the 8085 A register.

BX register is called the base register. This is the only general-purpose register, the contents of which can be used for addressing 8086 memory.

All memory references utilizing these register contents for addressing use DS as the default segment register. The BX register is similar to 8085 HL register. In other words, 8086 BH and BL are equivalent to 8085 H and L registers, respectively.

The CX register is known as the counter register. This is because some instructions such as shift, rotate, and loop instructions use the contents of CX as a counter. For example, the instruction LOOP START will automatically decrement CX by 1 without affecting flags and will check if [CX] = 0. If it is zero, the 8086 executes the next instruction; otherwise the 8086 branches to the label START.

The data register DX is used to hold high 16-bit result (data) in 16 × 16 multiplication or high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit remainder after the division.

The two pointer registers, SP (stack pointer) and BP (base pointer), are used to access data in stack segment. The SP is used as an offset from the current SS during execution of instructions that involve stack segment in external memory. The SP contents are automatically updated (incremented or decremented) due to execution of POP or PUSH instruction.

The base pointer contains an offset address in the current SS. This offset is used by the instructions utilizing the based addressing mode.

The FLAG register in the EU holds the status flags typically after an ALU operation. Figure 3.2 shows the 8086 registers.

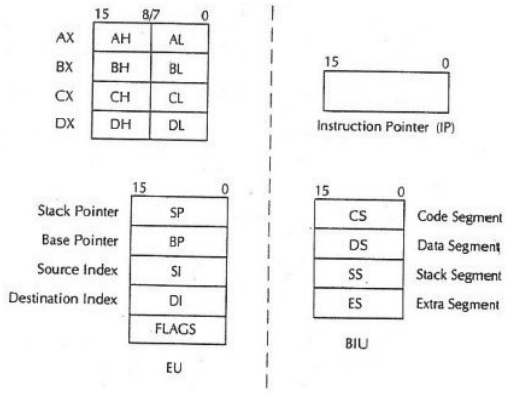


FIGURE 3.2 8086 registers.

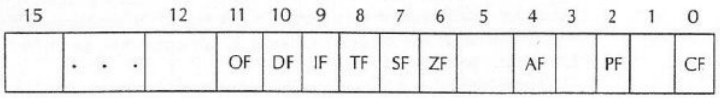


FIGURE 3.3 8086 flag register.

The 8086 has six one-bit flags. Figure 3.3 shows the flag register. Let us explain the 8086 flags. AF (Auxiliary carry Flag) is used by BCD arithmetic instructions. AF = 1 if there is a carry from the low nibble (4-bit) into the high nibble or a borrow from the high nibble into the low nibble of the low-order 8-bit of a 16-bit number. CF (Carry Flag) is set if there is a carry from addition or borrow from subtraction. OF (Overflow Flag) is set if there is an arithmetic overflow, that is, if the size of the result exceeds the capacity of the destination location. An interrupt on overflow instruction is available which will generate an interrupt in this situation. SF (Sign Flag) is set if the most significant bit of the result is one (negative) and is cleared to zero for non-negative result. PF (Parity Flag) is set if the result has even parity; PF is zero for odd parity of the result. ZF (Zero Flag) is set if result is zero; ZF is zero for nonzero result.

The 8086 has three control bits in the flag register which can be set or reset by the programmer: setting DF (Direction Flag) to one causes string instructions to autodecrement and clearing DF to zero causes string instructions to autoincrement. Setting IF (Interrupt Flag) to one causes the 8086 to recognize external maskable interrupts; clearing IF to zero disables these interrupts. Setting TF (Trace Flag) to one places the 8086 in the single-step mode. In this mode, the 8086 generates an internal interrupt after execution of each instruction. The user can write a service routine at the interrupt address vector to display the desired registers and memory locations. The user can thus debug a program.

3.3 8086 ADDRESSING MODES

The 8086 has 12 basic addressing modes. The various 8086 addressing modes can be classified into five groups:

1. Addressing modes for accessing immediate and register data (register and immediate modes)
2. Addressing modes for accessing data in memory (memory modes)
3. Addressing modes for accessing I/O ports (I/O modes)
4. Relative addressing mode
5. Implied addressing mode

3.3.1 ADDRESSING MODES FOR ACCESSING IMMEDIATE AND REGISTER DATA (REGISTER AND IMMEDIATE MODES)

3.3.1.a Register Addressing Mode

This mode specifies the source operand, destination operand, or both to be contained in an 8086 register. An example is MOV DX, CX which moves the 16-bit contents of CX into DX. Note that in the above both source and destination operands are in register mode. Another example is MOV CL, DL which moves 8-bit contents of DL into CL. MOV BX, CH is an illegal instruction; the register sizes must be the same.

3.3.1.b Immediate Addressing Mode

In immediate mode, 8- or 16-bit data can be specified as part of the instruction. For example, MOV CL, 03H moves the 8-bit data 03H into CL. On the other hand, MOV DX, 0502H moves the 16-bit data 0502H into DX. Note that in both of the above MOV instructions, the source operand is in immediate mode and the destination operand is in register mode.

A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H. An 8086 instruction with immediate mode such as MOV BH, VALUE can then be used to load 35H into BH. Note that even though the immediate mode specifies data with the instruction, these immediate data must be located in the memory addressed by the 8086 CS and IP registers. This is because these data are considered as part of the instruction.

3.3.2 ADDRESSING MODES FOR ACCESSING DATA IN MEMORY (MEMORY MODES)

As mentioned before, the Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However, the EU cannot directly access the memory operands. It must use the BIU in order to access memory operands. For example, when the EU needs to access a memory location, it sends an offset value to the BIU. This offset is also called the Effective Address (EA). Note that EA is displacement of the

desired location from the segment base. As mentioned before, the BIU generates a 20-bit physical address after shifting the contents of the desired segment register four bits to the left and then adding the 16-bit EA to it. The 8086 must use a segment register whenever it accesses the memory. Also, every memory addressing instruction uses an Intel-defined standard default segment register. However, a segment override prefix can be placed before most of the memory addressing instructions whose default segment register is to be overridden. For example, INC BYTE PTR [START] will increment the 8-bit content of memory location in DS with offset START by one. However, segment DS can be overridden by ES as follows: INC ES: BYTE PTR [START]; segments cannot be overridden for stack reference instructions (such as PUSH and POP). Destination segment of a string segment which must be ES (if a prefix is used with string instruction, only the source segment DS can be overridden) cannot be overridden. The Code Segment (CS) register used in program memory addressing cannot be overridden. There are six modes in this category. These are

a. Direct addressing mode
b. Register indirect addressing mode
c. Based addressing mode
d. Indexed addressing mode
e. Based indexed addressing mode
f. String addressing mode

3.3.2.a Direct Addressing Mode

In this mode, the 16-bit effective address (EA) is taken directly from the displacement field of the instruction. The displacement (unsigned 16-bit or sign-extended 8-bit number) is stored in the location following the Instruction op code. This EA or displacement is the distance of the memory location from the current value in the data segment (DS) register in which the data are stored. The BIU shifts the [DS] four times to left and adds the EA to generate the 20-bit physical address. The register can be overridden by the programmer by using a segment override instruction to tell the BIU to add EU to some other segment base for producing the 20-bit physical address. As an example of 16-bit read from memory, consider MOV CX, START. This instruction moves the contents of memory location which is offset by START from the current DS value into register CX.

Note that in the above instruction, the source is in direct addressing mode. If the 16-bit value assigned to the offset START by the programmer using an assembler pseudoinstruction such as DW is 0040_{16} and [DS] = 3050_{16}, then the BIU generates the 20-bit physical address 30540_{16} on the

8086 address pins and then initiates a memory read cycle to read the 16-bit data from memory location starting at 30540_{16} location. The memory logic places the 16-bit contents of locations 30540_{16} and 30541_{16} on the 8086 data pins. The BIU transfers these data to the EU; the EU then moves these data to CX; [30540_{16}] to CL and [30541_{16}] to CH. Now, for 8-bit read from memory consider MOV CH, START. If [DS] = 3050_{16}, the value of START is 0040_{16}, then the 8-bit content of memory location 30540_{16} is moved to register CH. An example of 16-bit write to memory is MOV START, BX with [DS] = 3050_{16}. If the value of START is 0040_{16}, then this MOV instruction moves [BL] and [BH] to locations 30540_{16} and 30541_{16}, respectively. Similarly, an example of 8-bit data write is MOV START, BL. The 8086 does not have a MOVE instruction for moving 8-bit or 16-bit data from one memory location to another in which both source and–destination operands are in direct addressing mode. String addressing mode (to be discussed later) allows string memory operation such as block move from source memory array to a destination memory array. Note that START in the above can be defined as an address by using the assembler DB (Define Byte) or DW (Define Word) pseudoinstructions.

3.3.2.b Register Indirect Addressing Mode

In this mode, the EA is specified in either a pointer register or an index register. The pointer register can be either base register BX or base pointer register BP and index register can be either Source Index (SI) register or Destination Index (DI) register. The 20-bit physical address is computed using DS and EA. For example, consider MOV [DI], BX. The destination operand of the above instruction is in register indirect mode, while the source operand is in register mode. The instruction moves the 16-bit content of BX into a memory location offset by the value of EA specified in DI from the current contents in DS. Now, if [DS] = 5004_{16}, [DI] = 0020_{16}, and [BX] = 2456_{16}, then after MOV [DI], BX, content of BX (2456_{16}) is moved to memory locations 50060_{16} and 50061_{16}.

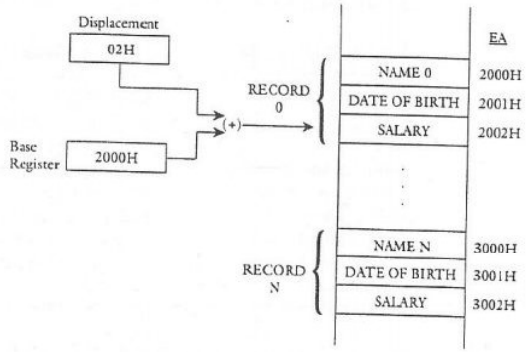
Using this mode, one instruction can operate on many different memory locations if the value in the base or index register is updated. The LEA (Load Effective Address) might be used to change the register value.

3.3.2.c Based Addressing Mode

In this mode, EA is obtained by adding a displacement (signed 8-bit or unsigned 16-bit) value to the contents of BX or BP. The segment registers used are DS and SS. When memory is accessed, the 20-bit physical address is computed from BX and DS. On the other hand, when the stack is accessed, the 20-bit physical address is computed from BP and SS. This

allows the programmer to access the stack without changing the SP contents. As an example of this mode, consider MOV AL, START [BX]. Note that some assemblers use MOV AL, [BX + START] rather than MOV AL, START [BX]. The source operand in the above instruction is in based mode. EA is obtained by adding the value of START and [BX]. The 20-bit physical address is produced from DS and EA. The 8-bit content of this memory location is moved to AL. The displacement START can be either unsigned 16-bit or signed 8-bit. However, a byte is saved for the machine code representation of the instruction if 8-bit displacement is used. The 8086 sign-extends the 8-bit displacement and then adds it to [BX] in the above MOV instruction for determining EA. On the other hand, the 8086 adds an unsigned 16-bit displacement directly with [BX] for determining EA.

Based addressing provides a convenient way to address structure which may be stored at different places in memory:



For example, the element salary in record 0 of the employee NAME 0 can be loaded into an 8086 internal register such as AL using the instruction MOV AL, ALPHA [BX], where ALPHA is the 8-bit displacement 02H and BX contains the starting address of the record 0. Now, in order to access the salary of RECORD N, the programmer simply changes the contents of the base register to 3000H.

If BP is specified as a base register in an instruction, the 8086 automatically obtains the operand from the current SS (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data. BP can be used as a stack pointer in SS to access local variables. Consider the following instruction sequence:


```
PUSH BP              ; Save BP
MOV BP, SP           ; Establish BP
PUSH DX              ; Save
PUSH AX              ; registers
SUB SP, 4            ; Allocate 2 words of
                     ; stack for accessing stack
MOV [BP - 6], AX     ; Arbitrary instructions for
MOV [BP - 8], BX     ; accessing stack data using BP
ADD SP, 4            ; Deallocate storage
POP AX               ; Restore
POP DX               ; all registers
POP BP               ; that were pushed before
```

This instruction sequence is arbitrarily chosen to illustrate the use of BP for accessing the stack.

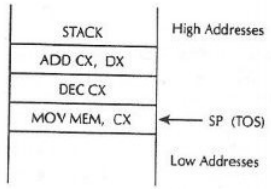
Figure 3.4 shows the 8086 stack during various stages. Figure 3.4A shows the stack before execution of the instruction sequence.

The instruction sequence from PUSH BP to SUB SP, 4 pushes BP, DX, and AX and then subtracts 4 from SP, and this allocates 2 words of the stack. The stack at this point is shown in Figure 3.4B. Note that in 8086, SP is decremented by 2 for PUSH and incremented by 2 for POP. The [BP] is not affected by PUSH or POP. The instruction sequence MOV [BP – 6], AX saves AX in the stack location addressed by [BP – 6] in SS. The instruction MOV [BP – 8], BX writes the [BX] into the stack location [BP – 8] in SS. These instructions are arbitrarily chosen to illustrate how BP can be used to access the stack. These two local variables can be accessed by the subroutine using BP. The instruction ADD SP, 4 releases two words of the allocated stack. The stack at this point is shown in Figure 3.4C. The last three POP instructions restore the contents of AX, DX, and BP to their original values and return the stack as it was before the instruction sequence was executed. This shown in Figure 3.4D.

3.3.2.d Indexed Addressing Mode

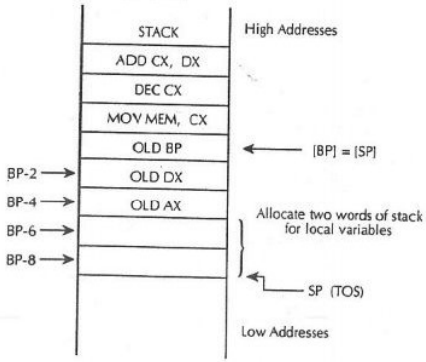
In this mode, the effective address is calculated by adding the unsigned 16-bit or sign-extended 8-bit displacement and the contents of SI or DI.

As an example, MOV BH, START [SI] moves the contents of the 20-bit address computed from the displacement START, SI and DS into BH.



A) Stack before executing the instruction sequence.

(Assumed)

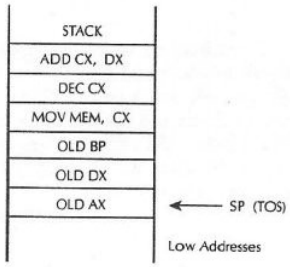


B) Stack after execution of the instruction sequence from PUSH BP to SUB SP, 4

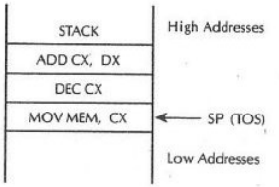
FIGURE 3.4 Accessing stack using BP.

The 8-bit displacement is provided by the programmer using the assembler pseudoinstruction such as EQU. For 16-bit displacement, the EU adds this to SI to determine EA. On the other hand, for 8-bit displacement the EU sign-extends it to 16 bits and then adds to SI for determining EA.

Note that for both based and indexed modes, the 8086 obtains the Effective Address (EA) by using the equation EA = RA + M, where RA is the Reference Address and M is the Modifier. In case of based addressing mode, RA is held in a register such as BX or BP and the modifier M is unsigned 16-bit or signed 8-bit displacement and is included as part of the instruction. The based mode is useful in accessing segmented memory in



C) Stack after execution of ADD SP, 4



D) Stack before execution of POP BP.

FIGURE 3.4 (continued)

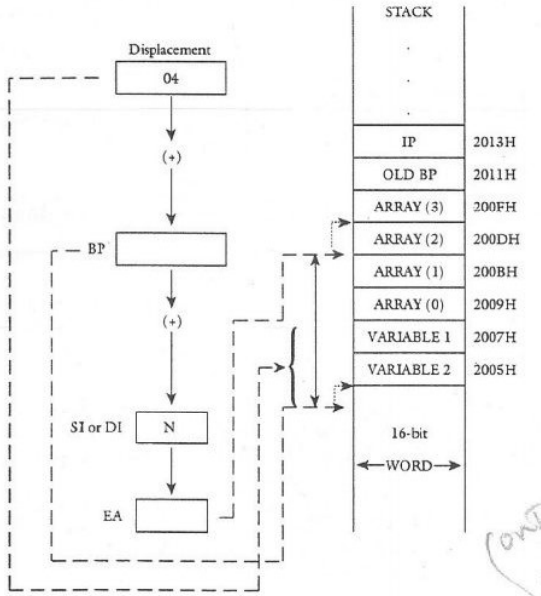
which BX or BP holds the base address of a segment. In indexed addressing mode, on the other hand, RA is included as part of the instruction and the 8086 register SI or DI contains the value of M. RA can be unsigned 16-bit or sign-extended 8-bit.

Indexed addressing mode can be used to access a single table. The displacement can be the starting address of the table. The content of SI or DI can then be used as an index from the starting address to access a particular element in the table.

