

MICROPROCESSOR ENGINEERING

Year: Third

Theory: 2H/W

1. Introduction to the microprocessor and computer. (2 Hrs)

- a- A Historical Background.
- b- The Microprocessor-Based Personal Computer System.
- c- High Level and Low Level Languages.

2. The 8086 Hardware Specifications. (8 Hrs)

- a- Internal Architecture.
- b- Pin-outs and the pin functions.
- c- Clock Generator (8284A).
- d- Bus timing.
- e- Ready and the wait state.
- f- Minimum and Maximum mode, 8288 Bus controller.

3. Addressing Modes. (2 Hrs)

Register, immediate, direct, register indirect, based-plus-index, register relative, and base relative-plus-index addressing.

4. Instruction Set and Programming. (18 Hrs)

- a- An Instruction Set.
 - Data Movement Instructions.
 - Arithmetic and Logical Instructions.
 - Program Control Instruction.
- b- Programming the Microprocessor.
 - Using Debugger
 - Using Assembler.

5. Memory Interface. (10 Hrs)

- a- Memory Devices.
ROM, EEPROM, SRAM, DRAM.
- b- Address Decoding.
- c- Memory System Design
- d- Memory Interfacing.

6. Input/Output. (14 Hrs)

Bus buffering and latching, Demultiplexing the busses, The buffered System

I/O Instructions, Isolated and Memory-Mapped I/O, I/O Map, Handshaking,

I/O Port Address Decoding, 8 and 16-Bit I/O Port, The PPI(8255)

Key Matrix Interface. The 8279 Programmable Keyboard/Display Interface 8254 Programmable Interval Timer.
ADC and DAC.

7. Interrupts.

(4 Hrs)

Basic Interrupt Processing, Hardware Interrupts, Expanding the Interrupt

Structure, 8259 PIC, Interrupt examples.

8. Direct Memory Access.

(2 Hrs)

1. Introduction to the microprocessor and computer

1-1 A HISTORICAL BACKGROUND

The mechanical computer age began with the advent of the **abacus** in 500 B.C by Babylonians. The abacus, which was used extensively and is still in use today, was not improved until 1642 when **Blaise Pascal** invented a calculator that was constructed of gears and wheels.

In 1801 **Joseph Jacquard** use punched card as input to Jacquard's loom (computer).

In 1823 **Charles Babbage** create **Analytical Engine** this engine was mechanical computer that stored 1000 20-digit decimal numbers and variable program. Input to this engine was through punched card.

The Electrical Age:

The 1800s saw advent of electrical motor (by Michael Faraday), all based on the mechanical calculator developed by **Blaise Pascal**.

In 1889 **Herman Hollerith** developed the punched card for storing data. He borrowed the idea of punched card from Jacquard.

In 1896 Hollerith formed a company called the Tabulating Machine Company. After a number of mergers, the Tabulating Machine Company was formed into International Business Machines Corporation (**IBM**).

In 1941 **Zuse** construct the first electronic calculating machine. His **Z3** calculating computer was used in aircraft and missile design during war II.

The first **fixed-program electronic computer** system was placed into operation in 1943 to break secret German military codes. This system was invented by Alan Tyring. Tyring called his machine **Colossus**.

The **ENIAC** (Electronics Numerical Integrator And Calculator) is the first **general purpose, programmable electronic computer** system was developed in 1946 at the University of Pennsylvania. The ENIAC has the following specifications:

- Containing over 17000 vacuum tubes.
- Containing over 500 miles of wires.
- Weighed over 30 tans.
- Perform about 100000 operation per second.
- Programmed by rewiring its circuit.

In 1948 the transistor was developed at Bell Labs, followed by the 1958 invention of the Integrated Circuit (IC) by Jack Kilby of Texas Instruments.

The Microprocessor Age:

The fist microprocessor was developed at Intel Corporation in 1971. Table-1 lists the early and modern Intel microprocessor.

Type Number	Speed	Addressing memory	word	Number of instructions
4004	0.2M Hz	4K nibble	4	45
8008	0.2M Hz	16K byte	8	48
8080	2M Hz	64k byte	8	246
8085	3M Hz	64k byte	8	246
8086	5M Hz	1M byte	16	Over 20000

8088	5M Hz	1M byte	8	Over 20000
80186	6-20MHz	1M byte	16	Over 20000
80188	6-20MHz	1M byte	8	Over 20000
80286	8M Hz	16M byte	16	Over 20000
80386DX	16 M Hz	4G byte	32	Over 20000
80386EX	16 M Hz	64M byte	16	Over 20000
80386SL	16 M Hz	32M byte	16	Over 20000
80386SLC	25 M Hz	32M+1Kcache	16	Over 20000
80386SX	25 M Hz	16M byte	16	Over 20000
80486DX	66 M Hz	4G+8Kcache	32	Over 20000
80486SX	50 M Hz	4G+8Kcache	32	Over 20000
80486DX4	100M Hz	4G+16Kcache	32	Over 20000
Pentium	100M Hz	4G+8Kcache	64	Over 20000
Pentium Overdrive	120M Hz	4G+8Kcache	32	Over 20000
Pentium pro	180M Hz	64G+8K L1 cache +256K L2 cache	64	Over 20000
Pentium II	233MHz- 450MHz	64G+32K L1 cache +512K L2 cache	64	Over 20000
Pentium II Xeon	400M Hz	64G+32K L1 cache +512K or 1M L2 cache	64	Over 20000
Pentium III Pentium 4	1G Hz 1.3G Hz	64G+32K L1 cache +256K L2 cache	64	Over 20000

Table 1-1 Many Intel microprocessors

Programming Advanced:

- The first programmable electronic computer system was programmed by **rewiring** its circuit.
- Machine language: The first language was constructed of ones and zeros.
- Assembly language: This language was used in 1950 with computer system UNIVAC. The **Assembler** converts mnemonic codes to binary number (e.g. ADD→ 01000111).
- In 1957 Grace Hopper developed first high level language called FLOW-MATIC. In the same year, IBM developed FORTRAN (FORmula TRANslator) for its computer system.
- Some other high level languages:

❖ ALGOL, COBOL, BASIC, C/C++, PASCAL, ADA, VISUAL BASIC.

1-2 Some terms and concepts for digital computer:

- Bit: a **B**inary **d**igit.
- Byte: a group of eight bits.
- Nibble: a group of four bits.
- ROM: **R**ead **O**nly **M**emory.
- Bus: a group of lines that carry the same type of information.
- RAM: **R**andom **A**ccess **M**emory. Or
R/WM: **R**ead/**W**rite **M**emory.
- Mnemonic: a combination of letters to suggest the operation of an instruction.
- Program: a set of instructions written in a specific sequence for the computer to accomplish a given task.
- Machine language.
- Assembly language.
- Low-Level- Language.
- High-Level-Language.
- Compiler.
- Interpreter.
- Assembler.

1-3 THE MICROPROCESSOR-BASED PERSONAL COMPUTER SYSTEM

Figure 1-1 shows the block diagram of the personal computer. The block diagram is composed of four parts:

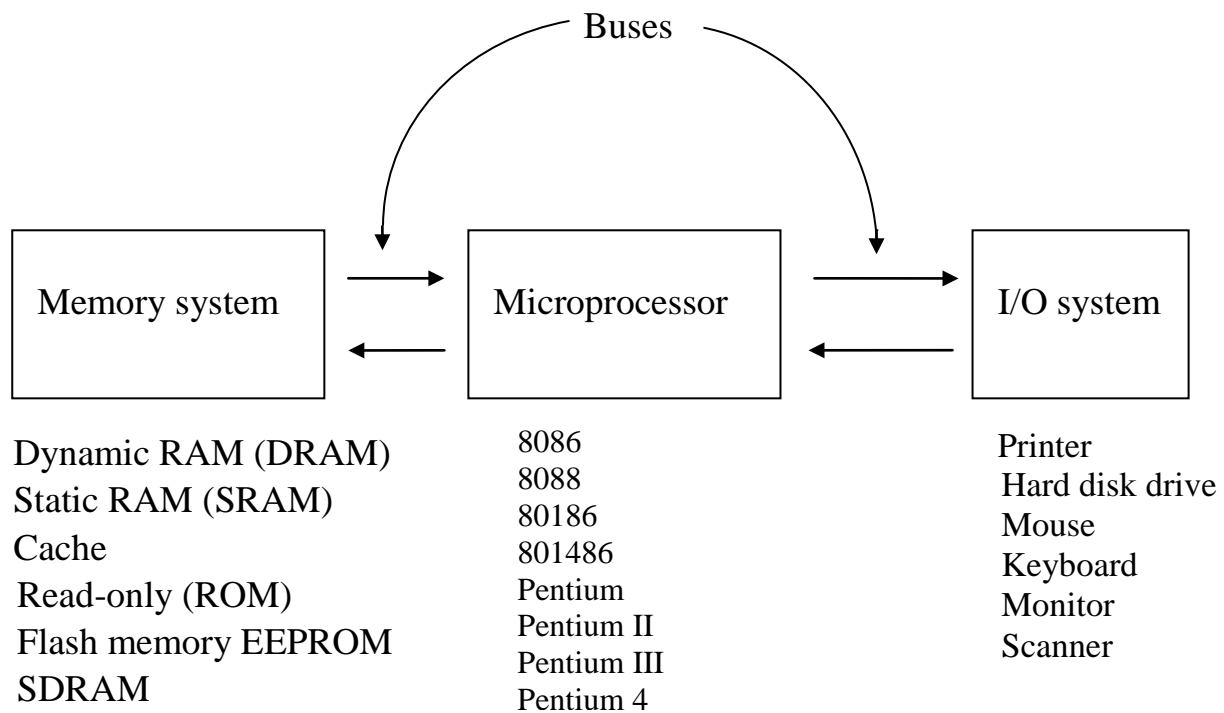


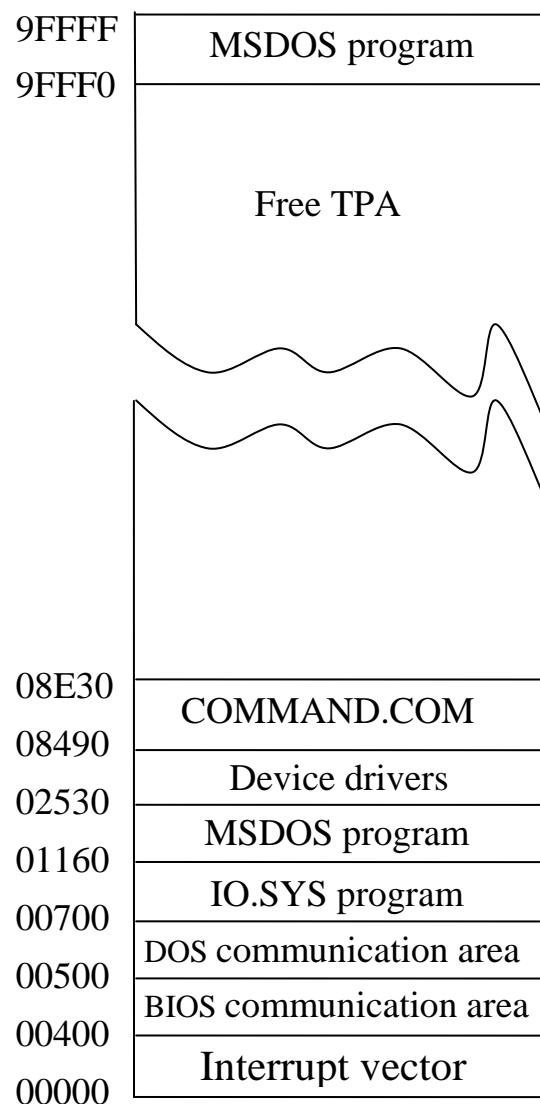
Fig 1-1 The block diagram of a microprocessor-based computer system

i- The memory:

The memory structures of all Intel 80X86-Pentium 4 personal computer systems are similar. This includes the first personal computers based upon the 8088 introduced in 1981 by IBM to the most powerful high-speed versions of today based on the Pentium 4. Figure 1-2 illustrates the memory map of a personal computer system.

The memory system is divided into three main parts: TPA (Transient Program Area), system area, and XMS (Extended Memory System). The type of microprocessor in your computer determines whether an extended memory system exists. If the computer is based upon an older 8086 or 8088 (a PC or XT), the TPA and system areas exist, but there is no extended memory area. The PC and XT contain 640K bytes of TPA and 384K bytes of system memory, for a total memory size of 1M bytes. We often call the first 1M bytes of memory the real or conventional memory system because each Intel microprocessor is designed to function in this area by using its real mode of operation.

Computer systems based on the 80286 through the Pentium 4 not only



Fig(1-3) The memory map of the TPA area of a PC

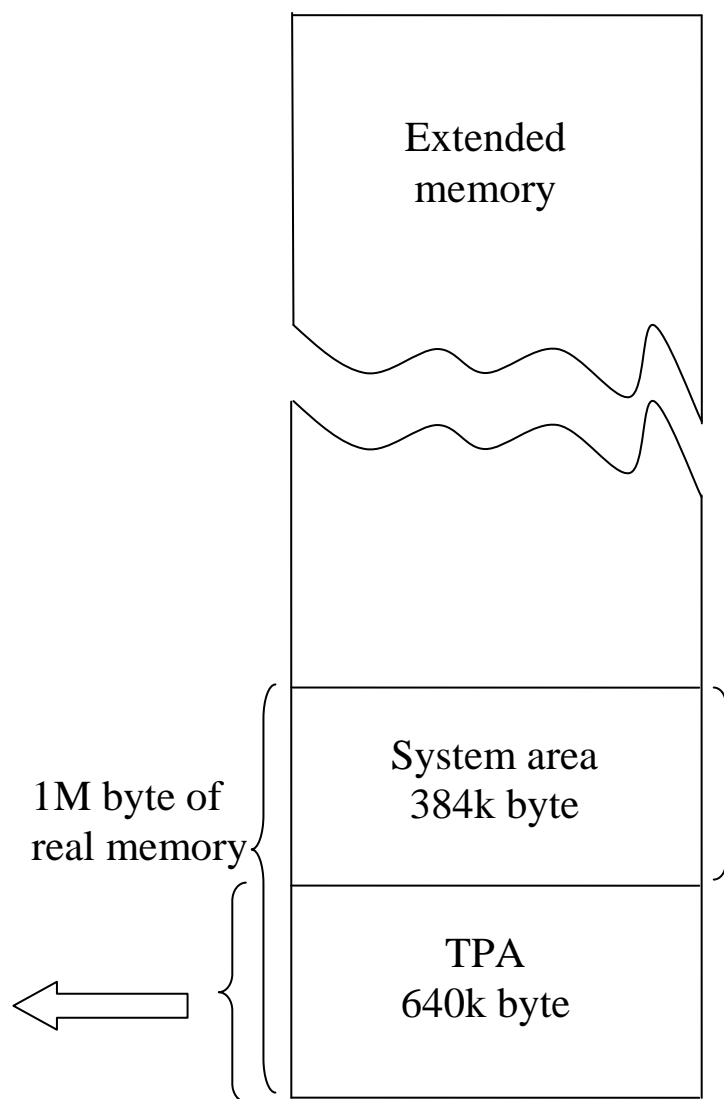


Fig (1-2) The memory map of the PC

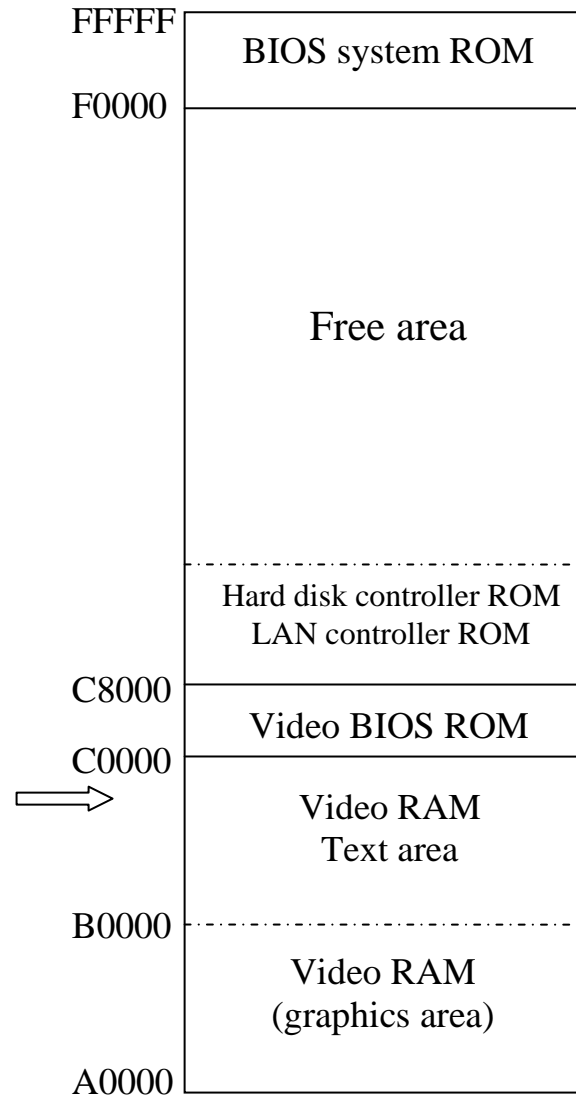


Fig (1-4) The system area of a typical PC

contain the TPA (640K bytes) and system area (384K bytes), they also contain extended memory. These machines are often called AT class machines.

