University of Duhok
College of Science

Computer Organization

The 80x86 Microprocessor

Department of Computer Science
First-Year(2nd course) 2022-2021
Jawaher A. Fadhil

CHAPTER 1

THE 80x86 MICROPROCESSOR

SECTION 1.1: BRIEF HISTORY OF THE 80x86 FAMILY

In this section we trace the evolution of Intel's family of microprocessors from the late 1970s, when the personal computer had not yet found widespread acceptance, to the powerful microcomputers widely in use today.

Evolution from 8080/8085 to 8086

In 1978, Intel Corporation introduced a 16-bit microprocessor called the 8086. This processor was a major improvement over the previous generation 8080/8085 series Intel microprocessors in several ways. First, the 8086's capacity of 1 megabyte of memory exceeded the 8080/8085's capability of handling a maximum of 64K bytes of memory. Second, the 8080/8085 was an 8-bit system, meaning that the microprocessor could work on only 8 bits of data at a time. Data larger than 8 bits had to be broken into 8-bit pieces to be processed by the CPU. In contrast, the 8086 is a 16-bit microprocessor. Third, the 8086 was a pipelined processor, as opposed to the nonpipelined 8080/8085. In a system with pipelining, the data and address buses are busy transferring data while the CPU is processing information, thereby increasing the effective processing power of the microprocessor. Although pipelining was a common feature of mini- and mainframe computers, Intel was a pioneer in putting pipelining on a single-chip microprocessor. Pipelining is discussed further in Section 1.2.