3.3.2.e Based Indexed Addressing Mode

In this mode, the EA is computed by adding a base register (BX or BP), an index register (SI or DI), and a displacement (unsigned 16-bit or sign-extended 8-bit). As an example, consider MOV ALPHA [SI] [BX], CL if [BX] = 0200H, value of ALPHA = 08H, [SI] = 1000H, and [DS] = 3000 H, then 8-bit content of CL is moved to 20-bit physical address 31208_{16}.

Based indexed addressing mode provides a convenient way for a subroutine to address an array allocated on a stack. Register BP can be loaded with the offset in segment SS (top of the stack after the subroutine has saved registers and allocated local storage). The displacement can be the value which is the difference between the top of the stack and the beginning of the array. An index register can then be used to access individual array elements as follows:



In the above, [BP] = top of the stack = 2005H; displacement = difference between the top of the stack and start of the array = 04H; [SI or DI] = N = 16-bit number (0, 2, 4, 6 in the example). As an example, the instruction MOV DX, 4 [SI] [BP] with [SI] = 6 will read the array (3) which is the content of 200FH in SS into DX. Since in the based indexed mode, the contents of two registers such as BX and SI can be varied, two-dimensional arrays such as matrices can also be accessed.

3.3.2.f String Addressing Mode

This mode uses index registers. The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) or decremented (by setting DF to 1 by STD instruction) to point to the next byte or word. The segment register for the source is DS and may be overridden.

The segment register for the destination must be ES and cannot be overridden. As an example, consider MOVS BYTE. If [DF] = 0, [DS] = 2000_{16}, [SI] = 0500_{16}, [ES] = 4000_{16}, [DI] = 0300_{16}, [20500]$_{16}$ = 38_{16}, and [40300$_{16}$] = 45_{16}, then after execution of the MOVS BYTE, [40300$_{16}$] = 38_{16}, [SI] = 0501_{16}, and [DI] = 0301_{16}. The contents of other registers and memory locations are unchanged. Note that SI and DI can be used in either the source or destination operand of a two-operand instruction, except for string instructions in which SI points to the source (may be overridden) and DI must point to the destination.

3.3.3 ADDRESSING MODES FOR ACCESSING I/O PORTS (I/O MODES)

Standard I/O uses port addressing modes. For memory-mapped I/O, memory addressing modes are used. There are two types of port addressing modes: direct and indirect.

In direct port mode, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0 to 255. For example, OUT 05H, AL outputs [AL] to 8-bit port 05H. In indirect port mode, the port number is taken from DX allowing 64K 8-bit ports or 32K 16-bit ports. For example, if [DX] = 5040_{16}, then IN AL, DX inputs the 8-bit content of port 5040_{16} into AL. On the other hand, IN AX, DX inputs the 8-bit contents of ports 5040_{16} and 5041_{16} into AL and AH, respectively. Note that 8-bit and 16-bit I/O transfers must take place via AL and AX, respectively.

3.3.4 RELATIVE ADDRESSING MODE

Instructions using this mode specify the operand as a signed 8-bit displacement relative to PC. An example is JNC START. This instruction means that if carry = 0, then PC is loaded with current PC contents plus the 8-bit signed value of START; otherwise the next instruction is executed.

3.3.5 IMPLIED ADDRESSING MODE

Instructions using this mode have no operands. An example is CLC which clears the carry flag to zero.

3.4 8086 INSTRUCTION SET

The 8086 instruction set includes equivalents of the 8085 instructions plus many new ones. The new instructions contain operations such as signed and unsigned multiplication and division, bit manipulation instructions, string instructions, and interrupt instructions.

The 8086 has approximately 117 different instructions with about 300 op codes. The 8086 instruction set contains no operand, single operand, and two operand instructions. Except for string instructions which involve array operations, the 8086 instructions do not permit memory-to-memory operations. Table 3.1 lists a summary of 8086 instructions in alphabetical order. Tables 3.A-1 through 3.A-12 (supplied at the end of the chapter) provide a detailed description of the 8086 instructions.

TABLE 3.1
Summary of 8086 Instructions

Instructions	Interpretation	Comments
AAA	ASCII adjust [AL] after addition	This instruction has implied addressing mode; this instruction is used to adjust the content of AL after addition of two ASCII characters
AAD	ASCII adjust for division	This instruction has implied addressing mode; converts two unpacked BCD digits in AX into equivalent binary numbers in AL; AAD must be used before dividing two unpacked BCD digits by an unpacked BCD byte
AAM	ASCII adjust after multiplication	This instruction has implied addressing mode; after multiplying two unpacked BCD numbers, adjust the product in AX to become an unpacked BCD result; ZF, SF, and PF are affected
AAS	ASCII adjust [AL] after subtraction	This instruction has implied addressing mode used to adjust [AL] after subtraction of two ASCII characters
ADC mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] + [mem/reg 2] + CY	Memory or register can be 8- or 16-bit; all flags are affected; no segment registers are allowed; no memory-to-memory ADC is permitted
ADC mem, data	[mem] ← [mem] + data + CY	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
ADC reg, data	[reg] ← [reg] + data + CY	Data can be 8- or 16-bit; register cannot be segment register; all flags are affected; reg is not usually used as AX or AL
ADC A, data	[A] ← [A] + data + CY	'A' can be AL or AX register; all flags are affected
ADD mem/reg 2, mem/reg 1	[mem/reg 1] ← [mem/reg 2] + [mem/reg 1]	Add two 8- or 16-bit data; no memory-to-memory ADD is permitted; all flags are affected; mem uses DS as the segment register; reg 1 or reg 2 cannot be segment register
ADD mem, data	[mem] ← [mem] + data	Mem uses DS as the segment register; data can be 8- or 16-bit; all flags are affected
ADD reg, data	[reg] ← [reg] + data	Data can be 8- or 16-bit; no segment registers are allowed; all flags are affected; this instruction should not be used to add AL or AX with 8- or 16- bit immediate data
ADD A, data	[A] ← [A] + data	Data can be 8- or 16-bit; 'A' can be AL or AX; all flags are affected
AND mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] ∧ [mem/reg 2]	This instruction logically ANDS 8- or 16-bit data in [mem/reg 1] with 8- or 16-bit data in [mem/reg 2]; all flags are affected; OF and CF are cleared to zero; no segment registers are allowed; no memory-to-memory operation is allowed; mem uses DS as the segment register
AND mem, data	[mem] ← [mem] ∧ data	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected with OF and CF always cleared to zero
AND reg, data	[reg] ← [reg] ∧ data	Data can be 8- or 16-bit; reg cannot be segment register; this instruction should not be used to AND AL or AX with 8- or 16-bit immediate data; all flags are affected with OF and CF cleared to zero
AND A, data	[A] ← [A] ∧ data	Data can be 8- bit or 16-bit; 'A' must be AL or AX; all flags are affected with OF and CF cleared to zero
CALL PROC (NEAR)	Call a subroutine in the same segment with signed 16-bit displacement (to CALL a subroutine in ±32K)	NEAR in the statement BEGIN PROC NEAR indicates that the subroutine 'BEGIN' is in the same segment and BEGIN is 16-bit signed; CALL BEGIN instruction decrements SP by 2 and then pushes IP onto the stack and then adds the signed 16-bit value of BEGIN to IP and CS is unchanged; thus, a subroutine is called in the same segment (intrasegment direct)

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
CALL Reg 16	CALL a subroutine in the same segment addressed by the contents of a 16-bit general register	The 8086 decrements SP by 2 and then pushes IP onto the stack, then specified 16-bit register contents (such as BX, SI, and DI) provide the new value for IP; CS is unchanged (intrasegment indirect)
CALL Mem 16	CALL a subroutine addressed by the content of a memory location pointed to by 8086 16-bit register such as BX, SI, and DI	The 8086 decrements SP by 2 and pushes IP onto the stack; the 8086 then loads the contents of a memory location addressed by the content of a 16-bit register such as BX, SI, and DI into IP; [CS] is unchanged (intrasegment indirect)
CALL PROC (FAR)	CALL a subroutine in another segment	FAR in the statement BEGIN PROC FAR indicates that the subroutine 'BEGIN' is in another segment and the value of BEGIN is 32 bit wide The 8086 decrements SP by 2 and pushes CS onto the stack and moves the low 16-bit value of the specified 32-bit number such as 'BEGIN' in CALL BEGIN into CS; SP is again decremented by 2; IP is pushed onto the stack; IP is then loaded with high 16-bit value of BEGIN; thus, this instruction CALLS a subroutine in another code segment (intersegment direct)
CALLDWORD PTR [reg 16]	CALL a subroutine in another segment	This instruction decrements SP by 2, and pushes CS onto the stack; CS is then loaded with the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3] in DS; the SP is again decremented by 2; IP is pushed onto the stack; IP is then loaded with the contents of memory locations addressed by [reg 16] and [reg 16 + 1] in DS; typical 8086 registers used for reg 16 are BX, SI, and DI (intersegment indirect)
CBW	Convert a byte to a word	Extend the sign bit (bit 7) of AL register into AH
CLC	CF ← 0	Clear carry to zero
CLD	DF ← 0	Clear direction flag to zero
CLI	IF ← 0	Clear interrupt enable flag to zero to disable maskable interrupts
CMC	CF ← CF'	One's complement carry
CMP mem/reg 1, mem/reg 2	[mem/reg 1] – [mem/reg 2]; flags are affected	Mem/reg can be 8- or 16-bit; no memory-to-memory comparison allowed; result of subtraction is not provided; all flags are affected

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
CMP mem/reg, data	[mem/reg] – data, flags are affected	Subtracts 8- or 16-bit data from [mem or reg] and affects flags; no result is provided
CMP A, data	[A] – data, flags are affected	Subtract 8- or 16-bit data from AL or AX, respectively, and affects flags; no result is provided
CMPS BYTE or CMPSB	FOR BYTE [[SI]] – [[DI]], flags are affected [SI] ← [SI] ± 1 [DI] ← [DI] ± 1	8- or 16-bit data addressed by [DI] in ES is subtracted from 8- or 16-bit data addressed by SI in DS and flags are affected without providing any result; if DF = 0, then SI and DI are incremented by one for byte and two for word; if DF = 1, then SI and DI are decremented by one for byte and two for word; the segment register ES in destination cannot be overridden
CMPS WORD or CPSW	FOR WORD [[SI]] – [[DI]], flags are affected [SI] ← [SI] + 2 [DI] ← [DI] + 2	
CWD	Convert a word to 32 bits	Extend the sign bit of AX (bit 15) into DX
DAA	Decimal adjust [AL] after addition	This instruction uses implied addressing mode; this instruction converts [AL] into BCD; DAA should be used after BCD addition
DAS	Decimal adjust [AL] after subtraction	This instruction uses implied addressing mode; converts [AL] into BCD; DAS should be used after BCD subtraction
DEC reg 16	[reg 16] ← [reg 16] – 1	This is a one-byte instruction; used to decrement a 16-bit register except segment register; does not affect the carry flag
DEC mem/reg 8	[mem] ← [mem] – 1 or [reg 8] ← [reg 8] – 1	Used to decrement a byte or a word in memory or an 8-bit register content; segment register cannot be decremented by this instruction; does not affect carry flag
DIV mem/reg	16/8 bit divide: [AX] / [mem 8/reg 8] [AH] ← Remainder [AL] ← Quotient 32/16 bit divide: [DX] [AX] / [mem 16/reg 16] [DX] ← Remainder [AX] ← Quotient	Mem/reg is 8-bit for 16-bit by 8-bit divide and 16-bit for 32-bit by 16-bit divide; this is an unsigned division; no flags are affected; division by zero automatically generates an internal interrupt
ESC external OP code, source	ESCAPE to external processes	This instruction is used to pass instructions to a coprocessor such as the 8087 floating point coprocessor which simultaneously monitors the system bus with the 8086; the coprocessor OP codes are 6-bit wide; the

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
ESC external OP code, source (continued)		coprocessor treats normal 8086 instructions as NOP's; the 8086 fetches all instructions from memory; when the 8086 encounters an ESC instruction, it usually treats it as NOP; the coprocessor decodes this instruction and carries out the operation using the 6-bit OP code independent of the 8086; for ESC OP code, memory, the 8086 accesses data in memory for the co-processor; for ESC data, register, the coprocessor operates on 8086 registers; the 8086 treats this as an NOP
HLT	HALT	Halt
IDIV mem/reg	Same as DIV mem/reg	Same as DIV mem/reg
IMUL mem/reg	For 8 × 8 [AX] ← [AL] * [mem 8/reg 8] FOR 16 × 16 [DX] [AX] ← [AX]* [mem 16/reg 16]	Mem/reg can be 8- or 16-bit; only CF and OF are affected; signed multiplication
IN AL, DX	[AL] ← [PORT DX]	Input AL with the 8-bit content of a port addressed by DX; this is a one-byte instruction
IN AX, DX	[AX] ← [PORT DX]	Input AX with the 16-bit content of a port addressed by DX and DX + 1; this is a one-byte instruction
IN AL, PORT	[AL] ← [PORT]	Input AL with the 8-bit content of a port addressed by the second byte of the instruction
IN AX, PORT	[AX] ← [PORT]	Input AX with the 16-bit content of a port addressed by the 8-bit address in the second byte of the instruction
INC reg 16	[reg 16] ← [reg 16] + 1	This is a one-byte instruction; used to increment a 16-bit register except the segment register; does not affect the carry flag
INC mem/reg 8	[mem] ← [mem] + 1 or [reg 8] ← [reg 8] + 1	This is a two-byte instruction; can be used to increment a byte or word in memory or an 8-bit register content; segment registers cannot be incremented by this instruction; does not affect the carry flag
INT n (n can be zero thru 255)	[SP] ← [SP] – 2 [[SP]] ← Flags IF ← 0 TF← 0	Software interrupts can be used as supervisor calls; that is, request for service from an operating system; a different interrupt type can be used for each type of service that

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
INT n (continued)	[SP] ← [SP] – 2 [[SP]] ← [CS] [CS] ← 4n + 2 [SP] ← [SP] –2 [[SP]] ← [IP] [IP] ← 4n	the operating system could supply for an application or program; software interrupt instructions can also be used for checking interrupt service routines written for hardware-initiated interrupts
INTO	Interrupt on Overflow	Generates an internal interrupt if OF = 1; executes INT 4; can be used after an arithmetic operation to activate a service routine if OF = 1; when INTO is executed and if OF = 1, operations similar to INT n take place
IRET	Interrupt Return	POPS IP, CS and Flags from stack; IRET is used as return instruction at the end of a service routine for both hardware and software interrupts
JA/JNBE disp 8	Jump if above/jump if not below original	Jump if above/jump if not below nor equal with 8-bit signed displacement; that is, the displacement can be from -128_{10} to $+127_{10}$, zero being positive; JA and JNBE are the mnemonic which represent the same instruction; Jump if both CF and ZF are zero; used for unsigned comparison
JAE/JNB/JNC disp 8	Jump if above or equal/jump if not below/jump if no carry	Same as JA/JNBE except that the 8086 Jumps if CF = 0; used for unsigned comparison
JB/JC/JNAE disp 8	Jump if below/jump if carry/ jump if not above or equal	Same as JA/JNBE except that the jump is taken CF = 1, used for unsigned comparison
JBE/JNA disp 8	Jump if below or equal/jump if not above	Same as JA/JNBE except that the jump is taken if CF = 1 or ZF = 0; used for unsigned comparison
JCXZ disp 8	Jump if CX = 0	Jump if CX = 0; this instruction is useful at the beginning of a loop to bypass the loop if CX = 0
JE/JZ disp 8	Jump if equal/jump if zero	Same as JA/JNBE except that the jump is taken if ZF = 1; used for both signed and unsigned comparison
JG/JNLE disp 8	Jump if greater/jump if not less or equal	Same as JA/JNBE except that the jump is taken if ((SF + OF) or ZF) = 0; used for signed comparison
JGE/JNL disp 8	Jump if greater or equal/ jump if not less	Same as JA/JNBE except that the jump is taken if (SF + OF) = 0; used for signed comparison

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
JL/JNGE disp 8	Jump if less/Jump if not greater nor equal	Same as JA/JNBE except that the jump is taken if (SF + OF) = 1; used for signed comparison
JLE/JNG disp 8	Jump if less or equal/ jump if not greater	Same as JA/JNBE except that the jump is taken if ((SF + OF) or ZF) = 1; used for signed comparison
JMP Label	Unconditional Jump with a signed 8-bit (SHORT) or signed 16-bit (NEAR) displacement in the same segment	The label START can be signed 8-bit (called SHORT jump) or signed 16-bit (called NEAR jump) displacement; the assembler usually determines the displacement value; if the assembler finds the displacement value to be signed 8-bit (–128 to +127, 0 being positive), then the assembler uses two bytes for the instruction: one byte for the OP code followed by a byte for the displacement; the assembler sign extends the 8-bit displacement and then adds it to IP; [CS] is unchanged; on the other hand, if the assembler finds the displacement to be signed 16-bit (±32 K), then the assembler uses three bytes for the instruction: one byte for the OP code followed by 2 bytes for the displacement; the assembler adds the signed 16-bit displacement to IP; [CS] is unchanged; therefore, this JMP provides a jump in the same segment (intrasegment direct jump)
JMP Reg 16	[IP] ← [reg 16] [CS] is unchanged	Jump to an address specified by the contents of a 16-bit register such as BX, SI, and DI in the same code segment; in the example JMP BX, [BX] is loaded into IP and [CS] is unchanged (intrasegment memory indirect jump)
JMP mem 16	[IP] ← [mem] [CS] is unchanged	Jump to an address specified by the contents of a 16-bit memory location addressed by 16-bit register such as BX, SI, and DI; in the example, JMP [BX] copies the content of a memory location addressed by BX in DS into IP; CS is unchanged (intrasegment memory indirect jump)
JMP Label (FAR)	Unconditionally jump to another segment	This is a 5-byte instruction: the first byte is the OP code followed by four bytes of 32-bit immediate data; bytes 2 and 3 are loaded into IP; bytes 4 and 5 are loaded into CS to JUMP unconditionally to another segment (intersegment direct)