The TPA: The transient program area (TPA) holds the DOS operating system and other programs that control the computer system. The TPA also stores DOS application programs. The length of the TPA is 640K bytes. Figure 1-3 shows the memory map of the TPA.

The System Area: The system area contains program on rather a ROM or flash memory, and areas of RAM for data storage. Figure 1-4 shows the memory map of the system area.

Note: The video card in some computer uses memory locations E1800000H-E2FFFFFF in Windows for its video memory aperture.

ii- **I/O devices:** The I/O devices allow the microprocessor to communicate between itself and the outside world. The I/O space in a computer system extends from port 0000H to port FFFFH The I/O space allows the computer to access up to 64K different 8-bit I/O devices. Figure 1-5 shows the I/O map found in many personal computer systems.

The I/O area contains two major sections. The area below I/O location 0400H is considered reserved for system devices. The remaining area is available I/O space for expansion on newer devices.

iii- **The Microprocessor:** The microprocessor, sometimes referred to as the **CPU** (**C**entral **P**rocessing **U**nit), is the controlling element in a computer.

The microprocessor performs three main tasks: 1- data transfer between itself and the memory or I/O system, 2- simple arithmetic and logic operations, and 3-program flow via simple decisions.

iv- **Buses:** A bus is a number of wires organized to provide a

means of communication among different elements in a microcomputer system. Fig (1-6) shows the buses of 8086 microprocessor, these buses are:

- **Address bus:** the address bus is a group of 20-bit (A0-A19). The address bus is **unidirectional**.
- **Data bus:** the data bus is a group of 16 lines. These lines are **bidirectional**.
- **Control bus:** It contains lines that select the memory or I/O and cause them to perform a read or write operation.

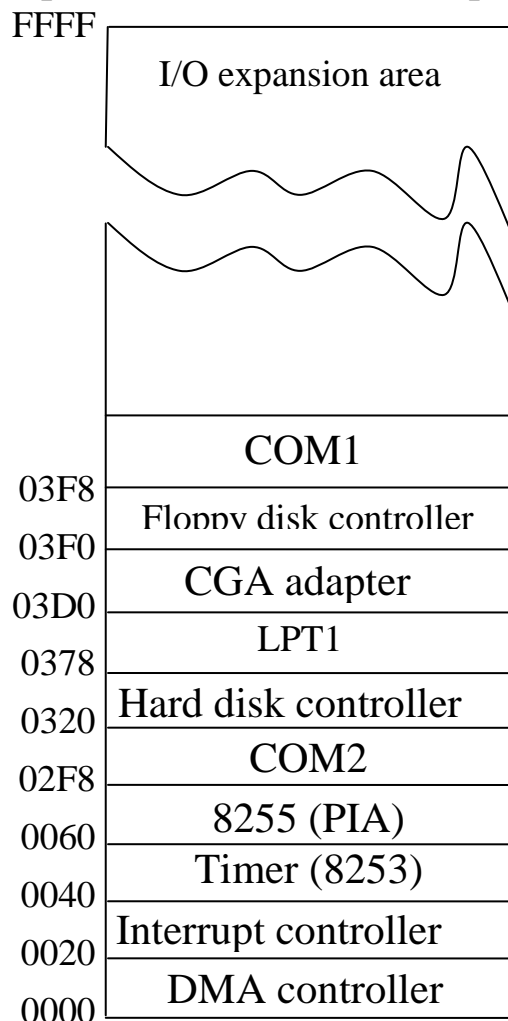


Fig (1-5) The I/O map of a personal computer

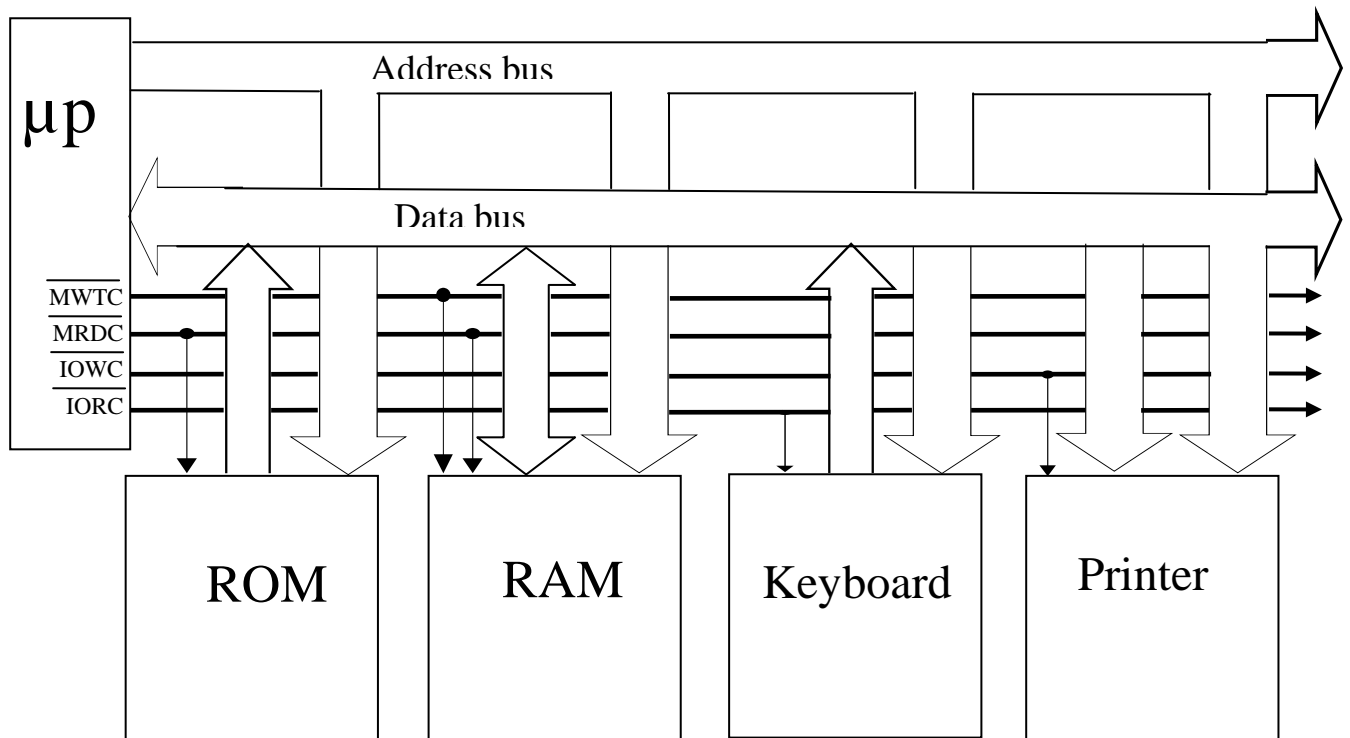


Fig1-6 The block diagram of computer system showing the buses structure

Software Architecture of the 8086 Microprocessor

2-1 MICROARCHITECTURE OF THE 8086 MICROPROCESSOR:

The microarchitecture of a processor is its internal architecture-that is, the circuit building blocks that implement the software and hardware architectures of the 8086 microprocessors.

The microarchitecture of the 8086 microprocessors employs *parallel processing*-that is, they are implemented with several simultaneously operating processing units. Figure 2-1(a) illustrates the internal architecture of the 8086 microprocessors. They contain two processing units: the **B**us **I**nterface **U**nit (**BID**) and the **E**xecution **U**nit (**EU**). Each unit has dedicated functions and both operate at the same time. In essence, this parallel processing effectively makes the fetch and execution of instructions independent operations. This results in efficient use of the system bus and higher performance for 8086 microcomputer systems.

The BID: It is the 8086's connection to the outside world. By interface, we mean the path by which it connects to external devices. The BID is responsible for performing all external bus operations, such as instruction fetching, reading and writing of data operands for memory, and inputting or outputting data for input/output peripherals. These information transfers take place over the system bus. The BID is not only responsible for performing bus operations; it also performs other functions related to instruction and data obtained. For instance, it is responsible for instruction queuing and address generation.

As shown in Figure 2-1(b) the BID contains the segment registers, the instruction pointer, the address generation adder, bus control logic, and an instruction queue.

The BID uses a mechanism known as an *instruction queue* to implement a pipelined architecture. This queue permits the 8086 to

prefetch up to 6 bytes of instruction code. The BID is free to read the next instruction code when:

- The queue is not full-that is, it has room for at least 2 more bytes.
- The execution unit is not asking it to read or write data from memory.

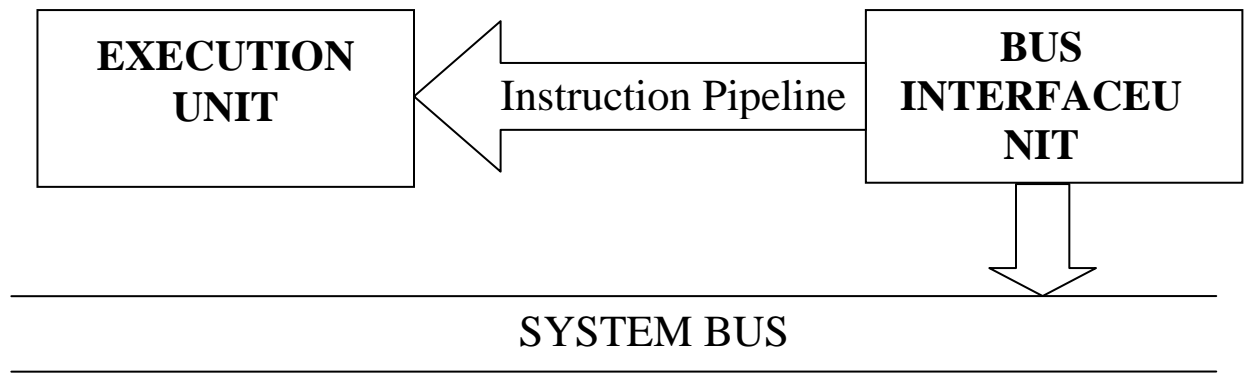
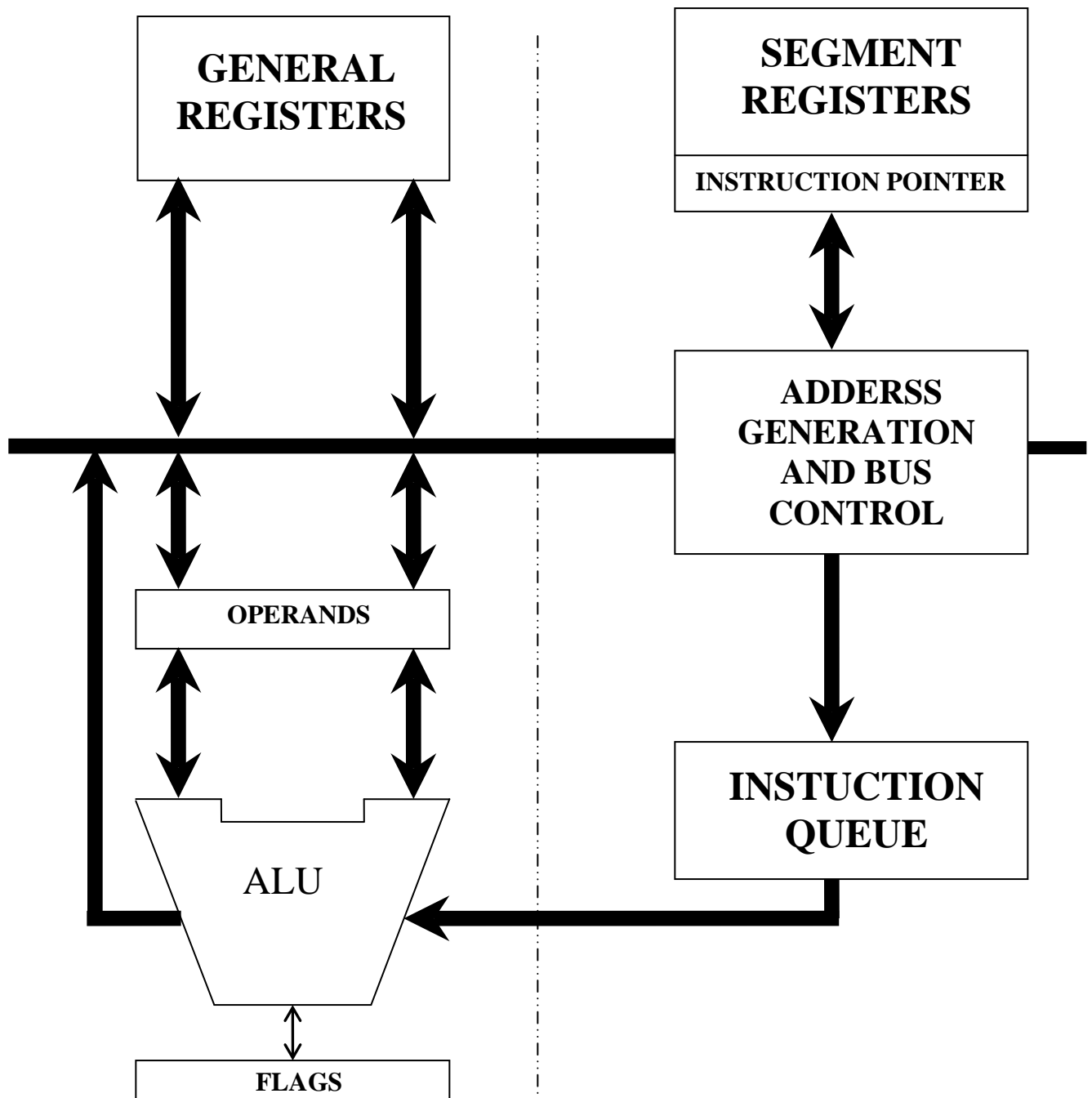