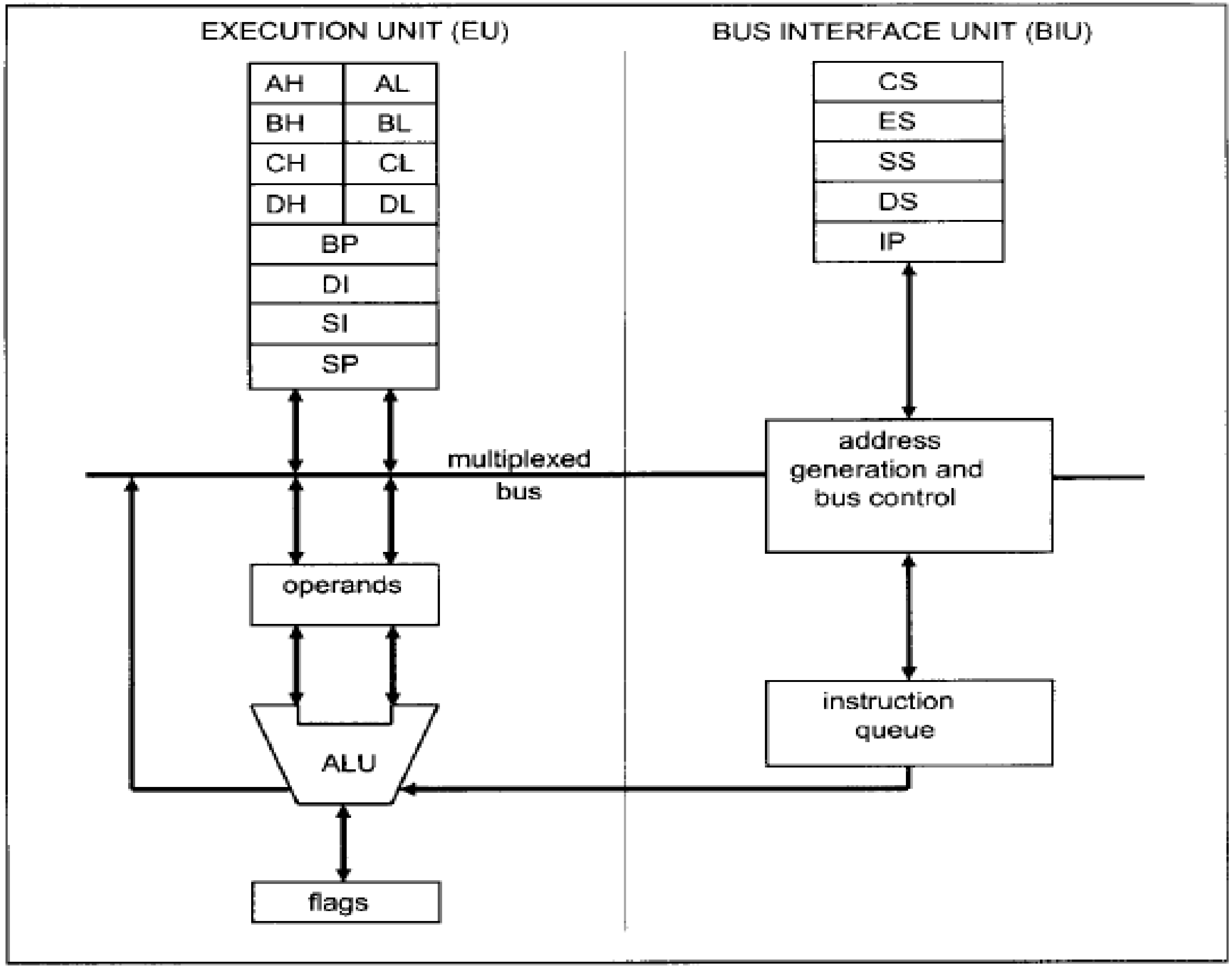


Figure 1-1. Internal Block Diagram of the 8088/86 CPU
(Reprinted by permission of Intel Corporation, Copyright Intel Corp. 1989)

SECTION 1.2: INSIDE THE 8088/8086

In this section we explore concepts important to the internal operation of the 8088/86, such as pipelining and registers. See the block diagram in Figure 1-1.

Pipelining

There are two ways to make the CPU process information faster: increase the working frequency or change the internal architecture of the CPU. The first option is technology dependent, meaning that the designer must use whatever technology is available at the time, with consideration for cost. The technology and materials used in making ICs (*integrated circuits*) determine the working frequency, power consumption, and the number of transistors packed into a single-chip microprocessor. A detailed discussion of IC technology is beyond the scope of this book. It is sufficient for the purpose at hand to say that designers can make the CPU work faster by increasing the frequency under which it runs if technology and cost allow. The second option for improving the processing power of the CPU has to do with the internal working of the CPU. In the 8085 microprocessor, the CPU could either

fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on. The idea of *pipelining* in its simplest form is to allow the CPU to fetch and execute at the same time as shown in Figure 1-2. It is important to point out that Figure 1-2 is not meant to imply that the amount of time for fetch and execute are equal.