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
JMP DWORD PTR [reg 16]	Unconditionally jump to another segment	This instruction loads the contents of memory locations addressed by [reg 16] and [reg 16 + 1] in DS into IP; it then loads the contents of memory locations addressed by [reg 16 + 2] and [reg 16 + 3] in DS into CS; typical 8086 registers used for reg 16 are BX, SI, and DI (intersegment indirect)
JNE/JNZ disp 8	Jump if not equal/jump if not zero	Same as JA/JNBE except that the jump is taken if ZF = 0; used for both signed and unsigned comparison
JNO disp 8	Jump if not overflow	Same as JA/JNBE except that the jump is taken if OF = 0
JNP/JPO disp 8	Jump if no parity/jump if parity odd	Same as JA/JNBE except that the jump is taken if PF = 0
JNS disp 8	Jump if not sign	Same as JA/JNBE except that the jump is taken if SF = 0
JO disp 8	Jump if overflow	Same as JA/JNBE except that the jump is taken if OF = 1
JP/JPE disp 8	Jump if parity/jump if parity even	Same as JA/JNBE except that the jump is taken if PF = 1
JS disp 8	Jump if sign	Same as JA/JNBE except that the jump is taken if SF = 1
LAHF	[AH] ← Flag low-byte	This instruction has implied addressing mode; it loads AH with the low byte of the flag register; no flags are affected
LDS reg, mem	[reg] ← [mem] [DS] ← [mem + 2]	Load a 16-bit register (AX, BX, CX, DX, SP, BP, SI, DI) with the content of specified memory and load DS with the content of the location that follows; no flags are affected; DS is used as the segment register for mem
LEA reg, mem	[reg] ← [offset portion of address]	LEA (load effective address) loads the value of the source operand rather than its content to register (such as SI, DI, BX) which are allowed to contain offset for accessing memory; no flags are affected
LES reg, mem	[reg] ← [mem] [ES] ← [mem + 2]	DS is used as the segment register for mem; in the example LES DX, [BX], DX is loaded with 16-bit value from a memory location addressed by 20-bit physical address computed from DS and BX; the 16-bit content of the next memory is loaded into ES; no flags are affected
LOCK	LOCK bus during next instruction	Lock is a one-byte prefix that causes the 8086 (configured in maximum mode) to assert its bus LOCK signal while following

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
LOCK (continued)		instruction is executed; this signal is used in multiprocessing; the LOCK pin of the 8086 can be used to LOCK other processors off the system bus during execution of an instruction; in this way, the 8086 can be assured of uninterrupted access to common system resources such as shared RAM
LODS BYTE or LODSB	FOR BYTE [AL] ← [[SI]] [SI] ← [SI] + 1	Load 8-bit data into AL or 16-bit data into AX from a memory location addressed by SI in segment DS; if DF = 0, then SI is incremented by 1 for byte or incremented by 2 for word after the load; if DF = 1, then SI is decremented by 1 for byte or decremented by 2 for word; LODS affects no flags
LODS WORD or LODSW	FOR WORD [AX] ← [[SI]] [SI] ← [SI] + 2	
LOOP disp 8	Loop if CX not equal to zero	Decrement CX by one, without affecting flags and loop with signed 8-bit displacement (from −128 to +127, zero being positive) if CX is not equal to zero
LOOPE/LOOPZ disp 8	Loop while equal/loop while zero	Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX is equal to zero, and if ZF = 1 which results from execution of the previous instructions
LOOPNE/LOOPNZ disp 8	Loop while not equal/loop while not zero	Decrement CX by one without affecting flags and loop with signed 8-bit displacement if CX is not equal to zero and ZF = 0 which results from execution of previous instruction
MOV mem/reg 2, mem/reg 1	[mem/reg 2] [mem/reg 1]	mem uses DS as the segment register; no memory-to-memory operation allowed; that is, MOV mem, mem is not permitted; segment register cannot be specified as reg or reg; no flags are affected; not usually used to load or store 'A' from or to memory
MOV mem, data	[mem] ← data	mem uses DS as the segment register; 8- or 16-bit data specifies whether memory location is 8- or 16-bit; no flags are affected
MOV reg, data	[reg] ← data	Segment register cannot be specified as reg; data can be 8- or 16-bit; no flags are affected
MOV A, mem	[A] ← [mem]	Takes fewer bytes than reg, mem; mem uses DS as segment register; 'A' can be AL or AX; no flags are affected

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
MOV mem, A	[mem] ← [A]	mem uses DS as segment register; 'A' can be AL or AX; no flags are affected; needs fewer bytes than MOV mem, reg
MOV segreg, mem/reg	[segreg] ← [mem/reg]	mem uses DS as segment register; used for initializing CS, DS, ES, and SS; no flags are affected
MOV mem/reg, segreg	[mem/reg] ← [segreg]	mem uses DS as segment register; no flags are affected
MOVS BYTE or MOVSB	FOR BYTE [[DI]] ← [[SI]] [SI] ← [SI] + 1	Move 8-bit or 16-bit data from the memory location addressed by SI in segment DS location addressed by DI in ES; segment DS can be overridden by a prefix but destination segment must be ES and cannot be overridden; if DF = 0, then SI is incremented by one for byte or incremented by two for word; if DF = 1, then SI is decremented by one for byte or by two for word
MOVS WORD or MOVSW	FOR WORD [[DI]] ← [[SI]] [SI] ← [SI] + 2	
MUL mem/reg	FOR 8 × 8 [AX] ← [AL]* [mem/reg] FOR 16 × 16 [DX] [AX] ← [AX]* [mem/reg]	mem/reg can be 8- or 16-bit; only CF and OF are affected; unsigned multiplication
NEG mem/reg	[mem/reg] ← [mem/reg]' + 1	mem/reg can be 8- or 16-bit; performs two's complement subtraction of the specified operand from zero, that is, two's complement of a number is formed; all flags are affected except CF = 0 if [mem/reg] is zero; otherwise CF = 1
NOP	NO Operation	8086 does nothing
NOT reg	[reg] ← [reg]'	mem and reg can be 8- or 16-bit; segment registers are not allowed; no flags are affected; ones complement reg
NOT mem	[mem] ← [mem]'	mem uses DS as the segment register; no flags are affected; ones complement mem
OR Mem/reg 1, Mem/reg 2	[mem/reg 1] ← [mem/reg 1] ∨ [mem/reg 2]	No memory-to-memory operation is allowed; [mem] or [reg 1] or [reg 2] can be 8- or 16-bit; all flags are affected with OF and CF cleared to zero; no segment registers are allowed; mem uses DS as segment register
OR mem, data	[mem] ←[mem] ∨ data	mem and data can be 8- or 16-bit; mem uses DS as segment register; all flags are affected with CF and OF cleared to zero

TABLE 3.1 (continued)

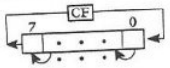
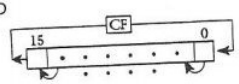
Instructions	Interpretation	Comments
OR reg, data	[reg] ← [reg] ∨ data	reg and data can be 8- or 16-bit; no segment registers are allowed; all flags are affected with CF and OF cleared to zero; should not be used to OR AL or AX with immediate data
OR A, data	[A] ← [A] ∨ data	Data can be 8- or 16-bit; A must be AL or AX; all flags are affected with OF and CF cleared to zero
OUT DX, AL	[PORT] ← [AL] DX	Output the 8-bit contents of AL into an I/O Port addressed by the 16-bit content of DX; this is a one-byte instruction
OUT DX, AX	[PORT] ← [AX] DX	Output the 16-bit contents of AX into an I/O Port addressed by the 16-bit content of DX; this is a one-byte instruction
OUT PORT, AL	[PORT] ← [AL]	Output the 8-bit contents of AL into the Port specified in the second byte of the instruction
OUT PORT, AX	[PORT] ← [AX]	Output the 16-bit contents of AX into the Port specified in the second byte of the instruction
POP mem	[mem] ← [[SP]] [SP] ← [SP] + 2	mem uses DS as the segment register; no flags are affected
POP reg	[reg] ← [[SP]] [SP] ← [SP] + 2	Cannot be used to POP segment registers or flag register
POP segreg	[segreg] ← [[SP]] [SP] ← [SP] + 2	POP CS is illegal
POPF	[Flags] ← [[SP]] [SP] ← [SP] + 2	This instruction pops the top two stack bytes in the 16-bit flag register
PUSH mem	[SP] ← [SP] − 2 [[SP]] ← [mem]	mem uses DS as segment register; no flags are affected; pushes 16-bit memory contents
PUSH reg	[SP] ← [SP] − 2 [[SP]] ← [reg]	reg must be a 16-bit register; cannot be used to PUSH segment register or Flag register
PUSH segreg	[SP] ← [SP] − 2 [[SP]] ← [segreg]	PUSH CS is illegal
PUSHF	[SP] ← [SP] − 2 [[SP]] ← [Flags]	This instruction pushes the 16-bit Flag register onto the stack
RCL mem/reg, 1	ROTATE through carry left once byte or word in mem/reg	FOR BYTE  FOR WORD 

TABLE 3.1 (continued)

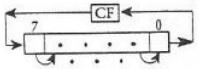
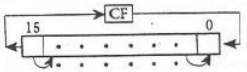
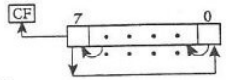
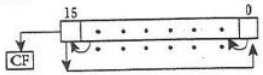
Instructions	Interpretation	Comments
RCL mem/reg, CL	ROTATE through carry left byte or word in mem/reg by [CL]	Operation same as RCL mem/reg, 1 except the number of rotates is specified in CL for rotates up to 255; zero or negative rotates are illegal
RCR mem/reg, 1	ROTATE through carry right once byte or word in mem/reg	FOR BYTE  FOR WORD 
RCR mem/reg, CL	ROTATE through carry right byte or word in mem/reg by [CL]	Operation same as RCR mem/reg, 1 except the number of rotates is specified in CL for rotates up to 255; zero or negative rotates are illegal
RET	.POPS IP for intrasegment CALLS .POPS IP and CS for intersegment CALLS	The assembler generates an intrasegment return if the programmer has defined the subroutine as NEAR; for intrasegment return, the following operations take place: [IP] ← [[SP]], [SP] ← [SP] + 2; on the other hand, the assembler generates an intersegment return if the subroutine has been defined as FAR; in this case, the following operations take place: [IP] ← [[SP]], [SP] ← [SP] + 2, [CS] ← [[SP]], [SP] ← [SP] + 2; an optional 16-bit displacement 'START' can be specified with the intersegment return such as RET START; in this case, the 16-bit displacement is added to the SP value; this feature may be used to discard parameter pushed onto the stack before the execution of the CALL instruction
ROL mem/reg, 1	ROTATE left once byte or word in mem/reg	FOR BYTE  FOR WORD 

TABLE 3.1 (continued)

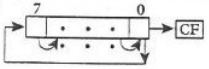
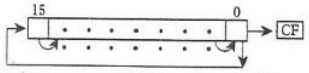
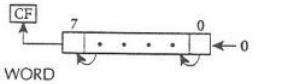
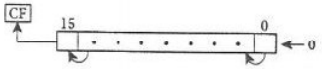
Instructions	Interpretation	Comments
ROL mem/reg, CL	ROTATE left byte or word by the content of CL	[CL] contains rotate count up to 255; zero and negative shifts are illegal; CL is used to rotate count when the rotate is greater than once; mem uses DS as the segment register
ROR mem/reg, 1	ROTATE right once byte or word in mem/reg	FOR BYTE  FOR WORD 
ROR mem/reg, CL	ROTATE right byte or word in mem/reg by [CL]	Operation same as ROR mem/reg, 1; [CL] specifies the number of rotates for up to 255; zero and negative rotates are illegal; mem uses DS as the segment register
SAHF	[Flags, low-byte] ← [AH]	This instruction has the implied addressing mode; the content of the AH register is stored into the low-byte of the flag register; no flags are affected
SAL mem/reg, 1	Shift arithmetic left once byte or word in mem or reg	FOR BYTE  FOR WORD 
SAL mem/reg, CL	Shift arithmetic left byte or word by shift count on CL	Mem uses DS as the segment register; reg cannot be segment registers; OF and CF are affected; if sign bit is changed during or after shifting, the OF is set to one Operation same as SAL mem/reg, 1; CL contains shift count for up to 255; zero and negative shifts are illegal; [CL] is used as shift count when shift is greater than one; OF and SF are affected; if sign bit of [mem] is changed during or after shifting, the OF is set to one; mem uses DS as segment register

TABLE 3.1 (continued)

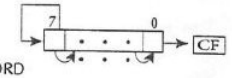
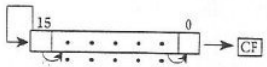
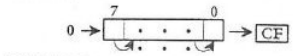
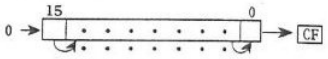
Instructions	Interpretation	Comments
SAR mem/reg,.1	SHIFT arithmetic right once byte or word in mem/reg	FOR BYTE  FOR WORD 
SAR mem/reg, CL	SHIFT arithmetic right byte or word in mem/reg by [CL]	Operation same as SAR mem/reg, 1; however, shift count is specified in CL for shifts up to 255; zero and negative shifts are illegal
SBB mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] – [mem/reg 2] – CY	Same as SUB mem/reg 1, mem/reg 2 except this is a subtraction with borrow
SBB mem, data	[mem] ← [mem] – data – CY	Same as SUB mem, data except this is a subtraction with borrow
SBB reg, data	[reg] ← [reg] – data – CY	Same as SUB reg, data except this is a subtraction with borrow
SBB A, data	[A] ← [A] – data – CY	Same as SUB A, data except this is a subtraction with borrow
SCAS BYTE or SCASB	FOR BYTE [AL] – [[DI]], flags are affected, [DI] [DI] ± 1	8- or 16-bit data addressed by [DI] in ES is subtracted from 8- or 16-bit data in AL or AX and flags are affected without affecting [AL] or [AX] or string data; ES cannot be overridden; if DF = 0, then DI is incremented by one for byte and two for word; if DF = 1, then DI is decremented by one for byte or decremented by two for word
SCAS WORD or SCASW	FOR WORD [AX] – [[DI]], flags are affected, [DI] ← [DI] ± 2	Same as SAL mem/reg, 1
SHL mem/reg, 1	SHIFT logical left once byte or word in mem/reg	
SHL mem/reg, CL	SHIFT logical left byte or word in mem/reg by the shift count in CL	Same as SAL mem/reg, CL except overflow is cleared to zero
SHR mem/reg, 1	SHIFT right logical once byte or word in mem/reg	FOR BYTE  FOR WORD 

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
SHR mem/reg, CL	SHIFT right logical byte or word in mem/reg by [CL]	Operation same as SHR mem/reg, 1; however, shift count is specified in CL for shifts up to 255; zero and negative shifts are illegal
STC	CF ← 1	Set carry to one
STD	DF ← 1	Set direction flag to one
STI	IF ← 1	Set interrupt enable flag to one to enable maskable interrupts
STOS BYTE or STOSB	FOR BYTE [[DI]] ← [AL] [DI] ← [DI] ± 1	Store 8-bit data from AL or 16-bit data from AX into a memory location addressed by DI in segment ES; segment register ES cannot be overridden; if DF = 0, then DI is incremented by one for byte or incremented by two for word after the store
STOS WORD or STOSW	FOR WORD [[DI]] ← [AX] [DI] ← [DI] ± 2	
SUB mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] − [mem/reg 2]	No memory-to-memory SUB permitted; all flags are affected; mem uses DS as the segment register
SUB mem, data	[mem] ← [mem] − data	Data can be 8- or 16-bit; mem uses DS as the segment register; all flags are affected
SUB reg, data	[reg] ← [reg] − data	Data can be 8- or 16-bit; this instruction is not usually used for subtracting data from AX or AL; SUB A, data is used for this; all flags are affected
SUB A, data	[A] ← [A] − data	'A' can be AL or AX; data can be 8- or 16-bit; all flags are affected
TEST mem/reg 1, mem/reg 2	[mem/reg 1] − [mem/reg 2], no result; flags are affected	No memory-to-memory TEST is allowed; no result is provided; all flags are affected with CF and OF cleared to zero; [mem], [reg 1] or [reg 2] can be 8- or 16-bit; no segment registers are allowed; mem uses DS as the segment register
TEST mem/data	[mem] − data, no result; flags are affected	Mem and data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OD cleared to zero; mem uses DS as the segment register
TEST reg, data	[reg] − data, no result; flags are affected	Reg and data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero; reg cannot be segment register; should not be used to test AL or AX with immediate data
TEST A, data	[A] − data, no results; flags are affected	'A' must be AX or AL; data can be 8- or 16-bit; no result is provided; all flags are affected with CF and OF cleared to zero