Figure 2-a



(b)

Fig 2-1 (a) Pipelined architecture of 8086 microprocessor

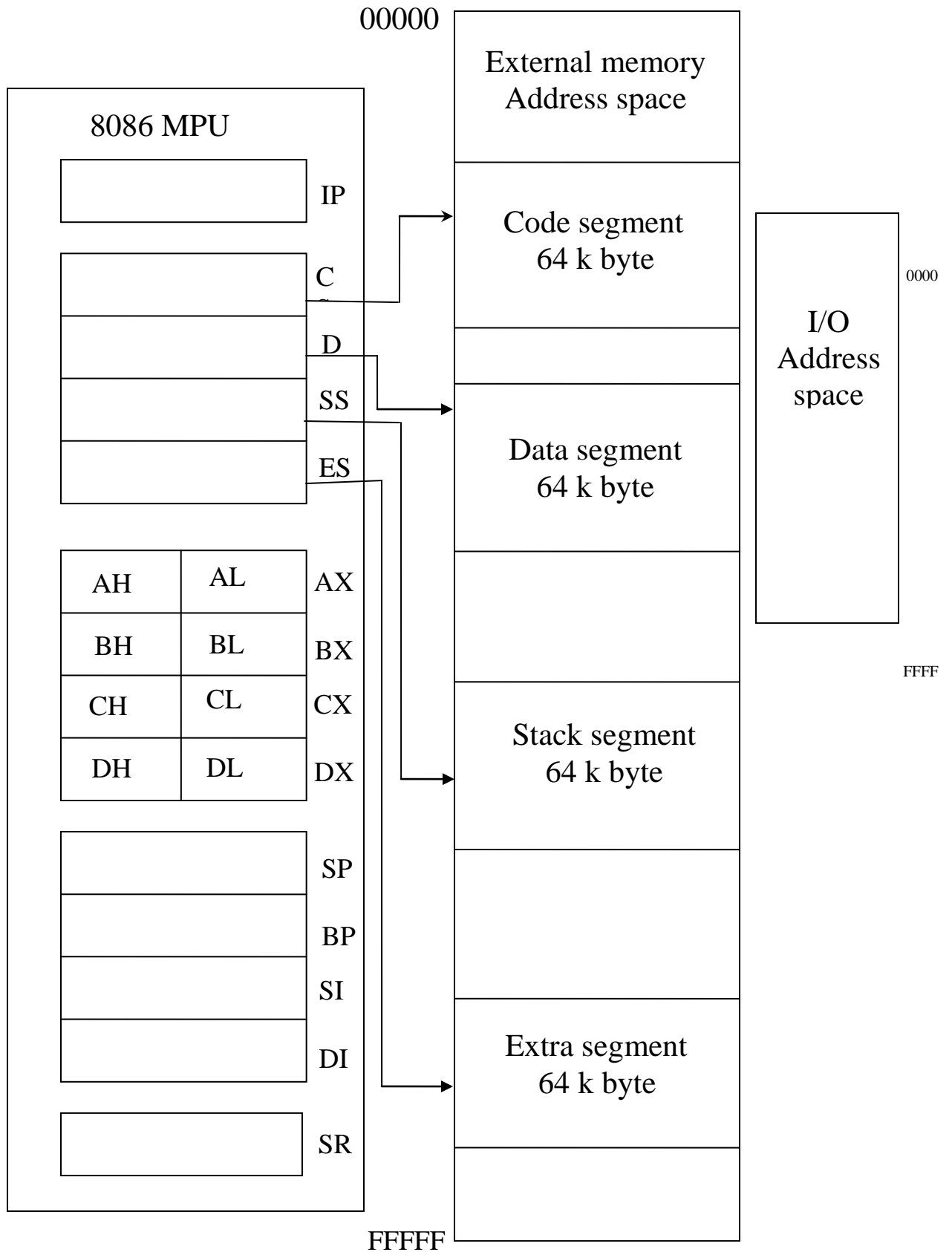


Figure 2-2 Software model of 8086 microprocessor

Prefetched instructions are held in the first-in first-out (FIFO) queue. Since instructions are normally waiting in the queue, the time needed to fetch many instructions of the microcomputer's program is eliminated. If the queue is full and the EU is not requesting access to data in memory, the BIU does not need to perform any bus operations. These intervals of no bus activity, which occur between bus operations, are known as **idle states**.

The EU: The execution unit is responsible for decoding and executing instructions. Notice in Fig. 2-1 (b) that it consists of the *arithmetic logic unit* (ALU), status and control flags, general-purpose registers, and temporary-operand registers. The EU accesses instructions from the output end of the instruction queue and data from the general-purpose registers or memory. It reads one instruction byte after the other from the output of the queue, decodes them, generates data addresses if necessary, passes them to the BIU and requests it to perform the read or write operations to memory or I/O, and performs the operation specified by the instruction. The ALU performs the arithmetic, logic, and shift operations required by an instruction. During execution of the instruction, the EU may test the status and control flags, and updates these flags based on the results of executing the instruction. If the queue is empty, the EU waits for the next instruction byte to be fetched and shifted to the top of the queue.

2.2 SOFTWARE MODEL OF THE 8086 MICROPROCESSOR:

To be able to program a microprocessor, one does not need to know all of its hardware architectural features. What is important to the programmer is to know the various registers within the device and to understand their purpose, functions, operating capabilities, and limitations.

Fig. 2-2 illustrates the software architecture of the 8086 microprocessor. From this diagram, we see that it includes fourteen 16-bit internal registers: the *instruction pointer* (IP), four *data registers* (AX, BX, CX, and DX), two *pointer registers* (BP and SP), two *index registers* (SI and DI), four *segment registers* (CS, DS, SS, and ES) and *status register* (SR), with nine of its bits implemented as status and control flags.

- **INSTRUCTION POINTER (IP):** IP is 16 bits in length and identifies the next word of instruction code to be fetched from the current code segment however; it contains the offset of the instruction code instead of its actual address. This is because IP and CS are both 16 in length, but a 20-bit address is needed to access memory. The value of the next address for the next code access is often denoted as CS: IP.

$$\text{Actual address} = \text{CS} + \text{IP}.$$

- **DATA REGISTER:** Fig 2-3 shows the 8086 has four **general-purpose data registers**. Notice that they are referred to as the accumulator register (A), the base register (B), the count register (C), and the data register (D). Any of the general-purpose data registers can be used as the source or destination of an operand during an arithmetic operation such as ADD or a logic operation such as AND.

AX		Register	Operation	
AH	AL		X	Word multiply, word divide, word I/O
BX		Base	L	Byte multiply, byte divide, byte I/O, translate, decimal arithmetic.
BH	BL		H	Byte multiply, byte divide.
CX		Count	X	Translate
CH	CL		X	String operation, loops.
DX			L	Variable shift and rotate.
DH	DL	Data	X	Word multiply, word divide, indirect I/O.

(a) (b)
Figure 2-3(a) General-purpose data registers. (b) Dedicated register functions

- **POINTER AND INDEX REGISTERS:** Figure 2-4 shows these register. These registers are four general-purpose registers: two pointer registers and two index registers. They store what are called **offset addresses**. An **offset address** selects any location within 64 k byte memory segment.

SP	Stack pointer
BP	Base pointer
SI	Source index
DI	Destination index

Figure 2-4 pointer and index registers.

- **SEGMENT REGISTERS AND MEMORY SEGMENTATION:** The 8086 microprocessor operate in the Real mode memory addressing. **Real mode operation** allows the microprocessor to address only the first 1M byte of memory space-even if it is the Pentium 4 microprocessor. Note that the first 1M byte of memory is called either the **real memory** or **conventional memory** system. Even though the 8086 has a 1M byte address space, not all this memory is active at one time. Actually, the 1M bytes of memory are partitioned into 64K byte (65,536) **segments**. The 8086-80286 microprocessors allow four memory segments a. Figure 2-5 shows these memory segments. Note that a memory segment can touch or even overlap if 64K bytes of memory are not required for a segment. Think of segments as windows that can be moved over any area of memory to access data or code. Also note that a program can have more than four segments, but can only access four segments at a time.

In the real mode a combinational of a segment address and offset address access a memory location. All real mode memory address must consist of a segment address plus an offset address. The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules define the segment register and offset register combination (see Table 2-1). For example, the code segment register is always used with the instruction pointer to address the next instruction in a program. This combination is CS:IP. The code segment register defines the start of the code segment and the instruction pointer locates the next instruction within the code segment. This combination (CS:IP) locates the next instruction executed by the microprocessor. Figure 2-6 show an example that if CS = 1400H and IP = 1200H, the microprocessor fetches its next instruction from memory location:

**Physical address=Segment base address*10+Offset (Effective)
address**

$$\begin{aligned} \text{PA} &= \text{SBA} * 10 + \text{EA} \\ &= 1400\text{H} * 10 + 1200\text{H} = 15200\text{H}. \end{aligned}$$

Table 2-2 shows more examples of memory addresses.

Segment register	Offset address	Physical address
002A	0023	002C3
4900	0A23	49A23
1820	FE00	28000

Table 2-2

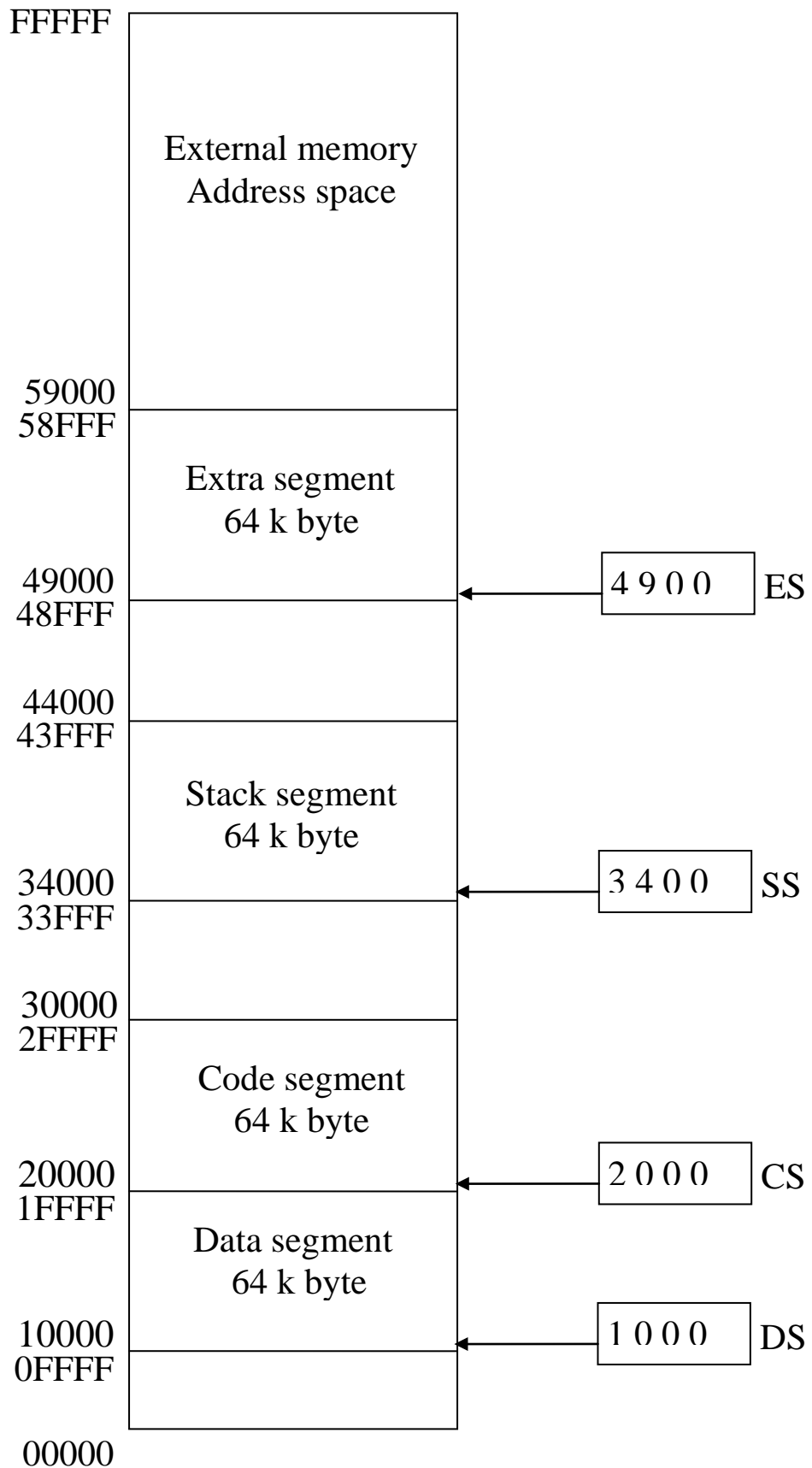


Figure 2-5 A memory system showing the placement of four memory segments.

segment	Offset	Special Purpose
CS	IP	Instruction address
SS	BP	Stack address
SS	SP	Top of the stack
DS	BX,DI,SI, an 8-bit number, or a 16-bit number	Data address
ES	DI for string instructions	String destination address

TABLE 2-1 8086 default 16 bit segment and offset address combinations

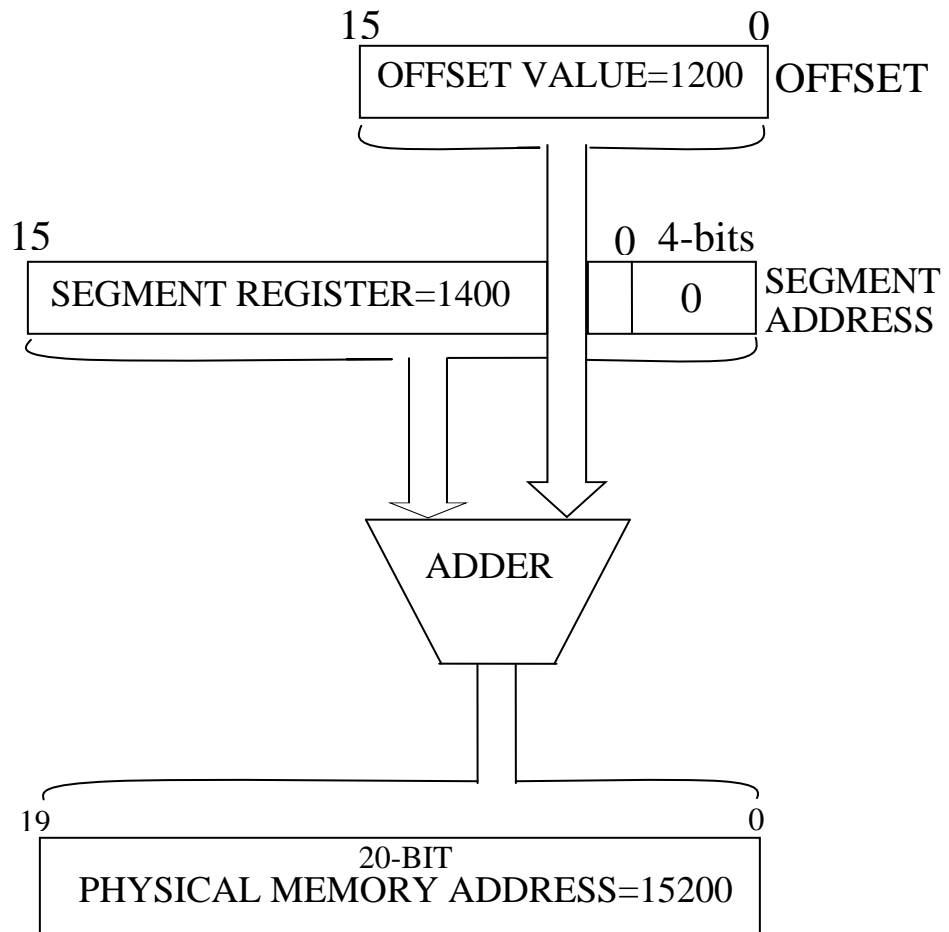


Figure 2-6 generating a physical address

FLAG (STATUS) REGISTER: This register is another 16-bit register within the 8086. Figure (2-7) shows the organization of this register. Notice that just nine of its bits are implemented. Six of these bits represent status flags:

- The carry flag (CF): CF is set if there is a carry-out or a borrow-in for the most significant bit of the result during the execution of an instruction. Otherwise, CF is reset.
- The parity flag (PF): PF is set if the result produced by the instruction has even parity-that is, if it contains an even

number of bits at the 1 logic level. If parity is odd, PF is reset.