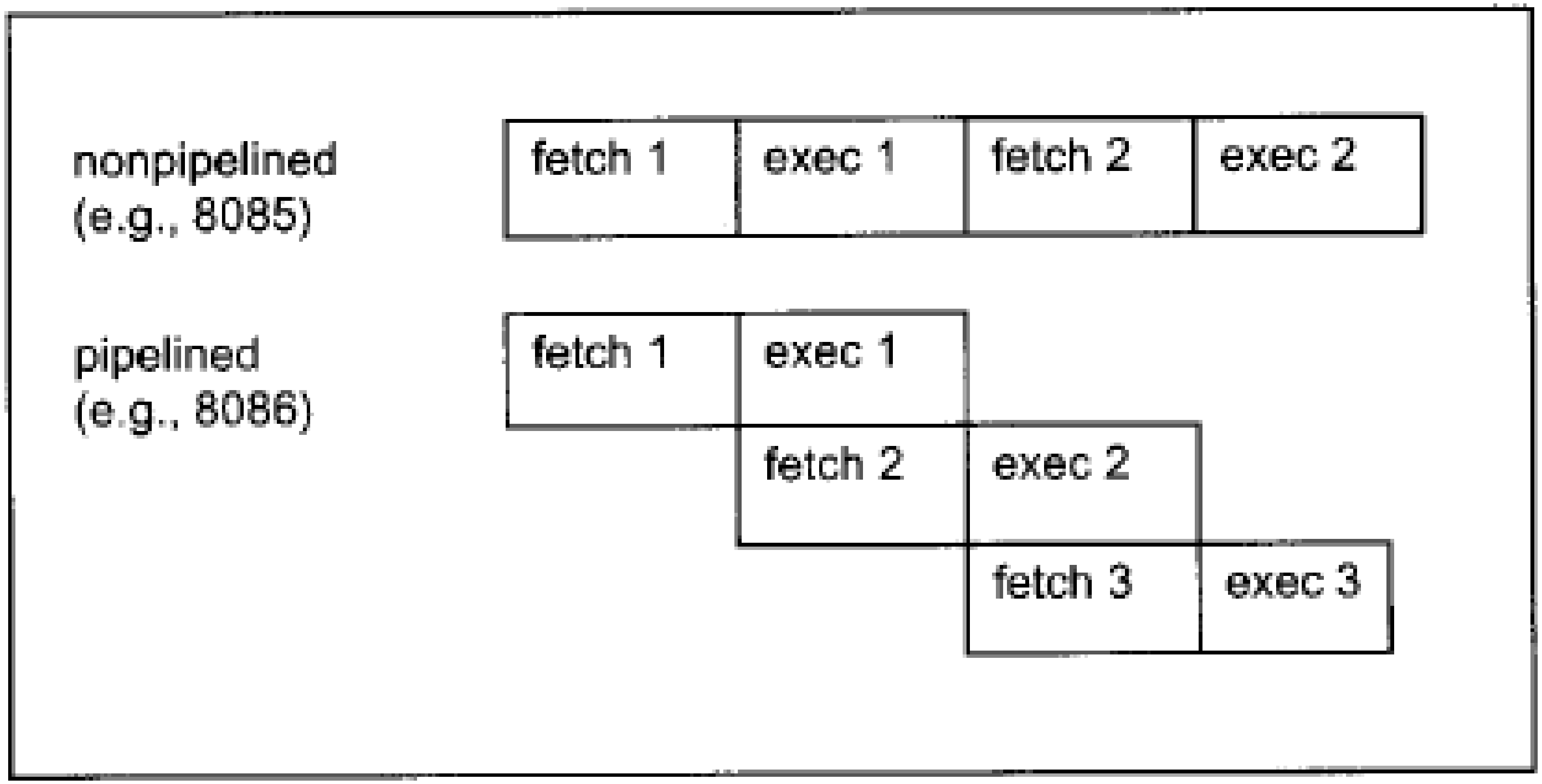


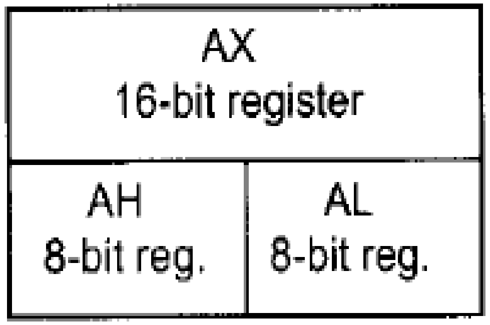
Figure 1-2. Pipelined vs. Nonpipelined Execution

Intel implemented the concept of pipelining in the 8088/86 by splitting the internal structure of the microprocessor into two sections: the *execution unit* (EU) and the *bus interface unit* (BIU). These two sections work simultaneously. The BIU accesses memory and peripherals while the EU executes instructions previously fetched. This works only if the BIU keeps ahead of the EU; thus the BIU of the 8088/86 has a buffer, or queue (see Figure 1-1). The buffer is 4 bytes long in the 8088 and 6 bytes in the 8086. If any instruction takes too long to execute, the queue is filled to its maximum capacity and the buses will sit idle.

Registers

Registers

In the CPU, registers are used to store information temporarily. That information could be one or two bytes of data to be processed or the address of data. The registers of the 8088/86 fall into the six categories outlined in Table 1-2. The general-purpose registers in 8088/86 microprocessors can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In the 8088/86, data types are either 8 or 16 bits. To access 12-bit data, for example, a 16-bit register must be used with the highest 4 bits set to 0. The bits of a register are numbered in descending order, as shown below.



8-bit register:

D7	D6	D5	D4	D3	D2	D1	D0

16-bit register:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Different registers in the 8088/86 are used for different functions, and since some instructions use only specific registers to perform their tasks, the use of registers will be described in the context of instructions and their application in a given program. The first letter of each general register indicates its use. AX is used for the accumulator, BX as a base addressing register, CX is used as a counter in loop operations, and DX is used to point to data in I/O operations.

Table 1-2: Registers of the 8086/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Note:
The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

Review Questions

1. Explain the functions of the EU and the BIU.
2. What is pipelining, and how does it make the CPU execute faster?
3. Registers of the 8086 are either _____ bits or _____ bits in length.
4. List the 16-bit registers of the 8086.

Assembly language programming

MOV instruction

Assembly language programming

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items. We introduce Assembly language programming with two widely used instructions: the move and add instructions.

MOV instruction

Simply stated, the MOV instruction copies data from one location to another. It has the following format:

```
MOV     destination,source          ;copy source operand to destination
```

This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand. For example, the instruction "MOV DX,CX" copies the contents of register CX to register DX. After this instruction is executed, register DX will have the same value as register CX. The MOV instruction does not affect the source operand. The following program first loads CL with value 55H, then moves this value around to various registers inside the CPU.

```
MOV     CL,55H          ;move 55H into register CL
MOV     DL,CL
MOV     AH,DL
MOV     AL,AH
MOV     BH,CL
MOV     CH,BH
MOV AX,133H
MOV BH,AL
MOV BL,AH
```

```
MOV CX,2F
MOV BL,9C
MOV CH,BL
MOV AX,CX
```

```
MOV DL,A0
MOV SI,0000
MOV DX,SI
MOV CS,DX
MOV DX,022
MOV AX,2F9
MOV DL,2A
MOV CX,AX
MOV AX,DX
MOV DX,CX
```

Q/ What is the value of the following Registers after executing the above mov instructions?