TABLE 3.1 (continued)

Instructions	Interpretation	Comments
WAIT	8086 enters wait state	Causes CPU to enter wait state if the 8086 TEST pin is high; while in wait state, the 8086 continues to check TEST pin for low; if TEST pin goes back to zero, the 8086 executes the next instruction; this feature can be used to synchronize the operation of 8086 to an event in external hardware
XCHG AX, reg	[AX] ↔ [reg]	Reg must be 16-bit; no flags are affected; reg cannot be segment register
XCHG mem, reg	[mem] ↔ [reg]	Reg and mem can be both 8- or 16-bit; mem uses DS as the sement register; reg cannot be segment register; no flags are affected
XCHG reg, reg	[reg] ↔ [reg]	Reg not used to exchange reg with AX; reg can be 8- or 16-bit; reg cannot be segment register; no flags are affected
XLAT	[AL] ← [AL] + [BX]	This instruction is useful for translating characters from one code such as ASCII to another such as EBCDIC; this is a no-operand instruction and is called an instruction with implied addressing mode; the instruction loads AL with the contents of a 20-bit physical address computed from DS, BX, and AL; this instruction can be used to read the elements in a table where BX can be loaded with a 16-bit value to point to the starting address (offset from DS) and AL can be loaded with the element number (0 being the first element number); no flags are affected; the XLAT instruction is equivalent to MOV AL, [AL] [BX]
XOR mem/reg 1, mem/reg 2	[mem/reg 1] ← [mem/reg 1] ⊕ [mem/reg 2]	No memory-to-memory operation is allowed; [mem] or [reg 1] or [reg 2] can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero; mem uses DS as the segment register
XOR mem, data	[reg] ← [mem] ⊕ data	Data and mem can be 8- or 16-bit; mem uses DS as the segment register; mem cannot be segment register; all flags are affected with CF and OF cleared to zero
XOR Reg, data	[reg] ← [reg] ⊕ data	Same as XOR mem, data; should not be used for XORing AL or AX with immediate data
XOR A, data	[A] ← [A] ⊕ data	'A' must be AL or AX; data can be 8- or 16-bit; all flags are affected with CF and OF cleared to zero

Explanation of Table 3.A-1 Instructions

- MOV CX, DX copies the 16-bit content of DX into CX. MOV AX, 0205H moves immediate data 0205H into 16-bit register AX. MOV CH, [BX] moves the 8-bit content of memory location addressed by BX in segment register DS into CH. If [BX] = 0050H, [DS] = 2000H, [20050H] = 08H, then after MOV CH, [BX], the content of CH will be 08H.
  MOV START [BP], CX moves the 16-bit (CL to first location and then CH) content of CX into two memory locations addressed by the sum of the displacement START and BP in segment register SS. For example, if [CX] = 5009H, [BP] = 0030H, [SS] = 3000H, START = 06H, then after MOV START [BP], CX physical memory location [30036H] = 09H and [30037H] = 50H. Note that the segment register SS can be overridden by CS using MOV CS: START [BP], CX.
- PUSH START [BX] pushes the 16-bit contents of two memory locations starting at the 20-bit physical address computed from START, BX, and DS after decrementing SP by 2.
- POP ES pops the top stack word into ES and then increments SP by 2.
- XCHG START [BX], AX exchanges the 16-bit word in AX with the contents of two consecutive memory locations starting at 20-bit physical address computed from START, BX, and DS. [AL] is exchanged with the content of the first location and [AH] is exchanged with the content of the next location.
- XLAT can be used to convert a code such as ASCII into another code such as EBCDIC.

Suppose that an 8086-based microcomputer is interfaced to an ASCII keyboard and an IBM printer (EBCDIC code). In such a system, any number entered into the microcomputer will be in ASCII code which must be converted to EBCDIC code before outputting to the printer. If the number entered is 4, then the ASCII code for 4 (34H) must be translated to the EBCDIC code for 4 (F4H). A look-up table containing EBCDIC code for all the decimal numbers (0 to 9) can be stored in memory at the 16-bit starting address (offset) in DS, 3030H with the code for 0 stored at 3030H, code for 1 at 3031H, and so on. Now, if AL is loaded with the ASCII code 34H and BX is loaded with 3000H, then by using the XLAT instruction, content of memory location 3034H in DS containing the EBCDIC code for 4 (F4H) from the look-up table is read into AL, thus replacing the ASCII code for 4 with the EBCDIC code for 4.

- Note that MOV A, mem can also be obtained from MOV mem/reg 2, mem/reg 1. This means that MOV A, mem has two different OP codes; one obtained from MOV mem/reg 2, mem/reg 1 and the other from the dedicated MOV A, mem instruction. The assembler usually provides the OP code for the MOV A, M since it takes fewer bytes. The same concept applies to other instructions involving "A" which have more than one OP code.
- Symbolic names such as START can be defined as an address containing a byte or word by using the DB or DW assembler directive. To define the symbolic name as a constant for immediate data, one can use the EQU directive.

Explanation of Table 3.A-2 Instructions

- Consider fixed port addressing in which the 8-bit port address is directly specified as part of the instruction. IN AL, 38H inputs 8-bit data from port 38H into AL. IN AX, 38H inputs 16-bit data from ports 38H and 39H into AX. OUT 38H, AL outputs the contents of AL to port 38H. OUT 38H, AX, on the other hand, outputs the 16-bit contents of AX to ports 38H and 39H.
- For the variable port addressing, the port address is 16-bit and is specified in the DX register. Consider ports addressed by 16-bit address contained in DX. Assume [DX] = 3124_{16} in all the following examples:

  IN AL, DX inputs 8-bit data from 8-bit port 3124_{16} into AL.
  IN AX, DX inputs 16-bit data from ports 3124_{16} and 3125_{16} into AX.
  OUT DX, AL outputs 8-bit data from AL into port 3124_{16} data.
  OUT DX, AX outputs 16-bit data from AX into ports 3124_{16} and 3125_{16}.

Variable port addressing allows up to 65,536 ports with addresses from 0000H to FFFFH. The port addresses in the variable port addressing can be calculated dynamically in a program. For example, assume that an 8086-based microcomputer is connected to three printers via three separate ports. Now, in order to output to each one of the printers, separate programs are required if fixed port addressing is used. However, with variable port addressing one can write a general subroutine to output to the printers and then supply the address of the port for a particular printer in which data output is desired to register DX in the subroutine.

Explanation of Table 3.A-3 Instructions

- LEA reg, mem loads an offset (mem) in DS directly into the specified register. The XLAT and string instructions assume that certain registers point to operands. LEA can be used to load these addresses. For example, LEA can be used to load the address of the table used by the XLAT instructions. Consider converting an ASCII code into EBCDIC code. Suppose that EBCDIC codes for the first seven symbols NUL (NULL), SOH (Start of Heading), STX (Start Text), and ETX (End Text); EOT (End of Transmission), ENQ (Enquiry), and ACK (Acknowledge) are stored in memory.
  Starting at location TABLE 1 as follows:

```
TABLE 1:   00H; EBCDIC Code for NUL
           01H; Code for SOH
           02H; Code for STX
           03H; Code for ETX
           37H; Code for EOT
           2DH; Code for ENQ
           2EH; Code for ACK
```

  Suppose that the ASCII codes for these characters are stored in memory starting at address TABLE 2 as follows:

```
TABLE 2:   00H; ASCII Code for NUL
           01H; Code for SOH
           02H; Code for STX
           03H; Code for ETX
           04H; Code for EOT
           05H; Code for ENQ
           06H; Code for ACK
```

  Now, in order to translate an ASCII Code for a symbol such as EOT into its EBCDIC Code, ASCII Code for EOT (04H) can be loaded into AL, BX can be initialized with the value of TABLE 1 using LEA, and XLAT can then be used to load the ABCDIC Code for EOT (37H) into AL as follows:

```
LEA BX, TABLE 1 ; Load the value TABLE 1 into BX
MOV AL, 04H     ; Load ASCII Code for EOT into AL
XLAT            ; Load the content of memory loca-
                  tion addressed by [TABLE 1 + 04H]
                  into AL.
```


- LDS reg, mem can be used to initialize SI and DS to point to the start of the source string before using one of the string instructions. For example, LDS SI, [BX] loads the 16-bit contents of memory offset by [BX] in DS into SI and the 16-bit contents of memory offset by [BX + 2] in DS into DS.
- LES reg, mem can be used to point to the start of the destination string before using one of the string instructions. For example, LES DI, [BX] loads the 16-bit contents of memory offset by [BX] in DS to DI and then initializes ES with the 16-bit contents of memory offset by [BX + 2] in DS.

  Table 3.A-4 is self-explanatory.

Explanation of Table 3.A-5 Instructions

- Numerical data received by an 8086-based microcomputer from a terminal is usually in ASCII code. The ASCII codes for numbers 0 to 9 are 30H through 39H. Two 8-bit data can be entered into an 8086-based microcomputer via a terminal. The ASCII codes for these data (with 3 as the upper middle for each type) can be added. AAA instruction can then be used to provide the correct unpacked BCD. Suppose that ASCII codes for 2 (32_{16}) and 5 (35_{16}) are entered into an 8086-based microcomputer via a terminal. These ASCII codes can be added and then the result can be adjusted to provide the correct unpacked BCD using AAA instructions as follows:

```
ADD CL, DL    ; [CL] = 32(16) = ACSII for 2
              ; [DL] = 35(16) = ASCII for 5
              ; Result [CL] = 67(16)
MOV AL, CL    ; Move ASCII result
              ; into AL since AAA
              ; adjust only [AL]
AAA           ; [AL] = 07, unpacked
              ; BCD for 7
```

  Note that in order to send the unpacked BCD result 07_{16} back to the terminal, [AL] = 07 can be ORed with 30H to provide 37H, the ASCII code for 7.
- DAA is used to adjust the result of adding two packed BCD numbers in AL to provide a valid BCD number. If after the addition, the low 4-bit of the result in AL is greater than 9 (or if AF = 1), then the DAA adds 6 to the low 4 bits of AL. On the other hand, if the high 4 bits

of the result in AL is greater than 9 (or if CF = 1), then DAA adds 60H to AL. As an example, consider adding two packed BCD digits 55 with 18 as follows:

```
ADD AL, DL  ; [AL] = 55 BCD
            ; [DL] = 18 BCD
            ; Result = [AL] = 6DH

DAA         ; Since low nibble
            ; D = 1101₂ > 9, add i.e. 1101₂ + 0110₂ →
                                    10011₂
                                      ↑
                                    3BCD
                                   carry
```

- The ASCII codes for two-bit numbers in an 8086-based microcomputer can be subtracted. Assume

```
0011 0101          0011 0101
+1100 1001         - 0011 0111
1111 1110

            [AL] = 35H = ASCII for 5
            [DL] = 37H = ASCII for 7
```

The following instruction sequence provides the correct subtraction result:

```
SUB AL, DL  ; [AL] = 1111 1110₂ = subtraction
              result
            ; in 2's complement
            ; CF = 1
AAS         ; [AL] = BCD02
            ; CF = 1 means
            ; borrow to be
            ; used in multi BCD subtraction
```

AAS adjusts [AL] and leaves zeros in the upper nibble. To output to the terminal from the microcomputer, BCD data can be ORed with 30H to produce the correct ASCII code.

- DAS can be used to adjust the result of subtraction in AL of two packed BCD numbers to provide the correct packed BCD. If low 4-bit in AL is greater than 9 (or if AF = 1), then DAS subtracts 6 from the low 4-bit of AL. On the other hand, if the upper 4-bit of the result in AL is greater than 9 (or if CF = 1), DAS subtracts 60 from AL.

While performing these subtractions, any borrows from LOW and HIGH nibbles are ignored. For example, consider subtracting BCD 55 in DL from BCD 94 in AL.

```
SUB AL, DL  ; [AL] = 3FH → Low nibble = 1111
DAS         ; CF = 0                -6 = 1010
                                         1001
            ; [AL] = 39 BCD           1
                                     ignore
```

- IMUL mem/reg provides signed 8 × 8 or signed 16 × 16 multiplication. As an example, if [CL] = FDH = -3_{10}, [AL] = FEH = -2_{10}, then after IMUL CL, register AX contains 0006H.
- Consider 16 × 16 unsigned multiplication, MUL WORDPTR [BX]. If [BX] = 0050H, [DS] = 3000H, [30050H] = 0002H, [AX] = 0006H, then after MUL WORDPTR [BX], [DX] = 0000H, [AX] = 000CH.
- Consider DIV BL. If [AX] = 0009H, [BL] = 02H, then after DIV BL,

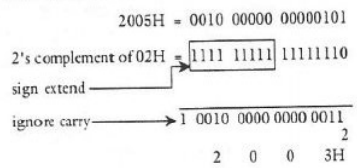
```
[AH] = Remainder = 01H
[AL] = Quotient = 04H
```

- Consider IDIV.WORDPTR [BX]. If [BX] = 0020H, [DS] = 2000H, [20020H] = 0004H, [DX] [AX] = 00000011H, then after IDIV. WORDPTR [BX],

```
[DX] = Remainder = 0001H
[AX] = Quotient = 0004H
```

- AAD converts two unpacked BCD digits in AH and AL to an equivalent binary number in AL. AAD must be used before dividing two unpacked BCD digits in AX by an unpacked BCD byte. For example, consider dividing [AX] = unpacked BCD 0508 (58 decimal) by [DH] = 07H. [AX] must first be converted to binary by using AAD. The register AX will then contain 003AH = 58 decimal. After DIV DH, [AL] = quotient = 08 unpacked BCD, [AH] = remainder = 02 unpacked BCD.
- Consider CBW. This instruction extends the sign from the AL register to AH register. For example, if [AL] = E2H, then after CBW, AH will contain FFH since the most significant bit of E2H is one. Note that sign extension is useful when one wants to perform an arithmetic

operation on two signed numbers of different sizes. For example, the 8-bit signed number 02H can be subtracted from 16-bit signed number 2005H as follows:



Another example of sign extension is that in order to multiply a signed 8-bit number by a signed 16-bit number, one must first sign-extend the signed 8-bit into a signed 16-bit number and then the instruction IMUL can be used for 16 × 16 signed multiplication.

- AAM adjusts the product of two unpacked BCD digits in AX. If [AL] = BCD3 = 00000011$_2$ and [CH] = BCD8 = 0000 1000$_2$, then after MUL CH, [AX] = 0000000000011000$_2$ = 0018H, and after using AAM, [AX] = 00000010 00000100$_2$ = unpacked 24. The following instruction sequence accomplishes this:

```
MUL CH
AAM
```

Note that the 8086 does not allow multiplication of two ASCII codes. Therefore, before multiplying two ASCII bytes received from a terminal, one must mask the upper 4-bits of each one of these bytes and then multiply them as two unpacked BCD digits and then use AAM for adjustment. In order to convert the unpacked BCD product back to ASCII, for sending back to the terminal, one must OR the product with 3030H.

Table 3.A-6 is self-explanatory.

Explanation of Table 3.A-7 Instructions

- LODS can be represented in four forms:

```
For Byte        LODS BYTE
                   or
                LODSB

For Word        LODS WORD
                   or
                LODSW
```


If [SI] = 0020H, [DS] = 3000H, [30020H] = 05H, DF = 0, then after LODS BYTE or LODSB, [AL] = 05H, [SI] = 0021H.

- [DS] = 2000H, [ES] = 3000H, [SI] = 0020H, [DI] = 0050H, DF = 0, [20020H] = 0205H, [30050H] 4071H, then after execution of MOVSW or MOVS WORD, memory location 30050H will contain 0205H. Since DF = 0, register SI will contain 0022H and register DI will contain 0052H.
- REP, a one-byte prefix, can be used with MOVS to cause the instruction MOVS to continue executing until CX = 0. Each time the instructions such as REP MOVSB or REP MOVSW are executed, CX is automatically decremented by 1, and if [CX] ≠ 0, MOVS is reexecuted and then CX is decremented by 1 until CX = 0; the next instruction is then executed. REP MOVSB or REP MOVSW can be used to move string bytes until the string length (loaded into CX before the instructions REP MOVSB or REP MOVSW) is decremented for zero.

REPE/REPZ or REPNE/REPNZ prefix can be used with CMPS or SCAS to cause one of these instructions to continue executing until ZF = 0 (for REPNE/REPNZ prefix) and CX = 0. Note that REPE and REPZ are two mnemonics for the same prefix byte. Similarly, REPNE and REPNZ also provide the same purpose. If CMPS is prefixed with REPE or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)". If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings not equal (ZF = 0)". Thus, repeated CMPS can be used to find matching or differing string elements. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)". This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)". This form may be used to locate a value in a string. Repeated string instructions are interruptible; the processor recognizes the interrupt before processing the next string element. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. When multiple prefixes (such as LOCK and segment override) are specified in addition to any of the repeat prefixes, program execution does not resume properly upon return from interrupt.