- The auxiliary carry flag (AF): AF is set if there is a carry-out from the low nibble into the high nibble or a borrow-in from the high nibble into the low nibble of the lower byte in a 16-bit word. Otherwise, AF is reset.
- The zero flag (ZF): ZF is set if the result produced by an instruction is zero. Otherwise, ZF is reset.
- The sign flag (SF): The MSB of the result is copied into SF. Thus, SF is set if the result is a negative number or reset if it is positive.
- The overflow flag (OF): When OF is set, it indicates that the signed result is out of range. If the result is not out of range, OF remains reset.

The other three flags are control flags. These three flags provide control functions of the 8086 as follows:

- The trap flag (TF): If TF is set, the 8086 goes into the single-step mode of operation.
- The interrupt flag (IF): The IF controls the operation of the INTR (interrupt request) input pin. If IF=1, the INTR pin is enabled; if IF=0, the INTR pin is disabled. The state of IF bit is controlled by the STI (**set IF**) and CLI (**clear IF**) instructions.
- The direction flag (DF): The direction flag selects either the increment or decrement mode for the DI and/or SI registers during string instructions. If D=1, the registers are automatically decremented; if D=0, these registers are automatically incremented. The DF is set with STD (**set direction**) and cleared with CLD (**clear direction**) instructions.

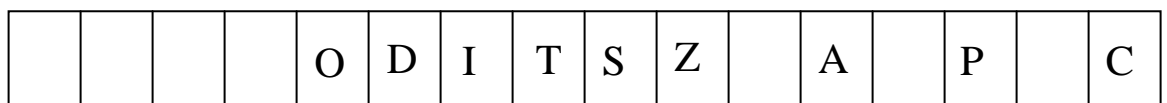


Figure 2-7 Status and control flags.

-ADDRESSING MODES in 8086

In this section we use the MOV instruction to describe the data-addressing modes. Figure 3-1 shows the MOV instruction.

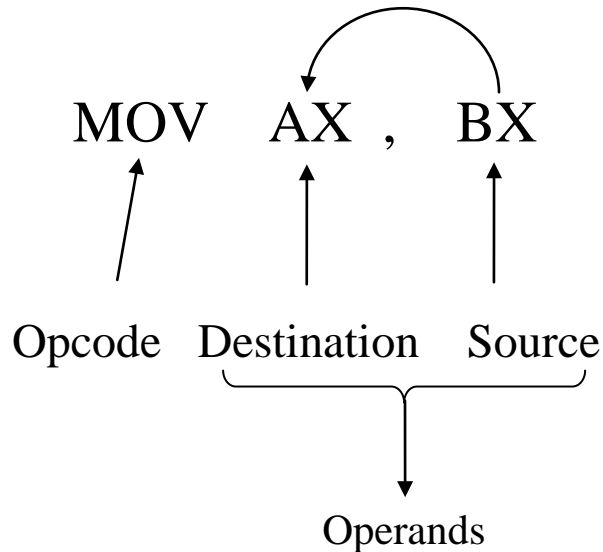


Fig 3-1 The MOV instruction

An addressing mode is a method of specifying an operand.

- **Register Addressing:** With the register addressing, the operand to be accessed is specified as residing in an internal register of 8086. All registers of 8086 can be used except Flag register. Table 3-1 shows many variations of register move instructions. A segment-to-segment register MOV instruction is the only type of MOV instruction not allowed. Note that the code segment register is not normally changed by a MOV instruction because the address of the next instruction is found in both IP and CS. If only CS were changed, the address of the next instruction would be unpredictable. Therefore, changing the CS register with a MOV instruction is not allowed.

- **Immediate Addressing:** The term immediate implies that data immediately follow the hexadecimal opcode in the memory. Table 3-2 shows many variations of immediate move instructions.

Assembly Language	size	Operation
MOV AL, BL	8-bits	Copies BL into AL
MOV CH, CL	8-bits	Copies CL into CH
MOV AX, CX	16-	Copies CX into AX
MOV SP, BP	bits	Copies BP into SP
MOV DS, AX	16-	Copies AX into DS
MOV SI, DI	bits	Copies DI into SI
MOV BX, ES	16-	Copies ES into BX
MOV CX, BX	bits	Copies BX into CX
MOV SP, DX	16-	Copies DX into SP
MOV ES, DS	bits	Not allowed (segment-to-
MOV BL, DX	16-	segment)
MOV CS, AX	bits	Not allowed (mixed sizes)
	16-	Not allowed (the code segment
	bits	register may not be the
	16-	destination register
	bits	

--	--	--

TABLE 3-1 Examples of the register-addressed instructions.

Assembly Language	size	Operation
MOV BL,44	8-bits	Copies a 44 decimal into BL
MOV AX,44H	16-bits	Copies a 0044H into AX
MOV SI,0	16-bits	Copies a 0000H into SI
MOV CH,100	8-bits	Copies a 100 decimal into CH
MOV AL,'A'	8-bits	Copies an ASCII A into AL
MOV AX,'AB'	16-bits	Copies an ASCII BA into AX
MOVCL,11001110B	8-bits	

		Copies a 11001110 binary into CL
--	--	----------------------------------

TABLE 3-2 Examples of the immediate-addressed instructions.

Memory Operand Addressing Modes: To reference an operand in memory, the 8086 must calculate the physical address (PA) of the operand and then initiate a read or write operation of this storage location. The 8086 MPU is provided with a group of addressing modes known as the memory operand addressing modes for this purpose. Physical address can be computed from a segment base address (SBA) and an effective address (EA). SBA identifies the starting location of the segment in memory, and EA represents the offset of the operand from the beginning of this segment of memory.

PA=SBA (segment): EA (offset)


PA=segment base: base + index + Displacement

$$\begin{array}{c}
 \text{PA} = \left\{ \begin{array}{c} \text{CS} \\ \text{DS} \\ \text{SS} \\ \text{ES} \end{array} \right\} : \left\{ \begin{array}{c} \text{BX} \\ \text{BP} \end{array} \right\} + \left\{ \begin{array}{c} \text{SI} \\ \text{DI} \end{array} \right\} + \left\{ \begin{array}{c} \text{8-bit displacement} \\ \text{16-bit displacement} \end{array} \right\} \\
 \downarrow \qquad \qquad \downarrow \qquad \qquad \downarrow \\
 \text{EA} = \text{base} + \text{index} + \text{Displacement}
 \end{array}$$

Not all these elements are always used in the effective address calculation. In fact, a number of memory addressing modes are defined by using various combinations of these elements. Next, we

will examine each of the memory operand addressing modes in detail.

- **Direct Addressing Mode:** Direct addressing mode is similar to immediate addressing in that information is encoded directly into the instruction. However, in this case, the instruction opcode is followed by an effective address, instead of the data. As shown below:

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \text{Direct address} \right\}$$


Default is DS

Direct addressing with a MOV instruction transfer data between a memory location, located within the data segment, and the **AL** or **AX** register. A MOV instruction using this type of addressing is usually a 3-byte instruction.

Example 3-1 The instruction:

MOV AL,DS:[2000 H] or MOV AL,[2000 H]

is **three** bytes instruction which move the contents of the memory location with offset 2000 in the current data segment into internal register AL.

The symbol [] mean contents of the memory.

- **Displacement Addressing Mode:** It is similar to direct addressing mode except that the instruction is four bytes wide instead of three, and the instruction not use AL or AX register.


Example 3-2 The instruction:

MOV CL,DS:[2000 H] or MOV CL,[2000 H]

is **four** bytes instruction which move the contents of the memory location with offset 2000 in the current data segment into internal register CL.

- **Register Indirect Addressing Mode:** This mode is similar to the direct address except that the effective address held in any of the following register: BP, BX, SI, and DI. As shown below:

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} BX \\ BP \\ SI \\ DI \end{array} \right\}$$


 Default is DS

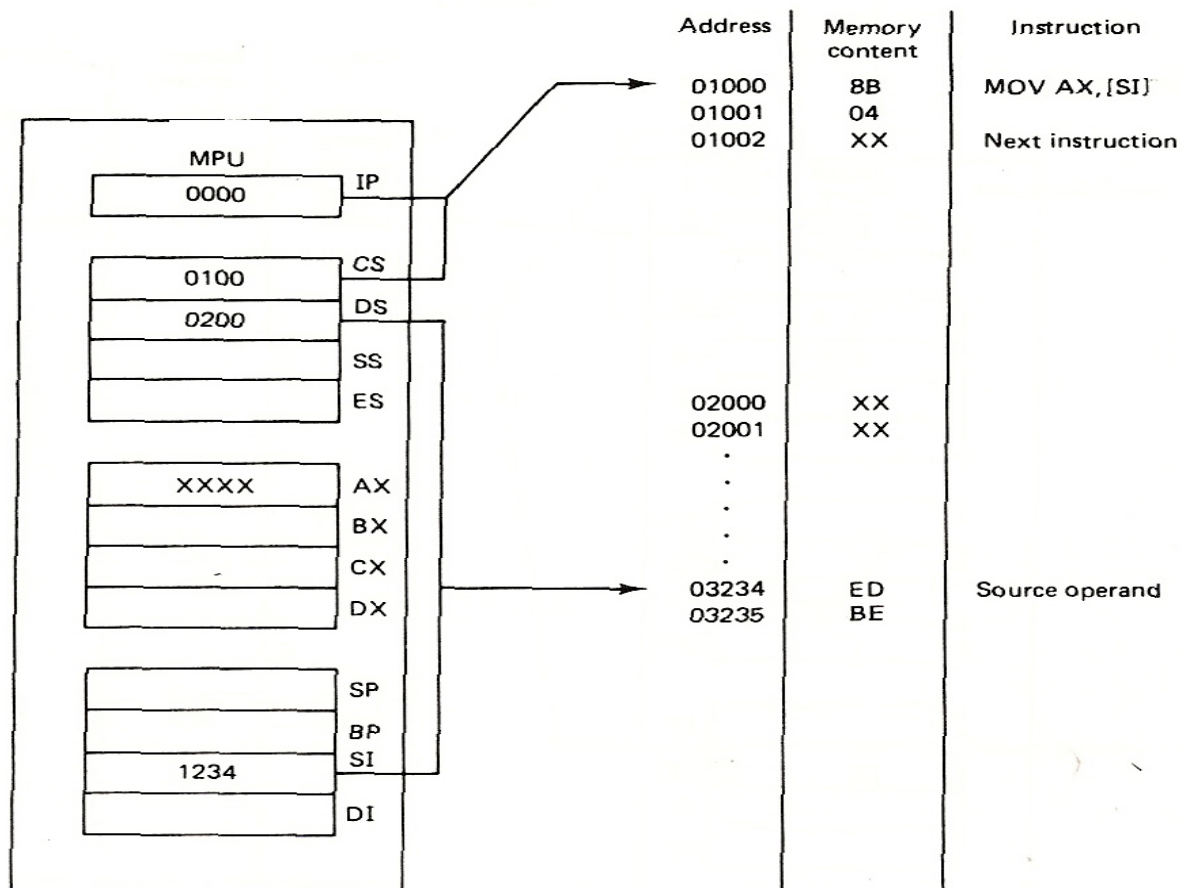
Example 3-3: The instruction:

MOV AX, DS: [SI] or MOV AX, [SI]

move the contents of the memory location that is offset from the beginning of the current data segment by the value of EA in register SI into internal register AX. For instance, Figure 3-2 show that if SI contains 1234_{16} and DS contains 0200_{16} , the result produced by executing the instruction is that the contents of the memory location at address:

$$PA = 02000_{16} + 1234_{16} = 03234_{16}$$

are moved to the AX register.



(a)

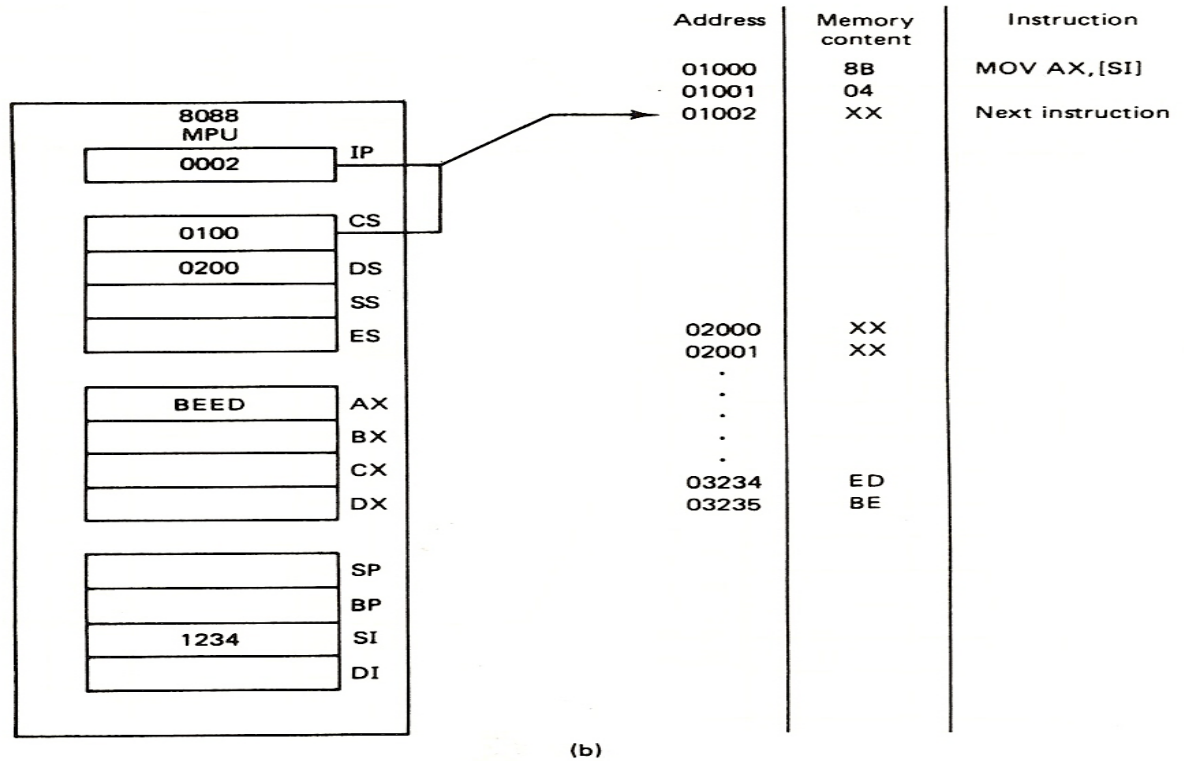


Figure 3-2 (a) Instruction using register indirect addressing mode before fetch and execution. (b) After execution.