Ax=	
DX=	
CX=	

The use of 16-bit registers is demonstrated below.

```
MOV    CX,468FH      ;move 468FH into CX (now CH=46,CL=8F)
MOV    AX,CX         ;copy contents of CX to AX (now AX=CX=468FH)
MOV    DX,AX         ;copy contents of AX to DX (now DX=AX=468FH)
MOV    BX,DX         ;copy contents of DX to BX (now BX=DX=468FH)
MOV    DI,BX         ;now DI=BX=468FH
MOV    SI,DI         ;now SI=DI=468FH
MOV    DS,SI         ;now DS=SI=468FH
MOV    BP,DI         ;now BP=DI=468FH
```

In the 8086 CPU, data can be moved among all the registers shown in Table 1-2 (except the flag register) as long as the source and destination registers match in size. Code such as "MOV AL,DX" will cause an error, since one cannot move the contents of a 16-bit register into an 8-bit register. The exception of the flag register means that there is no such instruction as "MOV FR,AX". Loading the flag register is done through other means, discussed in later chapters.

If data can be moved among all registers including the segment registers, can data be moved directly into all registers? The answer is no. Data can be moved directly into nonsegment registers only, using the MOV instruction. For example, look at the following instructions to see which are legal and which are illegal.

```
MOV    AX,58FCH     ;move 58FCH into AX    (LEGAL)
MOV    DX,6678H     ;move 6678H into DX    (LEGAL)
MOV    SI,924BH     ;move 924B  into SI    (LEGAL)
MOV    BP,2459H     ;move 2459H into BP    (LEGAL)
MOV    DS,2341H     ;move 2341H into DS    (ILLEGAL)
MOV    CX,8876H     ;move 8876H into CX    (LEGAL)
MOV    CS,3F47H     ;move 3F47H into CS    (ILLEGAL)
MOV    BH,99H       ;move 99H into BH      (LEGAL)
```

From the discussion above, note the following three points:

1. Values cannot be loaded directly into any segment register (CS, DS, ES, or SS). To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register, as shown next.

```
MOV    AX,2345H        ;load 2345H into AX
MOV    DS,AX           ;then load the value of AX into DS

MOV    DI,1400H        ;load 1400H into DI
MOV    ES,DI           ;then move it into ES, now ES=DI=1400
```

2. If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros. For example, in "MOV BX,5" the result will be BX = 0005; that is, BH = 00 and BL = 05.

3. Moving a value that is too large into a register will cause an error.

```
MOV    BL,7F2H         ;ILLEGAL: 7F2H is larger than 8 bits
MOV    AX,2FE456H      ;ILLEGAL: the value is larger than AX
```

ADD instruction

The ADD instruction has the following format:

```
ADD    destination,source        ;ADD the source operand to the destination
```

The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination. To add two numbers such as 25H and 34H, each can be moved to a register and then added together:

```
MOV    AL,25H          ;move 25 into AL
MOV    BL,34H          ;move 34 into BL
ADD    AL,BL           ;AL = AL + BL
```

Executing the program above results in AL = 59H (25H + 34H = 59H) and BL = 34H. Notice that the contents of BL do not change. The program above can be written in many ways, depending on the registers used. Another way might be:

```
MOV    DH,25H          ;move 25 into DH
MOV    CL,34H          ;move 34 into CL
ADD    DH,CL           ;add CL to DH: DH = DH + CL
```

The largest number that an 8-bit register can hold is FFH. To use numbers larger than FFH (255 decimal), 16-bit registers such as AX, BX, CX, or DX must be used. For example, to add two numbers such as 34EH and 6A5H, the following program can be used:

```
MOV    AX,34EH         ;move 34EH into AX
MOV    DX,6A5H         ;move 6A5H into DX
ADD    DX,AX           ;add AX to DX: DX = DX + AX
```

34E

+

6A5

Running the program above gives DX = 9F3H (34E + 6A5 = 9F3) and AX = 34E. Again, any 16-bit nonsegment registers could have been used to perform the action above:

```
MOV    CX,34EH          ;load 34EH into CX
ADD    CX,6A5H          ;add 6A5H to CX (now CX=9F3H)
```

The general-purpose registers are typically used in arithmetic operations. Register AX is sometimes referred to as the accumulator.