The processor remembers only one prefix in effect at the time of the interrupt, the prefix that immediately precedes the string instruc-

tions. Upon return from interrupt, program execution resumes at this point but any additional prefixes specified are not recognized. The multiple prefix must be used with a string instruction; maskable interrupts should be disabled for the duration of the repeated execution. However, this will not prevent a nonmaskable interrupt from being recognized.

- Note that the segment register for destination for all string instructions is always ES and cannot be overridden. However, DS can be overridden for the source using a prefix. For example, ES: MOVSB instruction uses the segment register as ES for both source and destination.

Table 3.A-8 is self-explanatory.

Explanation of Table 3.A-9 Instructions

All 8086 conditional branch instructions use 8-bit signed displacement. That is, the displacement covers a branch range of -128_{10} to $+127_{10}$ with 0 being positive. In order to branch out of this range, the 8086 unconditional jump instructions (having direct mode) must be used.

Conditional jumps are typically used with compare instructions to find the relationship (equal to, greater than, or less than) between two numbers. The use of conditional instructions depends whether the numbers to be compared are signed or unsigned. The 8-bit number $1111\ 1110_2$, when considered as signed, has a value of -2_{10}; the same number has a value of $+254_{10}$ when considered as unsigned. This number, when considered signed, will be smaller than zero, and when unsigned will be greater than zero. Some new terms are used to differentiate between signed and unsigned conditional transfers. For the unsigned numbers, the terms used are "below and above", while for signed numbers, the terms "less than and greater than" are used. In the above, the number $1111\ 1110_2$ when considered signed is greater than $0000\ 0000_2$, while the same number $1111\ 1110_2$ when considered unsigned is above $0000\ 0000_2$. The conditional transfer instructions for equality of two numbers are the same for both signed and unsigned numbers. This is because when two numbers are compared for equality irrespective of whether they are signed or unsigned, they will provide a zero result (ZF = 1) if equal or a nonzero result (ZF = 0) if not equal. Therefore, the same instructions apply for both signed and unsigned numbers for "equal to" or "not equal to" conditions, and the various signed and unsigned conditional branch instructions for determining the relationship between two numbers are as follows:

Signed		Unsigned	
Name	**Alternate name**	**Name**	**Alternate name**
JE disp8 (JUMP if equal)	JZ disp8 (JUMP if result zero)	JE disp8 (JUMP if equal)	JZ disp8 (JUMP if zero)
JNE disp8 (JUMP if not equal)	JNZ disp8 (JUMP if not zero)	JNE disp8 (JUMP if not equal)	JNZ disp8 (JUMP if not zero)
JG disp8 (JUMP if greater)	JNLE disp8 (JUMP if not less or equal)	JA disp8 (JUMP if above)	JNBE disp8 (JUMP if not below or equal)
JGE disp8 (JUMP if greater or equal)	JNL disp8 (JUMP if not less)	JAE disp8 (JUMP if above or equal)	JNB disp8 (JUMP if not below)
JL disp8 (JUMP if less than)	JNGE disp8 (JUMP if not greater or equal)	JB disp8 (JUMP if below)	JNAE disp8 (JUMP if not above or equal)
JLE disp8 (JUMP if less or equal)	JNG disp8 (JUMP if not greater)	JBE disp8 (JUMP if below or equal)	JNA disp8 (JUMP if not above)

3.4.1 SIGNED AND UNSIGNED CONDITIONAL BRANCH INSTRUCTIONS

There are also conditional transfer instructions that are concerned with the setting of status flags rather than relationship between two numbers. The table below lists these instructions:

JC disp8	JUMP if carry, i.e., CF = 1
JNC disp8	JUMP if no carry, i.e., CF = 0
JP disp8	JUMP if parity, i.e., PF = 1
JNP disp8	JUMP if no parity, i.e., PF = 0
JO disp8	JUMP if overflow, i.e., OF = 1
JNO disp8	JUMP if no overflow, i.e., OF = 0
JS disp8	JUMP if sign, i.e., SF = 1
JNS disp8	JUMP if no sign, i.e., SF = 0
JZ disp8	JUMP if result zero, i.e., Z = 1
JNZ disp8	JUMP if result not zero, i.e., Z = 0

3.4.2 CONDITIONAL JUMPS AFFECTING INDIVIDUAL FLAGS

The JP and JNP instructions use alternate names. They are JPE (Jump if Parity Even) and JPO (Jump if Parity Odd), respectively.

Now, let us look at the flag settings for the conditional transfer instructions concerned with the relationship between two numbers. These are listed in the table below:

Flag Settings for Instructions Concerned with Relationships between Two Numbers

Instruction	Flag setting
JE/JZ	ZF = 1
JNE/JNZ	ZF = 0
JL/JNGE	SF + OF = 1
JNL/JGE	SF + OF = 0
JG/JNLE	((SF + OF) OR ZF) = 0
JNG/JLE	((SF × + OR OF) OR ZF) = 1
JB/JNAE	CF = 1
JNB/JAE	CF = 0
JA/JNBE	(CF OR ZF) = 0
JNA/JBE	(CF OR ZF) = 1

The meanings of JE/JZ (ZF = 1) and JNE/JNZ (ZF = 0) are obvious.

If two signed numbers are compared and if the "less than" condition is satisfied (SF = 1) with no overflow (OF = 0), the processor after execution of the JL/JNGE instruction branches to a label with the specified displacement; otherwise the next instruction is executed. In this case, SF + OF = 1. Similarly, the meaning of JNL/JGE can be explained.

The processor when executing the JG/JNLE instruction (after comparing two signed numbers) branches to the label with the specified displacement if OF = 0 and SF = 0 (i.e., result is greater than zero). In this case, ((SF + OF) OR ZF) = 0. Similarly, the meaning of JNG/JLE can be explained.

The overflow flag OF is not involved while considering unsigned numbers. After comparing two unsigned numbers, the processor when executing the JB/JNAE instruction branches to a label with the specified displacement if result is below zero. (i.e., CF = 1 indicating a borrow). Similarly, the meaning of JNB/JAE can be explained.

After comparing two unsigned numbers, the processor when executing the JA/JNBE instruction branches to a label with the specified displacement if result is above zero (CF = 0) or not equal to zero (ZF = 0). Similarly, the meaning of JNA/JBE can be explained.

Tables 3.A-10, 3.A-11, and 3.A-12 are self-explanatory.

3.5 8086 INSTRUCTION FORMAT

The 8086 instruction sizes vary from one to six bytes. The general 8086 instruction format is shown in Figure 3.5. The op code, register direction bit (D) and data size bit (W) in byte 1 are defined by Intel as follows:

- Op code occupies six bits and it defines the operation to be carried out by the instruction.
- Register Direction bit (D) occupies one bit. It defines whether the register operand in byte 2 is the source or destination operand. D = 1 specifies that the register operand is the destination operand; on the other hand, D = 0 indicates that the register is a source operand.
- Data size bit (W) defines whether the operation to be performed is on 8- or 16-bit data. W = 0 indicates 8-bit operation while W = 1 specifies 16-bit operation.
- The second byte of the instruction usually identifies whether one of

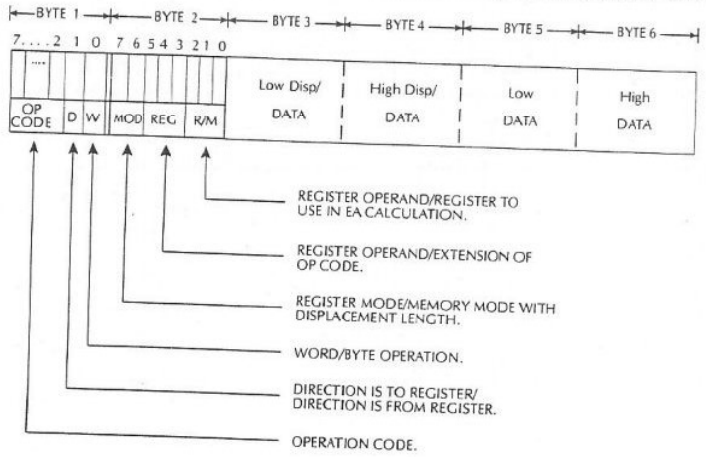


FIGURE 3.5 8086 Instruction format.

the operands is in memory or whether both are registers: this byte contains three fields. These are the Mode (MOD) field, the Register (REG) field, and the Register/Memory (R/M) field and are defined as follows.

The 2-bit MOD field specifies whether the operand is in register or memory as follows:

MOD	Interpretation
00	Memory mode with no displacement follows except for 16-bit displacement when R/M = 110
01	Memory mode with 8-bit displacement
10	Memory mode with 16-bit displacement
11	Register mode (no displacement)

REG field occupies 3 bits. It defines the register for the first operand which is specified as the source or destination by the D-bit (byte 1). The definition of REG and W fields are given below:

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

The R/M field occupies 3 bits. The R/M field along with the MOD field defines the second operand as shown below:

MOD 11			Effective address calculation			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD 01	MOD 10
000	AL	AX	000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	CL	CX	001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	DL	DX	010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	BL	BX	011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	AH	SP	100	(SI)	(SI) + D8	(SI) + D16
101	CH	BP	101	(DI)	(DI + D8	(DI) + D16

MOD 11			Effective address calculation			
R/M	W = 0	W = 1	R/M	MOD = 00	MOD 01	MOD 10
110	DH	SI	110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	BH	DI	111	(BX)	(BX) + D8	(BX) + D16

In the above, encoding of the R/M field depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated.

- Bytes 3 through 6 of an instruction are optional fields that normally contain the displacement value of a memory operand and/or the actual value of an immediate constant operand. As an example, consider the instruction MOV CH, BL. This instruction transfers the 8-bit content of BL into CH. We will determine the machine code of this instruction. The 6-bit op code for this instruction is 100010_2. The D-bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand. Let us define the BL in the REG field of byte 2. D = 0 indicates that the REG field of the next byte is the source operand. The W-bit of byte 1 is 0 since this is a byte operation.

  In byte 2, since the second operand is a register, MOD field is 11_2. The R/M field = 101_2 specifies that the destination register is CH and, therefore, R/M = 101_2. Hence the machine code for MOV CH, BL is $\underbrace{10001001}_{\text{BYTE 1}}\ \underbrace{11011101}_{\text{BYTE 2}}$

  = $89DD_{16}$.

  As another example, consider SUB BX, [DI]. This instruction subtracts the 16-bit content of memory location addressed by DI and DS from BX. The 6-bit op code for SUB is 001010_2.

  D = 1 so that the REG field of byte 2 is the destination operand and W = 1 indicates 16-bit operation. Therefore, byte 1 = $\underbrace{0010}_{2}\ \underbrace{1011}_{B}$

  = $2B_{16}$. Hence the machine code = 2 B1D$_{16}$.

3.6 8086 ASSEMBLER-DEPENDENT INSTRUCTIONS

Some 8086 instructions do not define whether an 8-bit or 16-bit operation is to be executed. Instructions with one of the 8086 registers as an operand typically define the operation as 8-bit or 16-bit based on the register size. An example is MOV CL, [BX] which moves an 8-bit number with the offset defined by [BX] in DS into register CL; MOV CX, [BX], on the other hand, moves the number from offsets [BX] and [BX + 1] in DS into CX.

Instructions with single-memory operand may define 8-bit or 16-bit operation by adding B for byte or W for word with the mnemonic. Typical examples are MULB [BX] and CMPB [ADDR]. The string instructions may define this in two ways. Typical examples are MOVSB or MOVS BYTE for 8-bit and MOVSW or MOVS WORD for 16-bit. Memory offsets can also be specified by including .BYTE PTR for 8-bit and .WORD PTR for 16-bit with the instruction. Typical examples are INC .BYTE PTR [BX] and INC.WORD PTR [BX].

3.7 ASM-86 ASSEMBLER PSEUDOINSTRUCTIONS

The ASM-86 is the assembler written by Intel for the 8086 microprocessor. The ASM-86 allows the programmer to assign the values of CS, DS, SS, and ES. One of the requirements of the ASM-86 assembler is that a variable's type must be declared as byte (8-bit), word (16-bit), or double word (4 bytes or 2 words) before using in a program. Some examples are included below:

```
START DB 0  ;  START is declared
            ;  as a byte offset
            ;  and initialized to zero.

BEGIN DW 0  ;  BEGIN is declared
            ;  as a word offset
            ;  and initialized to zero.
```



```
NAME DD 0   ;  NAME is declared
            ;  as a double word
            ;  (4 bytes) offset
            ;  and initialized to zero.
```

The EQU directive can be used to assign a name to constants. For example, the statement JOHN EQU 21H directs the assembler to assign the value 21H every time it finds JOHN in the program. This means that the assembler reads the statement MOV BH, JOHN as MOV BH, 21H. As mentioned before, DB, DW, and DD are the ASM-86 directives used to assign names and specify data types for variables in a program. For example, after execution of the statement BOB DW 2050H, the ASM-86 assembler assigns 50H to the offset name BOB and 20H to the offset name BOB + 1. This means that the program can use the instruction MOV BX, BOB to load the 16-bit content of memory starting at the offset BOB in DS into BX. The DW sets aside storage for a word in memory and gives the starting of this word the name BOB. Next, typical 8086 addressing mode examples for the ASM-86 assembler are given below:

```
MOV AH, BL          ;  Both source and
                    ;  destination are in
                    ;  register mode.

MOV CH, 8           ;  Source is in
                    ;  immediate mode
                    ;  and destination is in
                    ;  register mode.

MOV AX, START       ;  Source is in memory
                    ;  direct mode and
                    ;  destination is in
                    ;  register mode.

MOV CH, [BX]        ;  Source is in register
                    ;  indirect mode and
                    ;  destination is in register
                       mode.

MOV [SI], AL        ;  Source is in register
                    ;  mode and destination
                    ;  is in register indirect
                    ;  mode.
```

```
MOV [DI], BH        ;  Source is in register
                    ;  mode and destination
                    ;  is in register
                    ;  indirect mode.

MOV BH, VALUE       ;  Source is in register
 [DI]               ;  indirect with
                    ;  displacement mode
                    ;  and destination is
                    ;  in register mode
                    ;  VALUE is typically
                    ;  defined by the
                    ;  EQU directive prior
                    ;  to this instruction.

MOV AX, [DI + 4]    ;  Source is in
                    ;  indexed with
                    ;  displacement mode and
                    ;  destination is in
                    ;  register mode.

MOV SI, [BP + 2]    ;  Source is in
          [DI]      ;  bused indexed
                    ;  with displacement
                    ;  mode and
                    ;  destination is in
                    ;  register mode.

OUT 30H, AL         ;  Source is in
                    ;  register mode and
                    ;  destination is
                    ;  in direct port
                    ;  mode.

IN AX, DX           ;  Source is in
                    ;  indirect port mode and
                    ;  destination is
                    ;  in register mode.
```

Note that in the above, START must be previously defined by a data allocation statement such DB and DW.

In the following, typical ASM-86 assembler directives such as SEGMENT, ENDS, ASSUME, and DUP will be discussed.

3.7.1 SEGMENT AND ENDS

A section of a program or a data array can be defined by the SEGMENT and ENDS directives as follows:

```
JOHN    SEGMENT

X1      DB      0
X2      DB      0
X3      DB      0
JOHN    ENDS
```

The segment name is JOHN. The assembler will assign a numeric value to JOHN corresponding to the base value of the data segment. The programmer must use the 8086 instructions to load JOHN into DS as follows:

```
MOV BX, JOHN
MOV DS, BX
```

Note that the segment registers must be loaded via a 16-bit register such as AX or by content of a memory location.

3.7.2 ASSUME DIRECTIVE

As mentioned before, the 8086, at any time, can directly address four physical segments which include a code segment, a data segment, a stack segment, and an extra segment. The 8086 rnay contain a number of logical segments containing codes, data, and stack. The ASSUME pseudoinstruction assigns a logical segment to a physical segment at any given time. That is, the ASSUME directive tells the ASM-86 assembler what addresses will be in the segment registers at execution time.

For example, the statement ASSUME CS: PROGRAM 1, DS: DATA 1, SS: STACK 1 directs the assembler to use the logical code segment PROGRAM 1 as CS containing the instructions, the logical data segment DATA 1 as DS containing data, and the logical STACK segment STACK 1 as SS containing the stack.

3.7.3 DUP DIRECTIVE

The DUP directive can be used to initialize several locations to zero. For example, the statement START DW 4 DUP (0) reserves four words

starting at the offset START in DS and initializes them to zero. The DUP directive can also be used to reserve several locations which need not be initialized. A question mark must be used with DUP in this case. For example, the statement BEGIN DB 100 DUP (?) reserves 100 bytes of uninitialized data space to an offset BEGIN in DS. Note that BEGIN should be typed in the label field, DB in the op field, and 100 DUP (?) in the operand field.