- Based Addressing Mode:** In the *based addressing mode*, the effective address of the operand is obtained by adding a direct or indirect displacement to the contents of either base register BX or base pointer register BP. The physical address calculate as shown below:

$$PA = \left\{ \begin{matrix} CS \\ DS \\ SS \\ ES \end{matrix} \right\} : \left\{ \begin{matrix} BX \\ BP \end{matrix} \right\} + \left\{ \begin{matrix} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{matrix} \right\}$$

Example 3-4: If BX=1000, DS=0200, and AL=EDH, for the following instruction:

MOV [BX] + 1234H, AL

EA=BX+1234H

EA=1000H+1234H

EA=2234H

PH=DS*10+EA

PH=0200H*10+2234H

PH=4234H

So it writes the contents of source operand AL (EDH) into the memory location 04234H.

If BP is used instead of BX, the calculation of the physical address is performed using the contents of the stack segment (SS) register instead of DS. This permits access to data in the stack segment of memory.

- **Indexed Addressing Mode:** In the Indexed addressing mode, the effective address of the operand is obtained by adding a direct or indirect displacement to the contents of either SI or DI register. The physical address calculate as shown below:

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} SI \\ DI \end{array} \right\} + \left\{ \begin{array}{c} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{array} \right\}$$

- **Based-Indexed**

Addressing Mode: Combining the based addressing mode and the indexed addressing mode results in a new, more powerful mode known as based-indexed addressing mode. This addressing mode can be used to access complex data structures such as two-dimensional arrays. As shown below this mode can be used to access elements in an m X n array of data. Notice that

the displacement, which is a fixed value, locates the array in memory. The base register specifies the m coordinate of the array, and the index register identifies the n coordinate. Simply changing the values in the base and index registers permits access to any element in the array.

$$PA = \left\{ \begin{array}{c} CS \\ DS \\ SS \\ ES \end{array} \right\} : \left\{ \begin{array}{c} BX \\ BP \end{array} \right\} + \left\{ \begin{array}{c} SI \\ DI \end{array} \right\} + \left\{ \begin{array}{c} 8\text{-bit displacement} \\ 16\text{-bit displacement} \end{array} \right\}$$

MACHINE LANGUAGE CODING

4-1 THE INSTRUCTION SET:

The microprocessor's instruction set defines the basic operations that a programmer can specify to the device to perform. Table 4-1 contains list basic instructions for the 8086. For the purpose of discussion, these instructions are organized into groups of functionally related instructions. In Table 4-1, we see that these groups consist of the data transfer instructions, arithmetic instructions, logic instructions, string manipulation instructions, control transfer instructions, and processor control instructions.

4-2 CONVERTING ASSEMBLY LANGUAGE INSTRUCTIONS TO MACHINE CODE

To convert an assembly language program to machine code, we must convert each assembly language instruction to its equivalent machine code instruction. The machine code instructions of the 8086 vary in the number of bytes used to encode them. Some instructions can be encoded with just 1 byte, others can be done in 2 bytes, and many require even more. The maximum number of bytes of an instruction is 6. Single-byte instructions generally specify a simpler operation with a register or a flag bit.

The machine code for instructions can be obtained by following the formats used in encoding the instructions of the 8086 microprocessor. Most multibyte instructions use the general instruction format shown in Fig. 4-1.

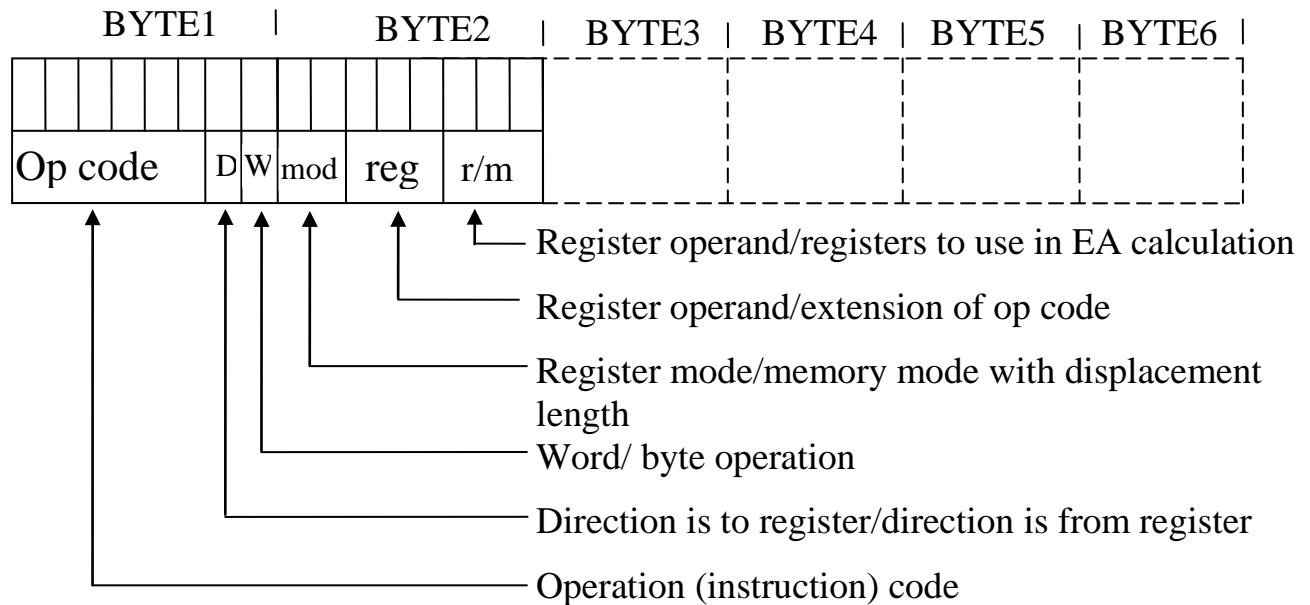


Fig 4-1 General instruction format

Looking at Fig. 4-1, we see that byte 1 contains three kinds of information:

- 1.Opcode field (6-bit): Specifies the operation, such as add, subtract, or move, that is to be performed.
- 2.Register direction bit (D bit): Tells whether the register operand specified by reg in byte 2 is the source or destination operand. Logic 1 in this bit position indicates that the register operand is a destination operand, and logic 0 indicates that it is a source operand.
- 3.Data size bit (W bit): Specifies whether the operation will be performed on 8-bit or 16-bit data. Logic 0 selects 8 bits and 1 selects 16 bits as the data size.

The second byte in Fig. 4-1 has three fields:

The register filed (reg.) is 3-bit. It is used to identify the register for the first operand, which is the one that was defined as the source or destination by the D bit in byte 1. Table-2 shows the encoding for each of the 8086 's registers. Here we find that the 16-bit register AX and the 8-bit register AL are specified by the same binary code. Note that (W) bit in byte 1 determined whether AX or AL is used. .

The 2-bit mod field and 3-bit r/m field together specify the second operand. Encoding for these two fields is shown in tables 4-3(a) and (b), respectively. Mod indicates whether the operand is in a register or memory. Note that in the case of a second operand in a register, the mod field is always 11. The r/m field, along with the W bit from byte1, selects the register.

Example 4-1: Encode the following instruction using the information in figure 4-1, tables 2,3 and the op code for MOV is 100010.

MOV BL, AL

Solution:-

For byte 1:

The six most significant bits of first byte is 100010.

D =0 to specify that a register AL is the source operand.

W=0 to specify a 8-bit data operation.

$$\therefore \text{byte 1} = (10001000)_2 = (88)_{16}$$

For byte 2:

mod=11 (the second operand is also register)

reg=000 (from table 4-2 code of AL=000)

r/m=011 (from table 4-2 code of BL=011)

$$\therefore \text{byte 2} = (11000011)_2 = (C3)_{16}$$

Thus, the hexadecimal machine code for instruction MOV BL,AL=88C3H

Example 4-2: Encode the following instruction using the information in figure 4-1, tables 2,3 and the op code for ADD is 000000.

ADD [BX][DI]+1234H, AX

Solution:-

For byte 1:

The six most significant bits of first byte is 000000.

D =0 to specify that a register AX is the source operand.

W=1 to specify a 16-bit data operation.

$$\therefore \text{Byte1} = (00000001)_2 = (01)_{16}$$

For byte 2:

mod=10 (Memory mode with 16-bit displacement)

reg=000 (from table 4-2 code of AX=000)

r/m=011 (from table 4-2 (b))

$\therefore \text{Byte2} = (10000011)_2 = (81)_{16}$

The displacement 1234_{16} is encoded in the next two bytes, with the Least Significant Byte (LSB) first. Therefore, the machine code is:

ADD [BX][DI]+1234H, AX=01813412

NOTES: The general form of figure (4-1) can not be used to encode all instructions of 8086. We can note the following:

1- In some instructions, one or more additional single bit fields need to be added. Table-4 shows these 1-bit fields and their functions.

2- Instructions that involve a segment register need a 2-bit field to encode which register is to be affected. This field is called seg field. Table-5 shows the encoded code of segment register.

Example 4-3: Encode the following instruction:

MOV [BP][DI]+1234H, DS

Solution:- Table-1 shows that this instruction is encoded as:

10001100 mod 0 seg r/m disp

$\therefore \text{Byte1} = (10001100)_2 = (8C)_{16}$

For byte 2:

mod=10 (Memory mode with 16-bit displacement)

seg=11 (from table-5)

r/m=011 (from table-1 (b))

$\therefore \text{Byte2} = (10011011)_2 = (9B)_{16}$

The machine code of MOV [BP][DI]+1234H, DS=8C9B3421

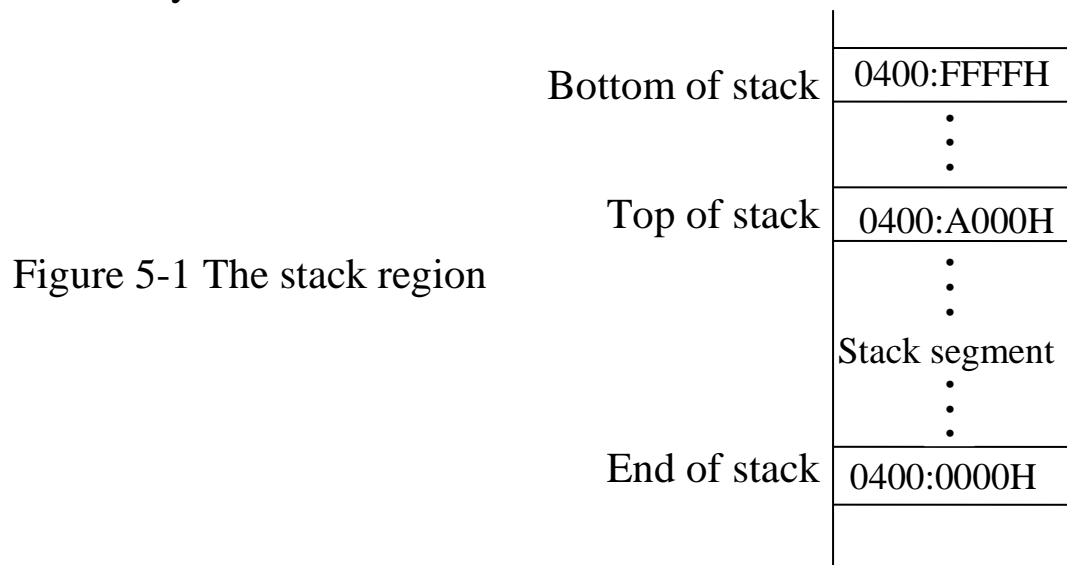
DATA TRANSFER AND STRING MANIPULATION GROUPS

5-1 Data transfer group:

THE STACK: The stack is implemented in the memory of 8086, and it is used for temporary storage.

Starting address of stack memory (top of the stack) obtained from the contents of the stack pointer (SP) and the stack segment (SS) (SS:SP). Figure 5-1 shows the stack region for SS=0400H and SP=A000H. Data transferred to and from the stack are **word-wide**, not byte-wide. Whenever a word of data is pushed onto the top of the stack, the high-order 8 bits are placed in the location addressed by SP-1. The low-order 8 bits are placed in the location addressed by SP-2. The SP is then decremented by 2.

Whenever data are popped from the stack, the low-order 8 bits are removed from the location addressed by SP. The high-order 8 bits are removed from the location addressed by SP+2. The SP is then incremented by 2.



The MOV instruction. The function of MOV instruction is to transfer a byte or word of data from a source location to a destination location. The general form of MOV instruction is as shown below:

Mnemonic	meaning	Format	Operation	Flags
----------	---------	--------	-----------	-------

				affected
MOV	move	MOV D,S	(S)→(D)	None

From table T1-a, we see that data can be moved between general purpose-registers, between a general purpose-register and a segment register, between a general purpose-register or segment register and memory, or between a memory location and the accumulator. Note that memory-to-memory transfers are not allowed.

• **PUSH/POP:** The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (Last In First Out) stack memory. The general forms of PUSH and POP instructions are as shown below:

Mnem.	Meaning	Format	operation	Flags and (S or D)
PUSH	Push word onto stack	PUSH S	((Sp))←(S) (Sp)←(Sp)-2	Register None Seg reg (CS illegal)
POP	Pop word onto stack	POP D	(D)←((SP)) (Sp)←(SP)+2	Memory None Flag register

• **LEA, LDS, and LES (load-effective address) INSTRUCTIONS:**

These instructions load a segment and general purpose registers with an address directly from memory. The general forms of these instructions are as shown below:

Mnem.	Meaning	Format	operation	Flags
LEA	Load effective address	LEA reg16,EA	EA→(reg16)	None
LDS	Load register and DS	LDS reg16,EA	[PA]→(reg16) [PA+2]→(DS)	None
LES	Load register and ES	LES reg16,EA	[PA]→(reg16) [PA+2]→(ES)	None

The LEA instruction is used to load a specified register with a 16-bit effective address (EA).

The LDA instruction is used to load a specified register with the contents of PA and PA+1 memory locations, and load DS with the contents of PA+2 and PA+3 memory locations.

The LES instruction is used to load a specified register with the contents of PA and PA+1 memory locations, and load ES with the contents of PA+2 and PA+3 memory locations.

EXAMPLE 5-1: Assuming that (BX)=20H, DI=1000H, DS=1200H, and the following memory contents:

Memory	12200	12201	12202	12203	12204
Contents	11	AA	EE	FF	22

What result is produced in the destination operand by execution the following instructions?

a- LEA SI, [DI+BX+5] b-LDS SI, [200].

SOLUTION:

a- $EA = 1000 + 20 + 5 = 1025 \rightarrow (SI) = 1025$

b- $PA = DS:EA = DS * 10 + EA = 1200 * 10 + 200 = 12200$

$\therefore (SI) = AA11H$ and $(DS) = FFEH$

MISCELLANEOUS DATA TRANSFER INSTRUCTIONS:

XCHG: The XCHG (exchange) instruction exchanges the contents of a register with the contents of any other register or memory. The general form of this instruction is as shown below:

Destination	source
Accumulator	Reg16
Memory	Register
Register	Register
Register	Memory

Mnem.	Meaning	Format	operation	Flags
XCHG	exchange	XCHG D,S	(D) \leftrightarrow (S)	None

Allowed

operand

XLAT: This instruction used to simplify implementation of the lookup table operation. The general form of this instruction is as shown below:

Mnem.	Meaning	Format	operation	Flags
XTAL	Translate	XTAL	$((AL)+(BX)+(DS)*10) \rightarrow (AL)$	None

LAHF and SAHF: The LAHF and SAHF instructions are seldom used because they were designed as bridge instructions. These instructions allowed 8085 microprocessor software to be translated into 8086 software by a translation program.

IN and OUT: There are two different forms of IN and OUT instructions: the direct I/O instructions and variable I/O instructions. Either of these two types of instructions can be used to transfer a byte or a word of data. All data transfers take place between an I/O device and the MPU's accumulator register. The general form of this instruction is as shown below:

Mnem.	Meaning	Format	operation	Flags
IN	Input direct	IN Acc, port	$(Acc) \leftarrow (port)$	None
	Input variable	IN Acc, DX	$(Acc) \leftarrow ((DX))$	
OUT	output direct	OUT port, Acc	$(Acc) \rightarrow (port)$	None
	Variable	OUT DX, Acc	$(Acc) \rightarrow ((DX))$	

5-2 String manipulation group: By string we mean a series of data words (or bytes) exist sequentially in memory. With string instructions we use SI, DI, and DF (direction flag) as shown below:

DS: SI

ES: DI

DI, SI= DI+ (1 or 2), SI+ (1 or 2) if DF=0

DI, SI= DI- (1 or 2), SI- (1 or 2) if DF=1

The general forms of these instructions are as shown below:

Mnem	Meaning	FORMAT	operation	Flags
MOV S	Move string	MOVSB / MOVSW	$((ES)*10+(DI)) \leftarrow ((DS)*10+(SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None
CMP S	Compare string	CMPSB/ CMPSW	Set flags as per $((DS)*10+(SI)) - ((ES)*10+(DI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	C,P,A,Z,S, O
SCAS	Scan string	SCASB/ SCASW	Set flags as per $(AL \text{ or } AX) - ((ES)*10+(DI))$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	C,P,A,Z,S, O
LODS	Load string	LODSB/ LODSW	$(AL \text{ or } AX) \leftarrow ((DS)*10+(SI))$ $(SI) \leftarrow (SI) \pm 1 \text{ or } 2$	None
STOS	Store string	STOSB/ STOSW	$((ES)*10+(DI)) \leftarrow (AL \text{ or } AX)$ $(DI) \leftarrow (DI) \pm 1 \text{ or } 2$	None

MOVSB,MOVSW: This instruction transfer data from one memory location to another. This is the only memory-to-memory transfers allowed in 8086 MPU.

EXAMPLE5-2: Assuming that (DS) =2000H, (SI)=1000H, (ES) =1200H, (DI) =1A00H, (DF) =0, content of 2000:1000=7AH, and content of 2000:1001=22H. Explain the operation performed by MOVSW instruction.

SOLUTION: MOVSW instruction copy content of memory locations addressed by DS:SI, & DS:SI+1 to memory locations addressed by ES:DI, & ES:DI+1.

$\therefore [DS:SI] \rightarrow [ES:DI] \text{ \& } [DS:SI+1] \rightarrow [ES:DI+1]$

$\therefore [ES:DI] = 7AH \text{ \& } [ES:DI+1] = 22H$

D=0 & move word ~~-(DI)~~=(DI) -2=19FEH & (SI)=(SI)-2=0FFEh

CMPSB/CMPSW: This instruction compares two elements in the same different strings. It subtracts the destination operand from the source operand and adjusts the flags accordingly. The result of subtraction is not saved.

EXAMPLE5-3: Assuming that (DS) =2000H, (SI) =1000H, (ES) =1200H, (DI) = 1A00H, content of 2000:1000=7AH, (DF) =0, and content of 2000:1001=8AH. Explain the operation performed by CMPSB instruction.

SOLUTION: CMPSB instruction content of memory locations addressed by DS: SI with memory location addressed by ES: DI.

$[DS:SI] - [ES:DI] = 7A - 8A = F0$

Flags: C=1, P=1, A=0, Z=0, S=1, O=1

D=0 & compare byte ~~-(DI)~~=(DI) -1=19FF & (SI)=(SI)-1=0FFFh

CSASB/ CSASB W: This instruction is similar to compare string; it compares the destination string with the contents of AL or AX. It subtracts the destination operand from AL or AX. The result of subtraction is not saved.

EXAMPLE5-4: Assuming that (DS) =2000H, (SI) =1000H content of 2000:1000=7AH and content of AL=8AH. Explain the operation performed by SCASB instruction.

SOLUTION: SCASB instruction content of memory locations addressed by DS: SI with the contents of AL.

$[DS:SI] - [ES:DI] = 7A - 8A = F0$

Flags: C=1, P=1, A=0, Z=0, S=1, O=1

D=0 & scan byte (B) \rightarrow (DI)=(DI)-1=19FF & (SI)=(SI)-1=0FFFh

LODSB/LODSW: This instruction loads a byte or word from a string in memory into AL or AX. The address of the source is specified by DS:SI.

STOSB/STOSW: This instruction stores a byte or word from AL or AX into memory location(s). The address of the destination is specified by ES:DI.

REP: In most applications, the basis string operations must be repeated. Inserting a repeat prefix before the instruction that is to be repeated does this. The general form of this instruction is as shown below:

Prefix	Used with	Meaning
REP	MOVS,STOS	Repeat while CX \neq 0
REPE/REPZ	CMPS,SCAS	Repeat while C Y 0 & ZF=1
REPNE/REPNZ	CMPS,SCAS	Repeat while C Y 0 & ZF=0

ARITHMETIC AND LOGICAL GROUPS

6-1 Arithmetic and logical groups: The arithmetic group includes instructions for the addition, subtraction, multiplication, and division operations.

The state that results from the execution of an arithmetic instruction is recorded in the flags register. The flags that are affected by the arithmetic instructions are C, A, S, Z, P, O.

For the purpose of discussion, we divide the arithmetic instructions into four subgroups: addition, subtraction, multiplication, and division.

6-1-1 Addition Instructions: The general forms of these instructions are shown in figure 6-1

Mnem.	Meaning	Format	Operation	Flags affected
ADD	Addition	ADD D,S	$(S)+(D) \rightarrow (D)$ Carry $\rightarrow (CF)$	O,S,Z,A,P,C
ADC	Add with carry	ADC D,C	$(S)+(D)+(CF) \rightarrow (D)$ Carry $\rightarrow (CF)$	O,S,Z,A,P,C
INC	Increment by 1	INC D	$(D)+1 \rightarrow D$	O,S,Z,A,P
AAA	ASCII adjust for addition	AAA		A,C O,S,Z,P undefined
DAA	Decimal adjust for addition	DAA		S,Z,A,P,C O undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(c)

Fig 6-1 (a) addition instructions
(b) Allowed operands for ADD and ADC instructions.

(c) Allowed operands for INC

Destination
Reg16
Reg8
Memory

(b)

The result of executing ADD is expressed as:

$$(S)+(D) \rightarrow (D)$$

That is the contents of the source operand are added to the contents of the destination operand. The carry is stored in the carry flag. This instruction performs the half-add binary arithmetic operation.

The result of executing ADC is expressed as:

$$(S)+(D)+(CF) \rightarrow (D)$$

This instruction performs the operation of full-adder logic function.

Another instruction of 8086 is INC. This instruction adds one to 8-bit register or 16-bit register or memory location.

Another instruction of 8086 is AAA. This instruction is ASCII arithmetic instruction with ASCII-coded numbers. These numbers range in value from 30H to 39H for the numbers 0-9. The AAA instruction should be executed immediately after the ADD instruction that adds ASCII data.

EXAMPLE6-1: What is the result of executing the following instruction sequence?

ADD AL, BL

AAA

Assume that AL contains 32_{16} (The ASCII code for number 2), BL contains 34_{16} (The ASCII code for number 4), and AH=0.

SOLUTION: Executing the ADD instruction give:

$$(AL)=(AL)+(BL)=32_{16}+34_{16}=66_{16}$$

The result after executing AAA is:

$$(AL)=06_{16} \quad \& \quad (AH)=00_{16}$$

AF and CF remain cleared.

Another instruction of 8086 is DAA. This instruction is used to perform an adjust operation similar to that performed by AAA but for the addition of BCD numbers instead of ASCII number.

EXAMPLE6-2: What is the result of executing the following instruction sequence?

```

MOV DX, 1234H
MOV BX, 3099H
MOV AL, BL
ADD AL, DL
DAA
MOV CL, AL
MOV AL, BH
ADC AL, DH
DAA
MOV CH, AL

```

SOLUTION: In this example, a 1234 adds to a 3099. The result is in BCD and stored in CX. CX=4333

6-1-2 Subtraction Instructions:. The general forms of these instructions are shown in figure 6-2

Mnem	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	(D)-(S)→(D) Borrow→ (CF)	O,S,Z,A,P,C
SBB	Subtract with borrow	SBB D,S	D)-(S)-(CF)→(D) Borrow→ (CF)	O,S,Z,A,P,C
DEC	Decrement by 1	DEC D	(D)-1→D	O,S,Z,A,P
NEG	Negate	NEG D	0-(D)→D 1→(CF)	O,S,Z,A,P,C
DAS	Decimal adjust for subtraction	DAS		S,Z,A,P,C O undefined
AAS	ASCII adjust for subtraction	AAS		A,C O,S,Z,P undefined

(a)

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate

Accumulator	Immediate
-------------	-----------

Destination
Reg16
Reg8
Memory

Destination
Register
Memory

(c)

(d)

(b)

Fig 6-2 (a) Subtraction instructions. (b) Allowed operands for SUB and SBB instructions. (c) Allowed operands for DEC instruction. (d) Allowed operands for NEG instruction

The subtraction subgroup is similar to the addition subgroup. It performs subtraction instead of addition.

The (NEG) instruction replaces the value of its operand by its negative. This is done by subtracts the contents of operand from zero and the result is returned to the operand. As an example if (BX)=003A₁₆ and the instruction NEG BX is execute, the result is:

$$(BX)=0000_{16}-(BX)=0000_{16}-003A_{16}=FFC6_{16}$$

6-1-3 Multiplication and Division Instructions: The 8086 has instructions for multiplication and division of binary, BCD numbers, and signed or unsigned integers. Multiplication and division are performed on bytes or on words. Fig 6-3 shows the form of these instructions.

Mnem	Meaning	Format	Operation	Flags affected
MUL	Multiply (unsigned)	MUL S	(AL).(S8) →(AX) (AX).(S16) →(DX),(AX)	O,C S,Z,A,P undefined
DIV	Division (unsigned)	DIV S	1-Q((AX)/(S8))→(AL) R((AX)/(S8))→(AH) 2-Q((DX,AX)/(S16))→(AX) R((DX,AX)/(S16))→(DX) If Q is FF ₁₆ in case 1 or FFFF ₁₆ in case 2, then	O,S,Z,A,P,C undefined

			type 0 interrupt occurs	
IMUL	Multiply (signed)	IMUL S	(AL).(S8) \rightarrow (AX) (AX).(S16) \rightarrow (DX),(AX)	O,C S,Z,A,P undefined
IDIV	Division (signed)	IDIV S	1-Q((AX)/(S8)) \rightarrow (AL) R((AX)/(S8)) \rightarrow (AH) 2- Q((DX,AX)/(S16)) \rightarrow (AX) R((DX,AX)/(S16)) \rightarrow (DX) If Q is positive and exceed 7FFF ₁₆ or if Q is negative and becomes less than 8001 ₁₆ , then type 0 interrupt occurs	O,S,Z,A,P,C undefined
AAM	Adjust AL for division	AAM	Q((AL)/10) \rightarrow (AH) R((AL)/10) \rightarrow (AL)	S,Z,P O,A,C undefined
AAD	Adjust AX for division	AAD	(AH)*10+(AL) \rightarrow (AL) 00 \rightarrow (AH)	S,Z,P O,A,C undefined
CBW	Convert byte to word	CBW	(MSB of AL) \rightarrow (All bits of AH)	None
CWD	Convert byte to double word	CWD	(MSB of AX) \rightarrow (All bits of DX)	None

(a)

Source
Reg8
Reg16
Mem8
Mem16

(b)

Fig 6-3 (a) Multiplication and division instructions. (b) Allowed operand

8-bit division: The result of a DIV or IDIV instruction for an 8-bit divisor is represented by:

$$\begin{array}{ccccccc} (AX) / (8\text{-bit operand}) \rightarrow & (AL) & , & (AH) \\ \uparrow & \uparrow & & \uparrow & \uparrow \\ \text{Dividend} & \text{Division} & & \text{quotient} & \text{remainder} \end{array}$$

16-bit division: The result of a DIV or IDIV instruction for an 16-bit divisor is represented by:

$$\begin{array}{ccccccc} (DX), (AX) / (16\text{-bit operand}) \rightarrow & (DX) & , & (AX) \\ \uparrow & \uparrow & & \uparrow & \uparrow \\ \text{Dividend} & \text{Division} & & \text{quotient} & \text{remainder} \end{array}$$

EXAMPLE 6-3: The 2's-complement signed data contents of AL equal -3 and the contents of CL equal -2. What result is produced in AX by executing the following instruction?

MUL CL

And

IMUL CL

SOLUTION: As binary data, the contents of AL and CL are:

$$(AL) = -3(\text{as } 2\text{'s-complement}) = (FF-3+1) = 11111101 = FD_{16}$$

$$(CL) = -2(\text{as } 2\text{'s-complement}) = (FF-2+1) = 11111110 = FE_{16}$$

Executing the MUL CL instruction give:

$$(AX) = 11111101_2 * 11111110_2 = 1111101100000110_2 = FA06_{16}$$

Executing the IMUL CL instruction give:

$$(AX) = -3_{16} * -2_{16} = 6_{16}$$

AAM & AAD instructions: AAM instruction is adjust instruction for unpacked BCD multiplication. It follows the multiplication instruction.