A typical example illustrating the use of these directives is given below:

```
DATA 1      SEGMENT
ADDR 1      DW 3005H
ADDR 2      DW 2003H
DATA 1      ENDS
STACK 1     SEGMENT
DW          60 DUP (0)          ;  Assign 60₁₀ words
                                ;  of stack with zeros.
STACK-TOP   LABEL WORD          ;  Initialize Stack-Top
                                ;  to the next
STACK 1     ENDS                ;  location after the
                                ;  top of the stack.
CODE 1      SEGMENT
            ASSUME CS: CODE 1, DS: DATA 1, SS:
            STACK 1
            MOV AX, STACK 1
            MOV SS, AX
            LEA SP, STACK-TOP
            MOV AX, DTATA 1
            MOV DS, AX
            LEA SI, ADDR 1
            LEA DI, ADDR 2
              -
              -          }      Main Program
              -                    Body
              -          }
CODE 1      ENDS
```

Note that LABEL is a directive used to initialize STACK-TOP to the next location after the top of the stack. The statement STACK-TOP LABEL WORD gives the name STACK-TOP to the next address after the 60 words are set aside for the stack. The WORD in this statement indicates that PUSH into and POP from the stack are done as words.

In the above program, the ASSUME directive tells the assembler the

names of the logical segments to use as code segment, data segment, and stack segment. The extra segment can be assigned a name in a similar manner. When the instructions are executed, the displacements in the instructions along with the segment register contents are used by the assembler to generate the 20-bit physical addresses. The segment register, other than the code segment, must be initialized by instructions before they are used to access data.

When the ASM-86 assembler translates an assembly language program, it computes the displacement, or offset, of each instruction code byte from the start of a logical segment that contains it. For example, in the above program the CS: CODE 1 in the ASSUME statement directs the assembler to compute the offsets or displacements by the following instructions from the start of the logical segment CODE 1. This means that when the program is run, the CS will contain the 16-bit value where the logical segment CODE 1 is located in memory. The assembler keeps track of the instruction byte displacements which are loaded into IP. The 20-bit physical address generated from CS and IP are used to fetch each instruction.

Another example to store data bytes in a data segment and to allocate stack is given in the following:

```
DSEG        SEGMENT
ARRAY       DB 02H, F1H, A2H    ;  Store 3 bytes
                                ;  of data in an
DSEG        ENDS                ;  address defined
            -                   ;  by DSEG as DS
            -                   ;  and ARRAY as
            -                   ;  offset
SSEG        SEGMENT
            DW 10 DUP (0)       ;  Allocate
                                ;  10 word stack
STACK-TOP   LABEL WORD          ;  Label initial
                                ;  TOS
SSEG        ENDS
            -
            -
            -
            -
            MOV AX, DSEG        ;  Initialize
            MOV DS, AX          ;  DS
            MOV AX, SSEG        ;  Initialize
            MOV SS, AX          ;  SS
```

```
MOV SP, STACK-TOP    ; Initialize SP
-
-
-
```

A logical segment is not normally given a physical starting address when it is declared. After the program is assembled, the programmer uses the linker to assign physical address.

Example 3.1

Determine the effect of each one of the following 8086 instructions:

i) `PUSH [BX]`
ii) `DIV DH`
iii) `CWD`
iv) `MOVSB`
v) `MOV START [BX], AL`

Assume the following data prior to execution of each one of the above instructions independently. Assume all numbers in hexadecimal.

[DS]	= 3000H	[SI]	= 0400H
[ES]	= 5000H	[DI]	= 0500H
[DX]	= 0400H	DF	= 0
[SP]	= 5000H	[BX]	= 6000H
[SS]	= 6000H	Value of START	= 05H
[AX]	= 00A9H		
[36000H]	= 02H, [36001H] = 03H		
[50500H]	= 05H		
[30400H]	= 02H, [30401H] = 03H		

Solution

i) 20-bit physical memory addressed by DS and BX = 36000H. 20-bit physical location pointed to by SP and SS = 65000H. PUSH [BX] pushes [36001H] and [36000H] into stack locations 64FFFH and 64FFEH, respectively. The SP is then decremented by 2 to contain the 20-bit physical address 64FFEH (SS = 6000H, SP = 4FFEH). Therefore, [64FFFH] = 03H and [64FFEH] = 02H.

ii) Before unsigned division, [DX] = 0400H, DH contains 04H, and [AX] = 00A9H = 169_{10}. After DIV DH, [AH] = remainder = 01H and [AL] = quotient = 2AH = 42_{10}.

iii) CWD sign extends AX register into the DX register. Since the sign bit of [AX] = 0, after CWD [DX AX] = 000000A9H.

iv) MOVSB moves the content of memory addressed by the source [DS] and [SI] to the destination addressed by [ES] and [DI] and then it increments SI and DI by 1 for byte move. Since DF = 0, [DS] = 3000H, [SI] = 0400H, [ES] = 5000H, [DI] = 0500H, and the content of physical memory location 30400H is moved to physical memory 50500H. Since [30400H] = 02H, the location 50500H will also contain 02H. Since DF = 0 after MOVSB, [SI] = 0401H, [DI] = 0501H.

v) Since [BX] = 6000H, [DS] = 3000H, START = 05H, and the physical memory for destination = 36005 H. After MOV START [BX], AL, memory location 36005H will contain A9H.

Example 3.2

Write 8086 assembly program to clear 100_{10} consecutive bytes. Assume CS and DS are already initialized.

Solution

```
        LEA BX, ADDR     ;  Initialize BX
        MOV CX, 100      ;  Initialize loop count
START   MOV [BX], 00H    ;  Clear memory byte
        INC BX           ;  Update pointer
        LOOP START       ;  Decrement CX and loop
        HLT
```

Example 3.3

Write 8086 assembly program to compute $\sum_{i=1}^{N} XiYi$ where Xi and Yi are signed 8-bit numbers. N = 100. Assume CS and DS are already initialized. Assume no overflow.

Solution

```
        MOV CX, 100      ;  Initialize Loop count
        LEA BX, ADDR 1   ;  Load ADDR 1 into BX
        LEA SI, ADDR 2   ;  Load ADDR 2 into SI
        MOV DX, 0000H    ;  Initialize sum to zero
START   MOV AL, [BX]     ;  Load data into AL
        IMUL [SI]        ;  Signed multiplication
        ADD DX, AX       ;  Sum XiYi
```

```
        INC BX          ; Update
        INC SI          ; Pointers
        LOOP START      ; Decrement CX and Loop
        HLT
```

Example 3.4

Write 8086 assembly language program to add two words; each word contains four packed BCD digits. The first word is stored in two consecutive locations with the low byte at the offset pointed by SI at 0500H, while the second word is stored in two consecutive locations with the low byte pointed by BX at the offset 1000H. Store the result in BX. Assume CS and DS are already initialized.

Solution

```
        MOV CX, 2       ; Initialize loop count
        MOV SI, 0500H   ; Initialize SI
        MOV BX, 1000H   ; Initialize BX
        CLC             ; Clear carry
START   MOV AL, [SI]    ; Move data
        ADC AL, [BX]    ; Perform addition
        DAA             ; BCD adjust
        MOV [BX], AL    ; Store result
        INC SI          ; Update
        INC BX          ; Pointers
        LOOP START      ; Decrement CX and loop
        HLT
```

Example 3.5

Write an 8086 assembly language program to add two words; each contains two ASCII digits. The first word is stored in two consecutive locations with the low byte pointed to by SI at offset 0300H, while the second byte is stored in two consecutive locations with the low byte pointed to by DI at offset 0700H. Store the result in DI. Assume CS and DS are already initialized.

Solution

```
        MOV CX, 2       ; Initialize loop count
        MOV SI, 0300H   ; Initialize SI
```



```
        MOV DI, 0700H   ; Initialize DI
        CLC             ; Clear carry
START   MOV AL, [SI]    ; Move data
        ADC AL, [DI]    ; Perform addition
        AAA             ; ASCII adjust
        MOV [DI], AL    ; Store result
        INC SI          ; Update
        INC DI          ; Pointers
        LOOP START      ; Decrement CX and Loop
        HLT
```

Example 3.6

Write an 8086 assembly language program to compare a source string of 50_{10} words pointed to by an offset of 2000H in DS with a destination string pointed to by an offset 3000H in DS. The program should be halted as soon as a match is found or the end of string is reached. Assume CS and DS are initialized.

Solution

```
        MOV SI, 2000H   ; Initialize SI
        MOV DI, 3000H   ; Initialize DI
        MOV CX, 50      ; Initialize CX
        CLD             ; DF is cleared
                        ; so that SI and DI
                        ; will autoincrement
                        ; after compare
        REPNE CMPSW     ; Repeat CMPSW
                        ; unit CX = 0 or
                        ; until compared
                        ; words are equal.
        HLT             ; STOP
```

Example 3.7

Write a subroutine in 8086 assembly language which can be called by a main program in a different code segment. The subroutine will multiply a signed 16-bit number in CX by a signed 8-bit number in AL. The main program will call this subroutine, store the result in two consecutive memory words, and stop. Assume SI and DI contain the signed 8-bit and 16-bit data, respectively.

Solution

Main Program

```
MAIN PROG SEGMENT                ;  Give assembler
          ASSUME CS: MAIN PROG   ;  CS name
          MOV AX, 5000H          ;  Initialize
          MOV DS, AX             ;  DS to 5000H
          MOV AX, 6000H          ;  Initialize
          MOV SS, AX             ;  SS to 6000H
          MOV SP, 0020H          ;  Initialize SP
          MOV BX, 2000H          ;  Initialize BX
          MOV AL, [SI]           ;  Move 8-bit data
          MOV CX, [DI]           ;  Move 16-bit data
          CALL MUL               ;  Call multiplica-
                                 ;  tion
                                 ;  subroutine
          MOV [BX], DX           ;  Store high
                                 ;  word of result
          MOV [BX + 2], AX       ;  Store low
                                 ;  word of result
          HLT
MAIN PROG ENDS
```

Subroutine

```
SUBR      SEGMENT
          ASSUME  CS: SUBR
MUL       PROC    FAR            ;  Must be called
                                 ;  from another code
                                 ;  segment
CBW                              ;  Sign extend AL
IMUL CX                          ;  [DX] [AX] ← [AX]
* [CX]
RET                              ;  Return
MUL       ENDP                   ;  End of Procedure
                                 ;  mul
SUBR      ENDS
```

Note that in the above programs SEGMENT, ASSUME, ENDP, and ENDS are used as assembler directives.

3.8 SYSTEM DESIGN USING 8086

This section covers the basic concepts associated with interfacing the 8086 to its support chips such as memory and I/O. Topics such as timing diagrams and 8086 pins and signals will also be included.

3.8.1 PINS AND SIGNALS

The 8086 pins and signals are shown in Figure 3.6. Unless otherwise indicated, all 8086 pins are TTL compatible. As mentioned before, the 8086 can operate in two modes. These are minimum mode (uniprocessor system — single 8086) and maximum mode (multiprocessor system — more than one 8086). MN/$\overline{MX}$ is an input pin used to select one of these modes. When MN/$\overline{MX}$ is HIGH, the 8086 operates in the minimum mode. In this mode, the 8086 is configured (that is, pins are defined) to support small, single processor systems using a few devices that use the system bus.

When MN/$\overline{MX}$ is LOW, the 8086 is configured (that is, pins are defined in the maximum mode) to support multiprocessor systems. In this case, the Intel 8288 bus controller is added to the 8086 to provide bus controls and compatibility with the multibus architecture. Note that in a particular application, the MN/$\overline{MX}$ must be tied to either HIGH or LOW. AD0-AD15 lines are a 16-bit multiplexed address/data bus. During the first clock cycle AD0-AD15 are the low order 16 bits of address. The 8086 has a total of 20 address lines. The upper four lines are multiplexed with the status signals for the 8086. These are the A16/S3, A17/S4, A18/S5, and A19/S6. During the first clock period of a bus cycle (read or write cycle), the entire 20-bit address is available on these lines. During all other clock cycles for memory and I/O operations, AD15-AD0 contain the 16-bit data, and S3, S4, S5, and S6 become status lines. S3 and S4 lines are decoded as follows:

A17/S4	A16/S3	Function
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data segment

Common Signals

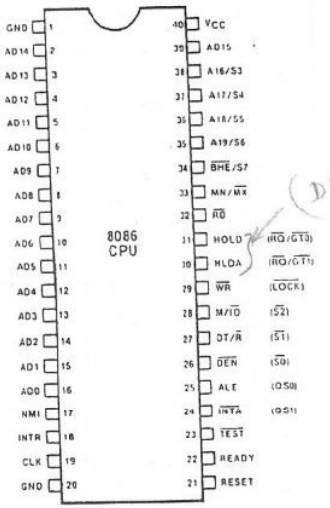
Name	Function	Type
AD15-AD0	Address/Data Bus	Bidirectional, 3-State
A19/S6-A16/S3	Address/Status	Output, 3-State
$\overline{BHE}$/S7	Bus High Enable/Status	Output, 3-State
MN/$\overline{MX}$	Minimum/Maximum Mode Control	Input
$\overline{RD}$	Read Control	Output, 3-State
$\overline{TEST}$	Wait On Test Control	Input
READY	Wait State Control	Input
RESET	System Reset	Input
NMI	Non-Maskable Interrupt Request	Input
INTR	Interrupt Request	Input
CLK	System Clock	Input
Vcc	+5V	Input
GND	Ground	

Minimum Mode Signals (MN/MX = Vcc)

Name	Function	Type
HOLD	Hold Request	Input
HLDA	Hold Acknowledge	Output
$\overline{WR}$	Write Control	Output, 3-State
M/$\overline{IO}$	Memory/IO Control	Output, 3-State
DT/$\overline{R}$	Data Transmit/Receive	Output, 3-State
$\overline{DEN}$	Data Enable	Output, 3-State
ALE	Address Latch Enable	Output
$\overline{INTA}$	Interrupt Acknowledge	Output

Maximum Mode Signals (MN/MX = GND)

Name	Function	Type
$\overline{RQ}$/GT1, 0	Request/Grant Bus Access Control	Bidirectional
$\overline{LOCK}$	Bus Priority Lock Control	Output, 3-State
$\overline{S2}$-$\overline{S0}$	Bus Cycle Status	Output, 3-State
QS1, QS0	Instruction Queue Status	Output



MAXIMUM MODE PIN FUNCTIONS (e.g., $\overline{LOCK}$) ARE SHOWN IN PARENTHESES

FIGURE 3.6 Pin definitions.

Therefore, after the first clock cycle of an instruction execution, the A17/S4 and A16/S3 pins specify which segment register generates the segment portion of the 8086 address. Thus, by decoding these lines and then using the decoder outputs as chip selects for memory chips, up to 4 megabytes (one megabyte per segment) can be provided. This provides a degree of protection by preventing erroneous write operations to one segment from overlapping into another segment and destroying information in that segment. A18/S5 and A19/S6 are used as A18 and A19, respectively, during the first clock period of an instruction execution. If an I/O instruction is executed, they stay low during the first clock period. During all other cycles, A18/S5 indicates the status of the 8086 interrupt enable flag and A19/S6 becomes S6, and a low A19/S6 pin indicates that the 8086 is on the bus. During a "Hold Acknowledge" clock period, the 8086 tristates the A19/S6 pin and thus allows another bus master to take control of the system bus.

The 8086 tristates AD0-AD15 during Interrupt Acknowledge or Hold Acknowledge cycles.

$\overline{BHE}$/S7 is used as $\overline{BHE}$ (Bus High Enable) during the first clock cycle of an instruction execution. The 8086 outputs a low on this pin during read, write, and interrupt acknowledge cycles in which data are to be transferred in a high-order byte (AD15-AD8) of the data bus. $\overline{BHE}$ can be used in conjunction with AD0 to select memory banks. A thorough discussion is provided later. During all other cycles $\overline{BHE}$/S7 is used as S7 and the 8086 maintains the output level ($\overline{BHE}$) of the first clock cycle on this pin.

$\overline{RD}$ is LOW whenever the 8086 is reading data from memory or an I/O location.

$\overline{TEST}$ is an input pin and is only used by the WAIT instruction. The 8086 enters a wait state after execution of the WAIT instruction until a LOW is seen on the $\overline{TEST}$ pin. This input is synchronized internally during each clock cycle on the leading edge of the CLK pin.

INTR is the maskable interrupt input. This line is not latched and, therefore, INTR must be held at a HIGH level until recognized to generate an interrupt.

NMI is the nonmaskable interrupt input activated by a leading edge.

RESET is the system reset input signal. This signal must be high for at least four clock cycles to be recognized, except after power-on which requires a 50-μ reset pulse. It causes the 8086 to initialize registers DS, SS, ES, IP and flags to all zeros. It also initializes CS to FFFFH. Upon removal of the RESET signal from the RESET pin, the 8086 will fetch its next instruction from 20-bit physical address FFFF0H (CS = FFFFH, IP = 0000H).

When the 8086 detects the positive going edge of a pulse on RESET, it stops all activities until the signal goes LOW. When the reset is low, the 8086 initializes the system as follows:

8086 component	Content
Flags	Clear
IP	0000H
CS	FFFFH
DS	0000H
SS	0000H
ES	0000H
Queue	Empty

The reset signal to the 8086 can be generated by the 8284. The 8284 has a Schmitt Trigger input ($\overline{RES}$) for generating reset from a low active external reset.