EXAMPLE 6-4: write a program that multiplies 5_{10} by 6_{10} .

SOLUTION:

```
MOV AL, 5
MOV CL, 5
MUL CL
AAM
HLT
```

AAD instruction is adjust instruction for unpacked BCD division. Unlike all other adjustment instructions AAD instruction appears before a division.

EXAMPLE 6-5: write a program that divided 72_{10} by 9_{10} .

SOLUTION:

```
MOV BL, 9
MOV AX, 0702H
AAD
DIV BL
HLT
```

CBW & CWD instructions: The division instruction can also be used to divide a signed 8-bit dividend in AL by an 8-bit divisor. For this purpose we use (CBW) instruction. When (CBW) instruction is executed the value of AX register is as shown below:

AH=0 if the number is positive.

AH=1 if the number is negative.

AL=8-bit dividend.

EXAMPLE 6-6: What is the result of executing the following sequence of instructions?

```
MOV AL, A1H
CBW
CWD
```

SOLUTION: The first instruction loads AL with $A1_{16}$. This gives:

$$(AL)=A1_{16}=10100001_2$$

Executing the second instruction extends the MSB of AL, 1, into all bits of AH. The result is:

$$(AH)=11111111_2=FF_{16}$$

$$\therefore (AX)=1111111110100001_2=FFA1_{16}$$

Executing the third instruction extends the MSB of AH, 1, into all bits of DX. The result is:

$$(AX)=FFA1_{16}$$

$$(DX)=FFFF_{16}$$

CMP instruction: The CMP (compare) instruction enables us to determine the relationship between two numbers-that is, whether they are equal or unequal, and when they are unequal, which one is larger. Fig 6-4 shows the form of these instructions.

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

Mnem	Meaning	Format	Operation	Flags
CMP	Compare	CMP D,S	(D)-(S)	C,A,O, P,F,Z

(a)

Figure 6-4 (a) Compare instruction.

(b) Allowed operand

(b)

EXAMPLE 6-7: What is the result of executing the following sequence of instructions?

MOV AL, 99H

MOV BL, 1BH

CMP AL, BL

Assume that flags Z, S, C, A, O, and P are all initially reset.

SOLUTION:

The first instruction loads AL with 99_{16} . No status flags are affected.

The second instruction loads AL with $1B_{16}$. No status flags are affected.

The third instruction CMP subtracts BL from AL:

$$\begin{array}{r} 10011001_2 = -103_{10} \\ -00011011_2 = -(+27)_{10} \\ \hline 01111110_2 = +126_{10} \end{array}$$

In this process of subtraction, we get the status flag as shown below:

A=1, C=0, O=1, P=1, S=0, Z=0.

6-2 Logical group:

6-2-1 AND, OR, XOR, and, NOT instructions: These instructions perform their respective logic operations. Fig 6-5 shows the format and the operand for these instructions.

EXAMPLE 6-8: What is the result of executing the following sequence of instructions?

```
MOV AL, 01010101B
AND AL, 00011111B
OR  AL, 11000000B
XOR AL, 00001111B
```

Solution: The result can be summarized as shown below:

Instruction		(AL)		
MOV AL, 01010101B		01010101		
AND AL, 00011111B		00010101		
OR AL, 11000000B		11010101		
XOR AL, 00001111B		11011010		

Mnem	Meaning	Format	Operation	Flags affected
AND	Logical AND	AND D,S	(S).(D)→(D)	O,S,Z,P,C A undefined
OR	Logical OR	OR D,S	(S)+(D)→(D)	O,S,Z,P,C A undefined

XOR	Logical inclusive OR	XOR D,S	$(S) \oplus (D) \rightarrow (D)$	O,S,Z,P,C A undefined
NOT	Logical NOT	NOT D	$(\bar{D}) \rightarrow (D)$	None

(a)

Fig6-5 (a) logical instructions (b) Allowed operands for AND, OR, XOR instructions. (c) Allowed operands for NOT instruction.

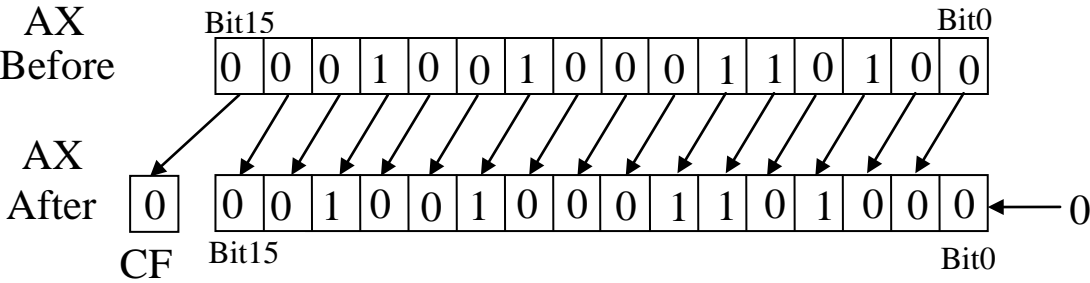
Destination	Source	Destination
Register	Register	Register
Register	Memory	Memory
Memory	Register	
Register	Immediate	
Memory	Immediate	
Accumulator	Immediate	

(c)

(b)

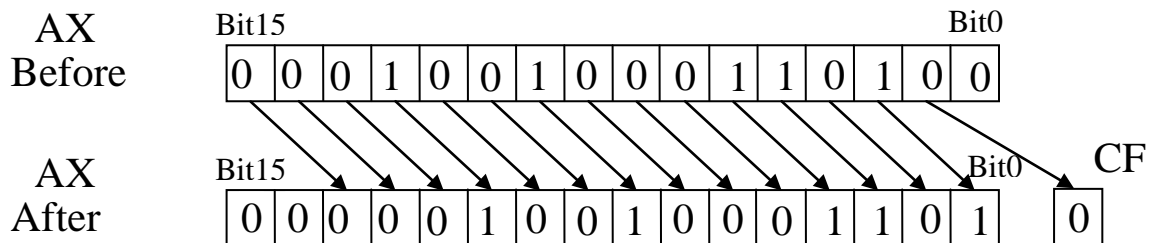
6-2-2 Shift instructions: Shift instructions can perform two basic types of shift operations; the logical shift and the arithmetic shift. Also, each of these operations can be performed to the right or to the left.

SHL, SHR, SAL, and, SAR instructions: The operation of these instructions is described in figure 6-6. As an example if (AX)=1234H, Executing SHL AX,1 can be illustrated as shown below:



The 16-bit contents of the AX register are shifted 1 bit position to the left. The vacated LSB location is filled with zero and the bit shifted out of the MSB is saved in CF.

If the source operand is specified as CL instead of 1, the count in this register represents the number of bit positions the contents of the operand are to be shifted. An example of executing SHR AX, CL FOR (AX=1234H) and CL=2 is shown below:



In an arithmetic shift to the left, the SAL operation, the vacated bits at the right of the operand are filled with zeros, whereas in an arithmetic shift to the right, the SAR operation, the vacated bits at the left are filled with the value of the original MSB of the operand.

Mnem	Meaning	Format	Operation	Flags affected
SAL/ SHL	Shift arithmetic left /shift logical left	SAL D,Count SHL D,Count	Shift the (D) left by the number of bit positions equal to Count and fill the vacated bits positions on the right with zeros	C, P, S, Z A undefined O undefined if count $\neq 1$
SHR	shift logical right	SHR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions on the left with zeros	C, P, S, Z A undefined O undefined if count $\neq 1$
SAR	Shift arithmetic right	SAR D,Count	Shift the (D) right by the number of bit positions equal to Count and fill the vacated bit positions	C, P, S, Z A undefined O undefined if count $\neq 1$

			on the left with the original most significant bit	
--	--	--	--	--

(a)

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

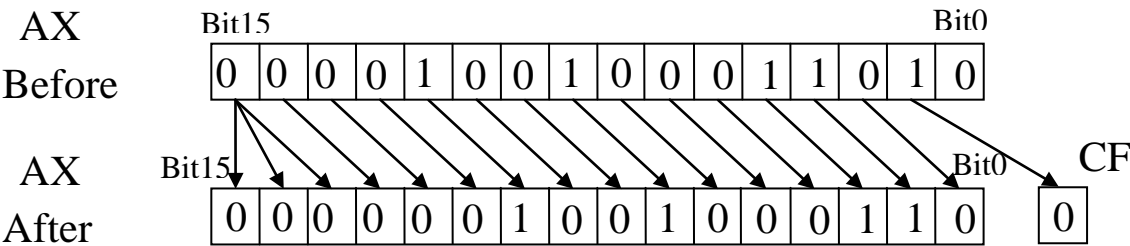
Figure 6-6 (a) Shift instructions.

(b) Allowed operands.

(b)

Example 6-8: If $(CL) = 02_{16}$ and $AX = 091A_{16}$. Determine the new contents of AX and the carry flag after executing the instruction
SAR AX, CL

Solution:



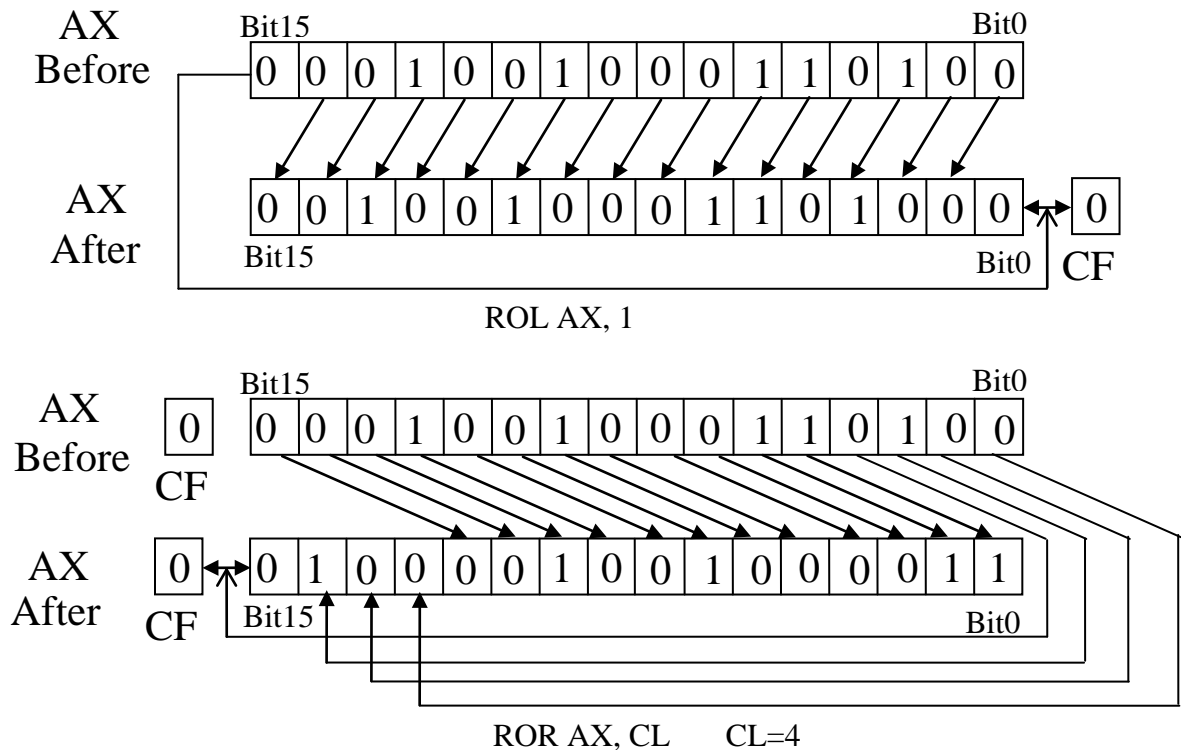
6-2-3 Rotate instructions: Rotate instructions, are similar to the shift instructions.

ROL, ROR, RCL, and, RCR instructions: The operation of these instructions is described in figure 6-7. They have the ability to rotate the contents of either an internal register or a storage location in memory. Also, the rotation that takes place can be from 1 to 255 bit positions to the left or to the right. Moreover, in the case of a multibit rotate, the number of bit positions to be rotated is specified by the value in CL.

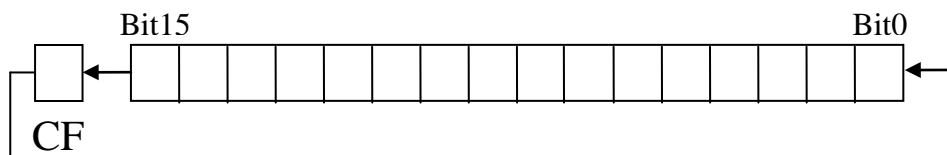
Example 6-9: If $(CL) = 02_{16}$ and $AX = 1234A_{16}$. Determine the new contents of AX and the carry flag after executing the instructions:

- ROL AX, 1
- ROR AX, CL

Solution:



The other two rotate instructions, RCL and RCR, differ from ROL and ROR in that the bits are rotated through the carry flag. The following figure illustrates the rotation that takes place due to execution of the RCL instruction.



Mnem	Meaning	Format	Operation	Flags affected
ROL	Rotate left	ROL D,Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the leftmost bit goes back into the rightmost bit position.	C O undefined if count $\neq 1$
ROR	Rotate right	ROR D,Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from	C O undefined if count $\neq 1$

			the rightmost bit goes back into the leftmost bit position.	
RCL	Rotate left through carry	RCL D,Count	Same as ROL except carry is attached to (D) for rotation.	C O undefined if count $\neq 1$
RCR	Rotate right through carry	RCR D,Count	Same as ROR except carry is attached to (D) for rotation.	C O undefined if count $\neq 1$

(a)

Figure 6-7 (a) Rotate instructions.
(b) Allowed operands.

Destination	Count
Register	1
Register	CL
Memory	1
Memory	CL

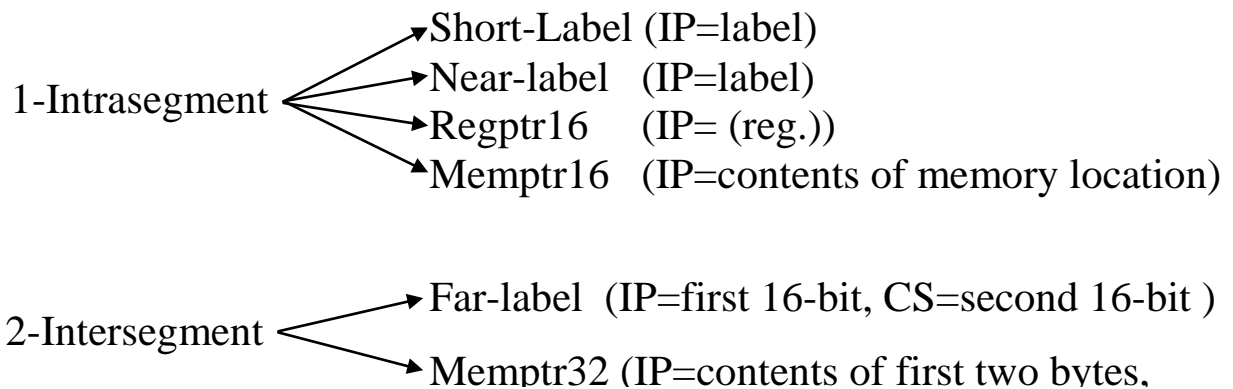
6-2-4 Test instruction: Test instruction, is similar to the AND instruction. The difference is that the AND instruction change the destination operand, while the TEST instruction does not. A TEST only affects the condition of the flag register, which indicates the result of the test. The TEST instruction uses the same addressing modes as AND instruction.

CONTROL TRANSFER AND PROCESSOR CONTROL GROUPS

7-1 Control Transfer Group

7-1-1 Jump instructions: 8086 allowed two types of jump operation. They are the unconditional jump and the conditional jump.

• **Unconditional jump:** Figure 7-1 shows the unconditional jump instruction. There are two basic kinds of unconditional jump. These kinds are as shown below:



				Operation
				Short-label
				Near-label
				Far-label
				Memptr16
				Regptr16
				Memptr32
Mnem.	Meaning	Format	Operation	Flags
JMP	Unconditional jump	JMP operand	Jump to the address specified by the operand.	None

(a)

Figure 7-1 (a) unconditional jump instruction.

(b)

Allowed

operand.

(b)

EXAMPLE 7-1: Assume the following state of 8086:

(CS)=1075H, (IP)=0300H, (SI)=A00H, (DS)=400H,
 (DS:A00)=10H, (DS:A01)=B3H, (DS:A02)=22H,
 (DS:A03)=1AH.

To what address is program control passed if each of the following
 JMP instruction is execute?

(a) JMP 10. (b) JMB 1000H. (c) JMP [SI]. (d) JMP SI. (e) JMP
 FAR [SI]. (f) JMP 3000:1000.

SOLUTION:

(a) 1075:10 (b) 1075:1000 (c)
 1075:B310
 (d)1075:A00 (e) 1A22: B310 (f)
 3000:1000

Mnem	Meaning	Format	Operation	Flags
JMP	Conditional jump	Jcc operand	If the specified condition cc is true the Jump to the address specified by the operand is initiated; otherwise the next instruction is executed.	None

(a)

Mnem	Meaning	Condition
JA	Above	CF= 0 and ZF=0
JAE	Above or equal	CF=0
JB	Below	CF=1
JBE	Below or equal	CF= 1 and ZF=1
JC	Carry	CF=1
JCXZ	CX register is zero	(CF or ZF)=0
JE	Equal	ZF=1
JG	Greater	ZF=0 and SF=OF
JGE	Greater or equal	SF=OF
JL	Less	(SF xor OF)=1
JLE	Less or equal	((SF xor OF)) or ZF=1
JNA	Not above	CF=1 or ZF=1
JNAE	Not above nor equal	CF=1
JNB	Not below	CF=0
JNBE	Not below nor equal	CF=0 and ZF=0
JNC	Not carry	CF=0

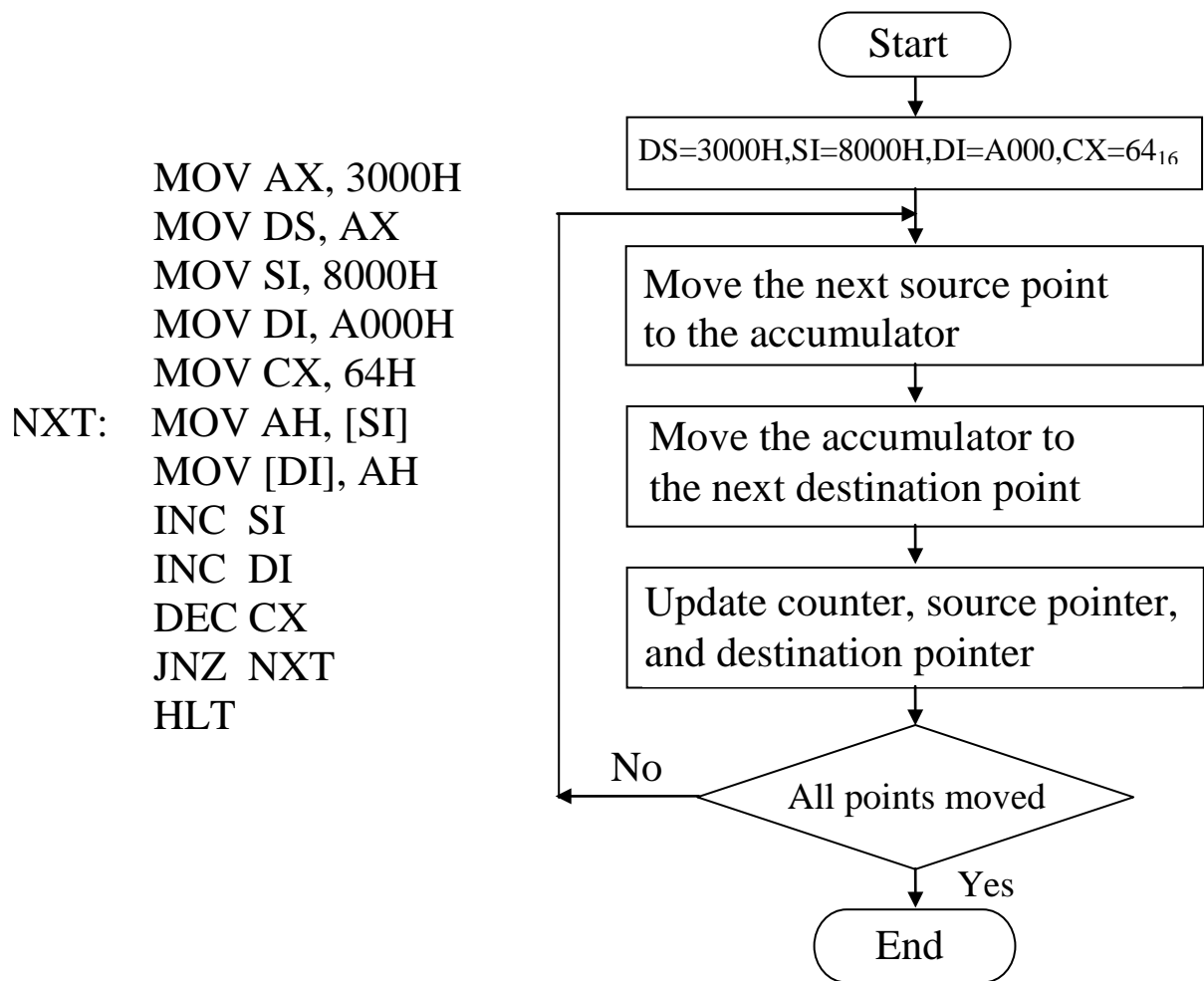
JNE	Not equal	ZF=0
JNG	Not greater	((SF xor OF)) or ZF=1
JNGE	Not greater nor equal	SF xor OF)=1
JNL	Not less	SF=OF
JNLE	Not less nor equal	ZF=0 and SF=OF
JNO	Not overflow	OF=0
JNP	Not parity	PF=0
JNS	Not sign	SF=0
JNZ	Not zero	ZF=0
JO	Overflow	OF=0
JP	Parity	PF=1
JPE	Parity even	PF=1
JPO	Parity odd	PF=0
JS	Sign	SF=1
JZ	Zero	ZF=1

Figure 7-2 (a) conditional jump (b) type of conditional jump instructions.

• **Conditional jump:** The conditional jump instructions test the following flag bits: S, Z, C, P, and O. If the condition under test is true, a branch to the label associated with jump instruction occurs. If the condition is false, the next sequential step in the program executes. Figure 7-2 (a) shows a general form of this instruction; figure 7-2 (b) is a list of each of the conditional jump instructions. The **above** and **below** are used for comparison of **unsigned** numbers, and **less** and **greater** are used for comparison of **signed** numbers.

Example 7-2: Write a program to move a block of 100₁₀ consecutive bytes of data string at offset address 8000H in memory to another block of memory location starting at offset address A000H. Assume that both blocks are in the same data segment value 3000H.

Solution: The flowchart and the program are as shown below:



7-1-2 Subroutine (procedure): A subroutine is a group of instructions that usually performs one task. A subroutine is a reusable section of the software that is stored in memory once, but used as often as necessary. Figure 7-3 illustrates the concept of a subroutine.

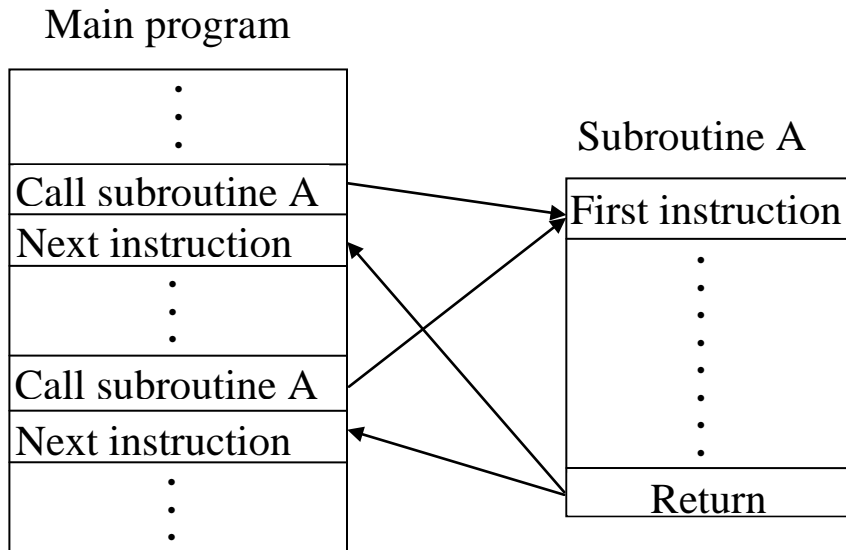


Figure 7-3

• **CALL and RET instructions:** the CALL instruction links to the procedure, and RET instruction returns from procedure. Figure 7-4 shows the CALL instruction.

Like JMP instruction, CALL has two types of operations:

1. Intra-segment call: In this case execution of instruction CALL case:

- The content of IP (the offset address of the instruction that immediately follows the CALL) is saved on the stack.
- The stack pointer (SP) is decremented by two.
- The operand of CALL instruction (the offset of the first instruction of the subroutine) is loaded into IP.

2. Inter-segment call: In this case execution of instruction CALL case:

- The content of CS and IP (the address of the instruction that immediately follows the CALL) is saved on the stack.
- The stack pointer (SP) is decremented by four.
- The operand of CALL instruction (the segment & offset of the first instruction of the subroutine) is loaded into IP and CS.

				Operation
				Near-proc
				Far-proc
				Memptr16
				Regptr16
				Memptr32
Mnem	Meaning	Format	Operation	Flags
CALL	Subroutine call	CALL operand	Execution contents from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack	None

(a)

(b)

Figure 7-4 (a) CALL instruction.(b) Allowed operand

Every subroutine must be ended by executing a RET instruction. Fig. 7-5 shows the RET instruction. Note that its execution cause the value of IP or both the value of IP and CS that were saved on the stack to be returned back to their corresponding registers and the stack pointer to be adjusted appropriately. In this way, program control is returned to the instruction that immediately follows the CALL instruction in program memory.

Example 7-3: Write a subroutine to display a character stored in DL register. Use this subroutine to print ‘TEST’ word.

Solution: The INT 21 interrupt instruction is used to print a character stored in DL register. The subroutine and the program are as shown below:

MOV DL, 'T'
CALL M1
MOV DL, 'E'
CALL M1
MOV DL, 'S'
CALL M1
MOV DL, 'T'
CALL M1
HLT

Main program

M1 PROC
MOV AH, 2
INT 21H
RET
M1 ENDP

subroutine

Mnem	Meaning	Format	Operation	Flags			
				<table> <tr><th>Operand</th></tr> <tr><td>None</td></tr> <tr><td>Disp16</td></tr> </table>	Operand	None	Disp16
Operand							
None							
Disp16							
RET	Return	RET or RET operand	Return to the main program by restoring IP (and CS for far-proc). If operand is present, it is added to the contents of SP.	None			

(a)

(b)

Figure 7-5 (a) RET instruction.(b) Allowed operand

7-1-3 LOOPS AND LOOP-HANDLING INSTRUCTIONS

Loop instructions can be used in place of certain conditional jump instructions and give the programmer a simpler way of writing loop sequences. The loop instructions are listed in Fig. 7-6.

The first instruction, *loop* (LOOP), works with respect to the

contents of the CX register. CX must be preloaded with a count that represents the number of times the loop is to repeat. Whenever LOOP is executed, the contents of CX are first decremented by one and then checked to determine if they are equal to zero. If equal to zero, the loop is complete and the instruction following LOOP is executed; otherwise, control is returned to the instruction at the label specified in the loop instruction. In this way, we see that LOOP is a single instruction that functions the same as a decrement CX instruction followed by a JNZ instruction. The other two loop instructions in figure 7-6 operate in a similar way except that they check for two conditions.

Mnem	Meaning	Format	Operation
LOOP	Loop	LOOP short-label	(CX) \leftarrow (CX)-1 Jump to location defined by short-label if (CX) = 0; otherwise, execute next instruction
LOOP/ LOOPZ	Loop while equal/ Loop while zero	LOOPE/LOOPZ short-label	(CX) \leftarrow (CX)-1 Jump to location defined by short-label if (CX) \neq 0; and (ZF)=1; otherwise, execute next instruction
LOOPNE/ LOOPNZ	Loop while not equal/ Loop while not zero	LOOPNE/LOOPNZ short-label	(CX) \leftarrow (CX)-1 Jump to location defined by short-label if (CX) \neq 0; and (ZF)=0; otherwise, execute next instruction

Figure 7-6 Loop instructions.

Example 7-4: Rewrite a program in example 7-2 use LOOP instruction.

Solution: The program is as shown below:

```

MOV AX, 3000H
MOV DS, AX
MOV SI, 8000H
MOV DI, A000H
MOV CX, 64H
NXT: MOV AH, [SI]
      MOV [DI], AH
      INC SI
      INC DI
      LOOP NXT
      HLT

```

7-2 FLAG CONTROL INSTRUCTIONS: These instructions directly affected the state of flags. Figure 7-7 shows these instructions.

Mnem	Meaning	Operation	Flags
LAHF	Load AH from flags	$(AH) \leftarrow (\text{Flags})$	None
SAHF	Store AH into flags	$(\text{Flags}) \leftarrow (AH)$	S,Z,A,P,C
CLC	Clear carry flag	$(CF) \leftarrow 0$	C
STC	Set carry flag	$(CF) \leftarrow 1$	C
CMC	Complement carry flag	$(CF) \leftarrow (\overline{CF})$	C
CLI	Clear interrupt flag	$(IF) \leftarrow 0$	I
STI	Set interrupt flag	$(IF) \leftarrow 1$	I



(b)

(a)

Figure 7-7 (a) flag control instructions. (b) Format of the flags in AH register for the LAHF and SAHF instructions.

Example 7-5: Write a program to save the contents of flags in the memory location at offset 1111H of the data segment and then reload the flags with the contents of the storage location of offset 2222H

Solution: The program is as shown below:

```

LAHF
MOV [1111H], AH

```

```
MOV AH, [2222H]  
SAHF  
MOV DI, A000H  
HLT
```