To guarantee reset from power-up, the reset input must remain below 1.05 volts for 50 microseconds after Vcc has reached the minimum supply voltage of 4.5V. The $\overline{RES}$ input of the 8284 can be driven by a simple RC circuit as shown in Figure 3.7.

The values of R and C can be selected as follows:

$$Vc\ (t) = V\ (1 - exp - (t/RC))$$

where t = 50 microseconds, V = 4.5 V, Vc = 1.05V, and RC = 188 microseconds. For example, if C is chosen arbitrarily to be 0.1 μF, then R = 1.88 KΩ.

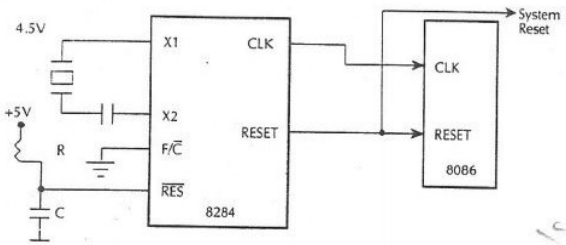


FIGURE 3.7 8086 reset and system reset.

As mentioned before, the 8086 can be configured in either minimum or maximum mode using the MN/$\overline{MX}$ input pin. In minimum mode, the 8086 itself generates all bus control signals. These signals are

- DT/$\overline{R}$ (Data Transmit/Receive). DT/$\overline{R}$ is an output signal required in minimum system that uses an 8286/8287 data bus transceiver. It is used to control direction of data flow through the transceiver.
- $\overline{DEN}$ (Data Enable) is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver.
- $\overline{DEN}$ is active LOW during each memory and I/O access and for $\overline{INTA}$ cycles.
- ALE (Address Latch Enable) is an output signal provided by the 8086 and can be used to demultiplex the AD0-AD15 into A0-A15 and D0-D15 at the falling edge of ALE. The 8086 ALE signal is same as the 8085 ALE.
- M/$\overline{IO}$. This 8086 output signal is similar to the 8085 IO/$\overline{M}$. It is used to distinguish a memory access (M/$\overline{IO}$ = HIGH) from an I/O access (M/$\overline{IO}$ = LOW). When the 8086 executes an I/O instruction such as IN or OUT, it outputs a LOW on this pin. On the other hand, the 8086 outputs HIGH on this pin when it executes a memory reference instruction such as MOVE AX, [SI].
- $\overline{WR}$. The 8086 outputs LOW on this pin to indicate that the processor is performing a write memory or write I/O operation, depending on the M/$\overline{IO}$ signal.
- $\overline{INTA}$. The 8086 $\overline{INTA}$ is similar to the 8085 $\overline{INTA}$. For Interrupt Acknowledge cycles (for INTR pin), the 8086 outputs LOW on this pin.
- HOLD (input), HLDA (output). These pins have the same purpose as the 8085 HOLD/HLDA pins and are used for DMA. A HIGH on the HOLD pin indicates that another master is requesting to take over the system bus. The processor receiving the HOLD request will output HLDA high as an acknowledgment. At the same time, the processor tristates the system bus. Upon receipt of LOW on the HOLD pin, the processor places LOW on the HLDA pin. HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.
- CLK (input) provides the basic timing for the 8086 and bus controller.

The maximum clock frequencies of the 8086-4, 8086, and 8086-

2 are 4 MHz, 5 MHz, and 8 MHz, respectively. Since the design of these processors incorporates dynamic cells, a minimum frequency of 2 MHz is required to retain the state of the machine. The 8086-4, 8086, and 8086-2 will be referred to as 8086 in the following. Minimum frequency requirement, single stepping, or cycling of the 8086 may not be accomplished by disabling the clock. Since the 8086 does not have on-chip clock generation circuitry, an 8284 clock generator chip must be connected to the 8086 CLK pin as shown in Figure 3.8.

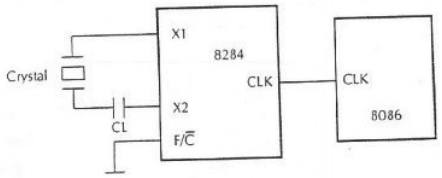


FIGURE 3.8 8284 clock generator connections to the 8086.

The crystal must have a frequency three times the 8086 internal frequency. That is, the 8284 divides the crystal clock frequency by 3. In other words, to generate 5 MHz 8086 internal clock, the crystal clock must be 15 MHz. To select the crystal inputs of the 8284 as the frequency source for clock generation, the F/$\overline{C}$ input must be strapped to ground. This strapping option allows either the crystal or the external frequency input as the source for clock generation. When selecting a crystal for use with the 8284, the crystal series resistance should be as low as possible. The oscillator delays in the 8284 appear as inductive elements to the crystal and cause the 8284 to run at a frequency below that of the pure series resonance; a capacitor CL should be placed in series with the crystal and the X2 pin of the 8284. This capacitor cancels the inductive element. The impedance of the capacitor XC = $1/2\pi FCL$, where F is the crystal frequency. It is recommended that the crystal series resistance plus XCL be kept less than 1 kohm. As the crystal frequency increases, CL should be decreased. For example, a 12-MHz crystal may require CL ≈ 24 PF, while 22 MHz may require CL ≈ 8 PF. If very close correlation with the pure series resonance is not necessary, a nominal CL value of 12 to 15 PF may be used with a 15-MHz crystal. Two crystal manufacturers recommended by Intel are Crystle Corp. model CY 15A (15 MHz) and CTS Knight Inc. model CY 24 A (24 MHz).

Note that the 8284 can be used to generate the 8086 READY input signal based on inputs from slow memory and I/O devices which are not capable of transferring information at the 8086 rate.

In the maximum mode, some of the 8086 pins in the minimum mode are redefined. For example, pins HOLD, HLDA, $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, and $\overline{INTA}$ in the minimum mode are redefined as $\overline{RQ/GT0}$, $\overline{RQ/GT1}$, $\overline{LOCK}$, $\overline{S2}$, $\overline{S1}$, $\overline{S0}$, QS0, and QS1, respectively. In maximum mode, the 8288 bus controller decodes the status information from $\overline{S0}$, $\overline{S1}$, $\overline{S2}$ to generate bus timing and control signals required for a bus cycle. $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ are 8086 outputs and are decoded as follows:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Inactive

$\overline{RQ/GT0}$, $\overline{RQ/GT1}$. These request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional, with $\overline{RQ/GT0}$ having higher priority than $\overline{RQ/GT1}$. These pins have internal pull-up resistors so that they may be left unconnected. The request/grant function of the 8086 works as follows:

1. A pulse (one clock wide) from another local bus master ($\overline{RQ/GT0}$ or $\overline{RQ/GT1}$ pins) indicates a local bus request to the 8086.
2. At the end of 8086 current bus cycle, a pulse (one clock wide) from the 8086 to the requesting master indicates that the 8086 has relinquished the system bus and tristated the outputs. Then the new bus master subsequently relinquishes control of the system bus by sending a LOW on $\overline{RQ/GT0}$ or $\overline{RQ/GT1}$ pins. The 8086 then regains bus control.

$\overline{LOCK}$. The 8086 outputs LOW on the $\overline{LOCK}$ pin to prevent other bus masters from gaining control of the system bus. The $\overline{LOCK}$ signal

is activated by the 'LOCK' prefix instruction and remains active until the completion of the instruction that follows.

- QS1, QS0. The 8086 outputs to QS1 and QS0 pins to provide status to allow external tracking of the internal 8086 instruction queue as follows:

QS1	QS0	
0	0	No operation
0	1	First byte of op code from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

QS0 and QS1 are valid during the clock period following any queue operation. The 8086 can be operated from a +5V to +10V power supply. There are two ground pins on the chip to distribute power for noise reduction.

3.8.2 8086 BASIC SYSTEM CONCEPTS

This section describes basic concepts associated with 8086 bus cycles, address and data bus, system data bus, and multiprocessor environment.

3.8.2.a 8086 Bus Cycle

In order to communicate with external devices via the system bus for transferring data or fetching instructions, the 8086 executes a bus cycle. The 8086 basic bus cycle timing diagram is shown in Figure 3.9. The minimum bus cycle contains four CPU clock periods called T States. The bus cycle timing diagram depicted in Figure 3.9 can be described as follows:

1. During the first T State (T1), the 8086 outputs the 20-bit address computed from a segment register and an offset on the multiplexed address/data/status bus.
2. For the second T State (T2), the 8086 removes the address from the bus and either tristates or activates the AD15-AD0 lines in preparation for reading data via AD15-AD0 lines during the T3 cycle. In case of a write bus cycle, the 8086 outputs data on AD15-AD0 lines. Also, during T2, the upper four multiplexed bus lines switch from address (A19-A16) to bus cycle status (S6, S5, S4, S3). The 8086 outputs LOW on $\overline{RD}$ (for read cycle) or $\overline{WR}$ (for write cycle) during portions of T2, T3, and T4.

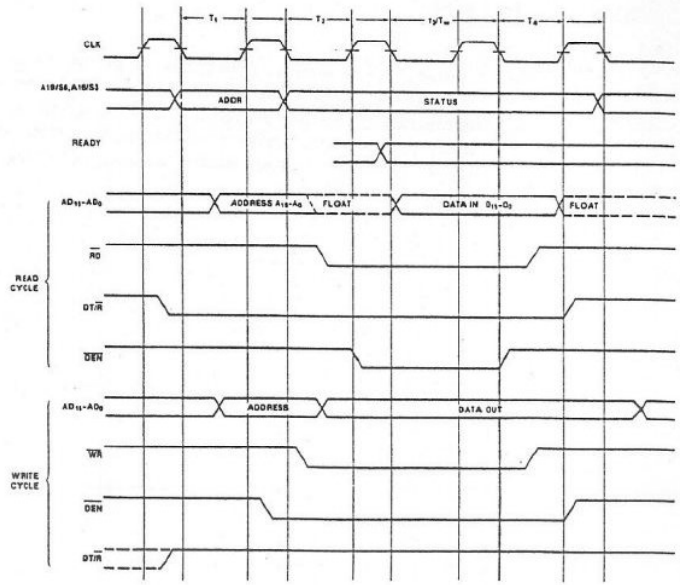


FIGURE 3.9 Basic 8086 bus cycle.

3. During T3, the 8086 continues to output status information on the four A19-A16/S6-S3 lines and will either continue to output write data or input read data to or from AD15-AD0 lines. If the selected memory or I/O device is not fast enough to transfer data at the 8086, the memory or I/O device activates the 8086's READY input line low by the start of T3. This will force the 8086 to insert additional clock cycles (wait states TW) after T3. Bus activity during TW is the same as T3. When the selected device has had sufficient time to complete the transfer, it must activate the 8086 READY PIN HIGH. As soon as the TW clock periods end, the 8086 executes the last bus cycle, T4. The 8086 will latch data on AD15-AD0 lines during the last wait state or during T3 if no wait states are requested.

4. During T4, the 8086 disables the command lines and the selected memory and I/O devices from the bus. Thus, the bus cycle is terminated in T4. The bus cycle appears to devices in the system as an asynchronous event consisting of an address to select the device, a register or memory location within the device, a read strobe, or a write strobe along with data.
5. $\overline{DEN}$ and DT/$\overline{R}$ pins are used by the 8286/8287 transceiver in a minimum system. During the read cycle, the 8086 outputs $\overline{DEN}$ LOW during part of T2 and all of T3 cycles. This signal can be used to enable the 8286/8287 transceiver. The 8086 outputs LOW on the DT/$\overline{R}$ pin from the start of T1 and part of T4 cycles. The 8086 uses this signal to receive (read) data from the receiver during T3-T4. During a write cycle, the 8086 outputs $\overline{DEN}$ LOW during part of T1, all of T2 and T3, and part of T4 cycles. The signal can be used to enable the transceiver. The 8086 outputs HIGH on DT/$\overline{R}$ throughout the four bus cycles to transmit (write) data to the transceiver during T3-T4.

3.8.2.b 8086 Address and Data Bus Concepts

The majority of memory and I/O chips capable of interfacing to the 8086 require a stable address for the duration of the bus cycle. Therefore, the address on the 8086 multiplexed address/data bus during T1 should be latched. The latched address is then used to select the desired I/O or memory location. Note that the 8086 has a 16-bit multiplexed address and data bus, while the 8085's 8-bit data lines and LOW address byte are multiplexed. Hence, the multiplexed bus components of the 8085 family are not applicable to the 8086. To demultiplex the bus, the 8086 provides an ALE (Address Latch Enable) signal to capture the address in either the 8282 (noninverting) or 8283 (inverting) 8-bit bistable latches. These latches propagate the address through to the outputs while ALE is HIGH and latch the address in the following edge of ALE. This only delays address access and chip select decoding by the propagation delay of the latch. Figure 3.10 shows how the 8086 demultiplexes the address and data buses.

The programmer views the 8086 memory address space as a sequence of one million bytes in which any byte may contain an eight-bit data element and any two consecutive bytes may contain a 16-bit data element. There is no constraint on byte or word addresses (boundaries). The address space is physically implemented on a 16-bit data bus by dividing the address space into two banks of up to 512K bytes as shown in Figure 3.11. These banks can be selected by $\overline{BHE}$ and A0 as follows:

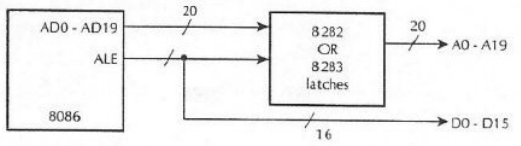


FIGURE 3.10 Separate address and data buses.

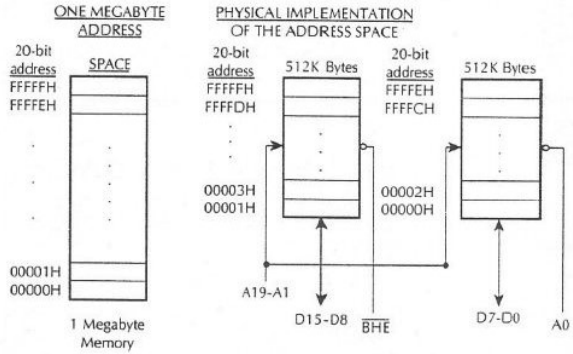


FIGURE 3.11 8086 memory.

$\overline{BHE}$	A0	Byte transferred
0	0	Both bytes
0	1	Upper byte to/from odd address
1	0	Lower byte to/from even address
1	1	None

One bank is connected to D7-D0 and contains all even addressed bytes (A0 = 0). The other bank is connected to D15-D8 and contains odd-addressed bytes (A0 = 1). A particular byte in each bank is addressed by A19-A1. The even-addressed bank is enabled by LOW A0 and data bytes transferred over D7-D0 lines. The 8086 outputs HIGH on $\overline{BHE}$ (Bus High Enable) and thus disables the odd-addressed bank. The 8086 outputs

LOW on $\overline{BHE}$ to select the odd-addressed bank and HIGH on A0 to disable the even-addressed bank. This directs the data transfer to the appropriate half of the data bus. Activation of A0 and $\overline{BHE}$ is performed by the 8086 depending on odd or even addresses and is transparent to the programmer. As an example, consider execution of the instruction MOV DH, [BX]. Suppose the 20-bit address computed by BX and DS is even. The 8086 outputs LOW on A0 and HIGH on $\overline{BHE}$. This will select the even-addressed bank. The content of the selected memory is placed on the D7-D0 lines by the memory chip. The 8086 reads this data via D7-D0 and automatically places it in DH. Next, consider accessing a 16-bit word by the 8086 with low byte at an even address as shown in Figure 3.12.

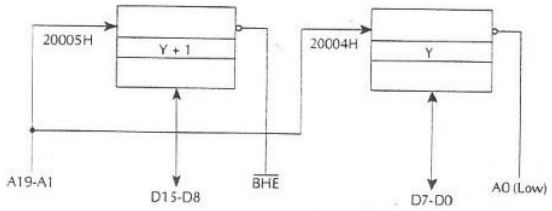


FIGURE 3.12 Even-addressed word transfer.

For example, suppose that the 8086 executes the instruction MOV [BX], CX. Assume [BX] = 0004H, [DS] = 2000H. The 20-bit physical address for the word is 20004H. The 8086 outputs LOW on both A0 and $\overline{BHE}$, enabling both banks simultaneously. The 8086 outputs [CL] to D7-D0 lines and [CH] to D15-D8 lines with $\overline{WR}$ LOW and M/IO HIGH. The enabled memory banks obtain the 16-bit data and write [CL] to location 20004H and [CH] to location 20005H. Next, consider accessing an odd-addressed 16-bit word by the 8086. For example, suppose the 20-bit physical address computed by the 8086 is 20005H. The 8086 accomplishes this transfer in two bus cycles. In the first bus cycle, the 8086 outputs HIGH on A0, LOW on $\overline{BHE}$, and thus enables the odd-addressed bank and disables the even-addressed bank. The 8086 also outputs LOW on $\overline{RD}$ and HIGH on M/$\overline{IO}$ pins. In this bus cycle, the odd memory bank places [20005H] on D15-D8 lines. The 8086 reads this data into CL. In the second bus cycle, the 8086 outputs LOW on A0, HIGH on $\overline{BHE}$, and thus enables the even-addressed bank and disables the odd-addressed bank.

The 8086 also outputs LOW on $\overline{RD}$ and HIGH on M/$\overline{IO}$ pins. The selected even-addressed memory bank places [20006H] on D7-D0 lines. The 8086 reads this data into CH. This is shown in Figure 3.13.

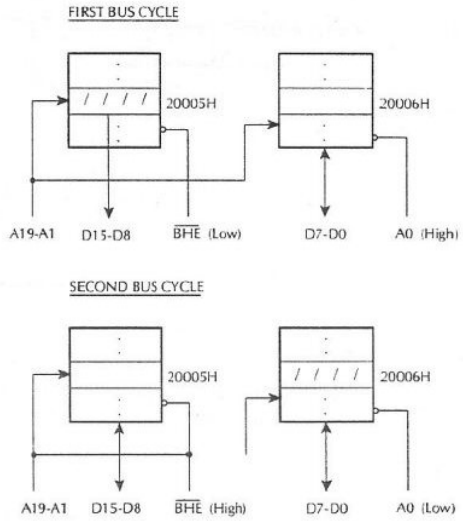


FIGURE 3.13. Odd-addressed word transfer.

During a byte read, the 8086 floats the entire D15-D0 lines during portions of T2 cycle even though data are expected on the upper or lower half of the data bus. As will be shown later, this action simplifies the chip select decoding requirements for ROMs and EPROMs. During a byte write, the 8086 will drive the entire 16-bit data bus. The information on the half of the data bus not transferring data is indeterminate. These concepts also apply to I/O transfers.

If memory of I/O devices are directly connected to the multiplexed bus, the designer must guarantee that the devices do not corrupt the address on the bus during T1. To avoid this, the memory or I/O devices should have an output enable controlled by the 8086 read signal. This is shown in Figure 3.14.

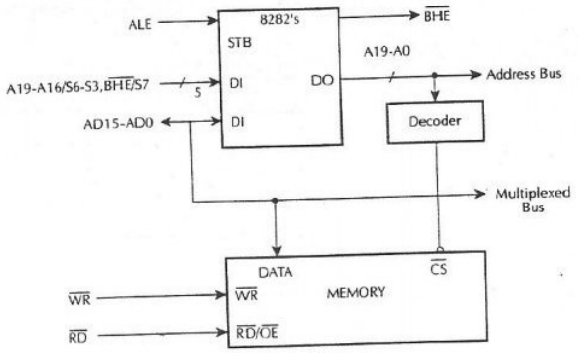


FIGURE 3.14 Devices with output enabling the multiplexed bus.

The 8086 timing guarantees that the read is not valid until after the address is latched by ALE as shown in Figure 3.15.

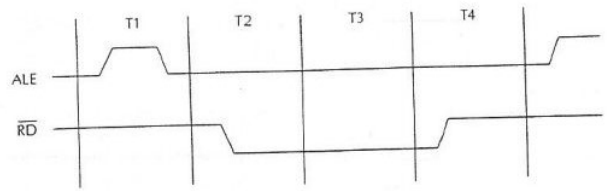


FIGURE 3.15 Relationship of ALE and Read.

All Intel peripherals, EPROMs, and RAMs for microprocessors provide output enable for read inputs to allow connection to the multiplexed bus. Several techniques are available for interfacing the devices without output enables to the 8086 multiplexed bus. However, these techniques will not be discussed here.

3.8.3 INTERFACING WITH MEMORIES

Figure 3.16 shows a general block diagram of an 8086 memory array. In Figure 3.16, the 16-bit word memory is partitioned into high and low

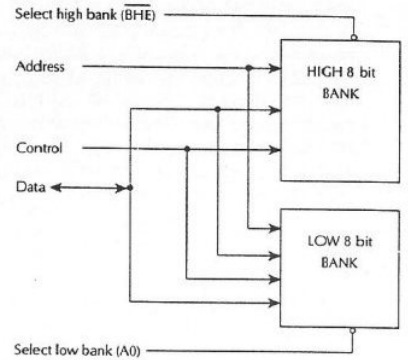


FIGURE 3.16 8086 memory array.

8-bit banks on the upper and lower halves of the data bus selected by $\overline{BHE}$ and A0.

3.8.3.a ROM and EPROM

ROMs and EPROMs are the simplest memory chips to interface to the 8086. Since ROMs and EPROMs are read-only devices, A0 and $\overline{BHE}$ are not required to be part of the chip enable/select decoding (chip enable is similar to chip select except chip enable also provides whether the chip is in active or standby power mode). The 8086 address lines must be connected to the ROM/EPROM chips starting with A1 and higher to all the address lines of the ROM/EPROM chips. The 8086 unused address lines can be used as chip enable/select decoding. To interface the ROMs/RAMs directly to the 8086 multiplexed bus, they must have output enable signals. Figure 3.17 shows the 8086 interfaced to two 2716s.

Byte accesses are obtained by reading the full 16-bit word onto the bus with the 8086 discarding the unwanted byte and accepting the desired byte. If $\overline{RD}$, $\overline{WR}$, and $M/\overline{IO}$ are not decoded to generate separate memory and I/O commands for memory and I/O chips and the I/O space overlaps with the memory space of ROM/EPROM, then $M/\overline{IO}$ must be a condition of chip select decode.

3.8.3.b Static RAMs

Since static RAMs are read/write memories, both A0 and $\overline{BHE}$ must

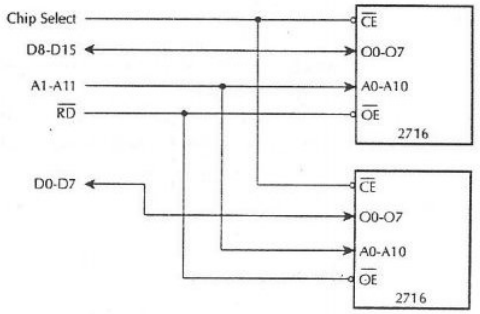


FIGURE 3.17 ROM/EPROM interface to the 8086.

be included in the chip select/chip enable decoding of the devices and write timing must be considered in the compatibility analysis.

For each static RAM, the memory data lines must be connected to either the upper half AD15-AD0 or lower half AD7-AD0 of the 8086 data lines.

For static RAMs without output enable pins, read and write lines must be used as enables for chip select generation to avoid bus contention. If read and write lines are not used to activate the chip selects, static RAMs with common input/output data pins such as 2114 will face extreme bus contentions between chip selects and write active. The 8086 A0 and BHE pins must be used to enable the chip selects. A possible way of generating chip selects for high and low static RAM banks is given in Figure 3.18. Note that Intel 8205 has three enables E1, E2, and E3, three inputs A0-A2, and eight outputs O0-O7.

For devices with output enables such as 2142, one way to generate chip selects for the static RAMs is by gating the 8086 WR signal with BHE and A0 to provide upper and lower bank write strobes. A possible configuration is shown in Figure 3.19. Since the Intel 2142 is a 1024 × 4 bit static RAM, two chips for each bank with a total of 4 chips for 2K × 8 static RAM is required. Note that DATA is read from the 2142 when the output disable OD is low, WE is HIGH, and DATA is written into 2142. When the output disable OD is HIGH, WE is LOW, CS2 is HIGH, and CS1 is LOW. If multiple chip selects are available with the static RAM, BHE and A0 may be used directly as the chip selects. A possible configuration for 2K × 8 array is shown in Figure 3.20.

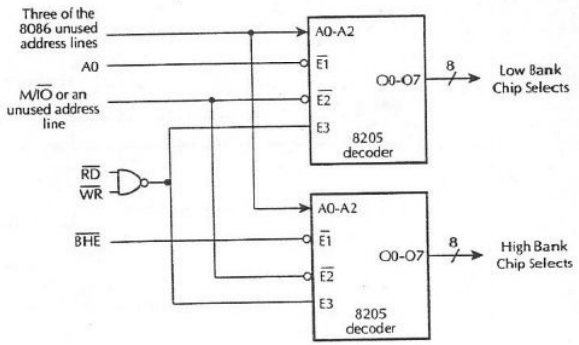


FIGURE 3.18 Generating chip selects for static RAMs without output enables.

3.8.3.c Dynamic RAM

Dynamic RAMs store information as charges in capacitors. Since capacitors can hold charges for a few milliseconds, refresh circuitry is necessary in dynamic RAMs for retaining these charges. Therefore, dynamic RAMs are complex devices to design a system. To relieve the designer of most of these complicated interfacing tasks, Intel provides the 8202 dynamic RAM controller as part of the 8086 family of peripheral devices. The 8202 can be interfaced with the 8086 to build a dynamic memory system. A thorough discussion on this topic can be found in Intel manuals.

3.8.4 8086 PROGRAMMED I/O

The 8086 can be interfaced to 8- and 16-bit I/O devices using either standard or memory-mapped I/O. The standard I/O uses the instructions IN and OUT and is capable of providing 64K bytes of I/O ports. Using standard I/O, the 8086 can transfer 8- or 16-bit data to or from a peripheral device. The 64K byte I/O locations can then be configured as 64K 8-bit ports or 32K 16-bit ports. All I/O transfer between the 8086 and the peripheral devices take place via AL for 8-bit ports (AH is not involved) and AX for 16-bit ports. The I/O port addressing can be done either directly or indirectly as follows:

DIRECT • IN AL, PORTA or IN AX, PORTB inputs 8-bit contents of port A into AL or 16-bit contents of port

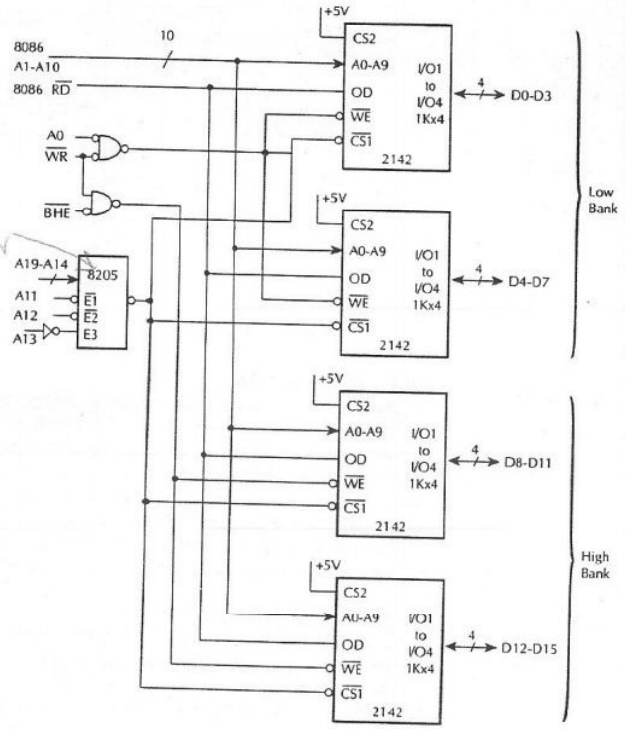


FIGURE 3.19 2K × 8 static RAM array for the 8086.

B into AX, respectively. Port A and port B are assumed as 8- and 16-bit ports, respectively.

- OUT PORTA, AL or OUT PORT B, AX outputs 8-bit contents of AL into port A or 16-bit contents of AX into port B, respectively.

INDIRECT

- IN AX, DX or IN AL, DX inputs 16-bit data addressed by DX into AX or 8-bit data addressed by DX into AL, respectively.
- OUT DX, AX or OUT DX, AL outputs 16-bit contents of AX into the port addressed by DX or 8-bit contents of AL into the port addressed by DX, respectively. In indirect addressing, register DX is used to hold the port address.

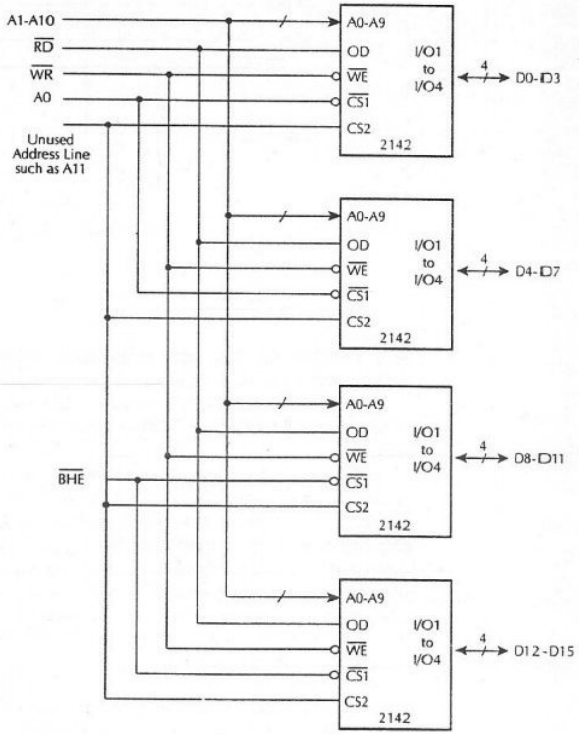


FIGURE 3.20 2K × 8 static RAM array with A0 and $\overline{BHE}$ as direct chip select inputs.

Data transfer using the memory-mapped I/O is accomplished by using memory-oriented instructions such as MOV reg 8 or reg 16, [BX] and MOV [BX], reg 8 or reg 16 for inputting and outputting 8- or 16-bit data from or to an 8-bit register or a 16-bit register addressed by the 20-bit memory-mapped port location computed from DS and BX.

Note that the indirect I/O transfer method is desirable for service routines that handle more than one device such as multiple printers by allowing the desired device (a specific printer) to be passed to the procedure as a parameter.

3.8.4.a Eight-Bit I/O Ports

Devices with 8-bit I/O ports can be connected to either the upper or lower half of the data bus. Bus loading is distributed by connecting an equal number of devices to the upper and lower halves of the data bus. If the I/O port chip is connected to the 8086 lower half of the data lines (AD0-AD7), the port addresses will be even (A0 = 0). On the other hand, the port addresses will be odd (A0 = 1) if the I/O port chip is connected to the upper half of the 8086 data lines (AD8-AD15). A0 will always be one or zero for a partitioned I/O chip. Therefore, A0 cannot be used as an address input to select registers within a particular I/O chip. If two chips are connected to the lower and upper halves of the 8086 address bus that differ only in A0 (consecutive odd and even addresses), A0 and $\overline{BHE}$ must be used as conditions of chip select decode to avoid a write to one I/O chip from erroneously performing a write to the other. Figure 3.21 shows two ways of generating chip selects for I/O port chip.

The first method shown in Figure 3.21A uses separate 8205s to generate chip selects for odd- and even-addressed byte peripherals. If a 16-bit word transfer is performed to an even-addressed I/O chip, the adjacent odd-addressed I/O chip is also selected. Figure 3.21B generates chip selects for byte transfers only.

3.8.4.b Sixteen-Bit I/O Ports

For efficient bus utilization and simplicity of I/O chip selection sixteen-bit I/O ports should be assigned even addresses. Both A0 and $\overline{BHE}$ should be the chip select conditions to ensure that the I/O chip is selected only for word operations. Figure 3.22 shows a method for generating chip for 16-bit ports. Note that in Figure 3.22, the 8086 will output low on both A0 and $\overline{BHE}$ when it executes a 16-bit I/O instruction with an even port address such as IN AX, 0006H. This instruction inputs the 16 bit contents of ports 0006H and 0007H in AX.

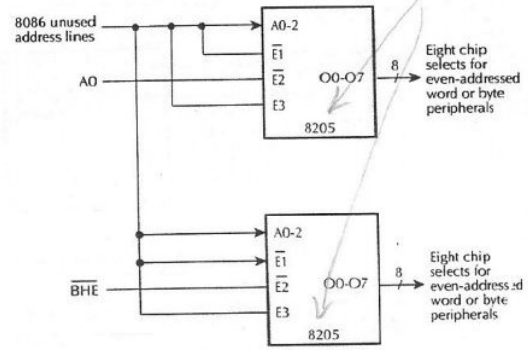


FIGURE 3.21A Techniques for generating I/O device chip selects.

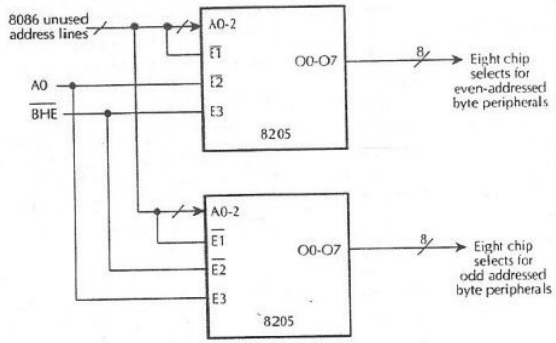


FIGURE 3.21B Techniques for generating I/O device chip selects.

3.8.5 8086-BASED MICROCOMPUTER

In this section, an 8086 will be interfaced in the minimum mode to provide 2K × 16 EPROM, 1K × 16 static RAM, and six 8-bit I/O ports. 2716 EPROM, 2142 static RAM, and 8255 I/O chips will be used for this purpose. Memory and I/O maps will also be determined. Figure 3.23 shows a hardware schematic for accomplishing this.