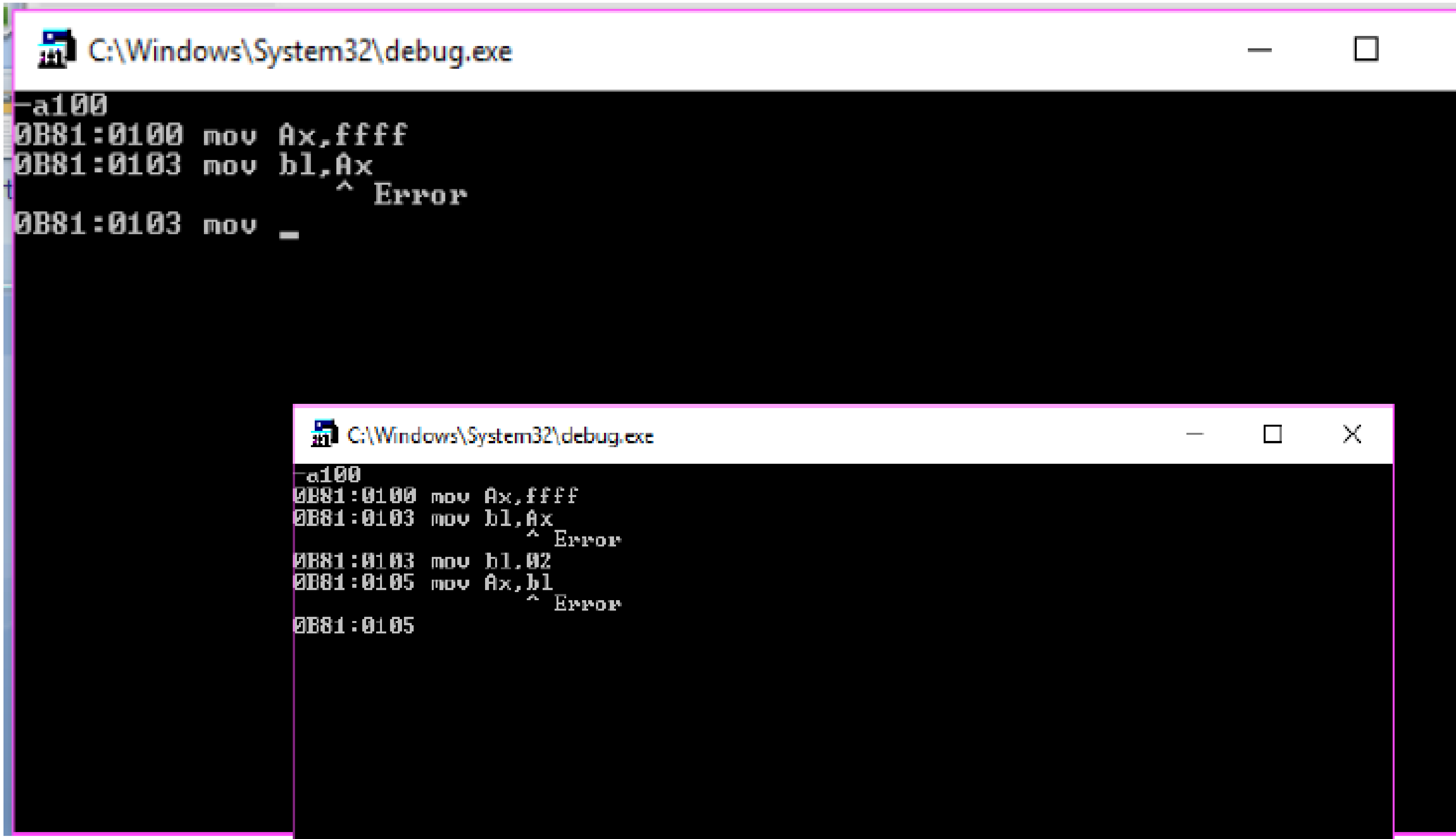
Review Questions

1. Write the Assembly language instruction to move value 1234H into register BX.
2. Write the Assembly language instructions to add the values 16H and ABH. Place the result in register AX.
3. No value can be moved directly into which registers?
4. What is the largest hex value that can be moved into a 16-bit register? Into an 8-bit register? What are the decimal equivalents of these hex values?

Q/ Write assembly program to add the following hexadecimal numbers(69H+3BH+102H), Store the result in BX register?

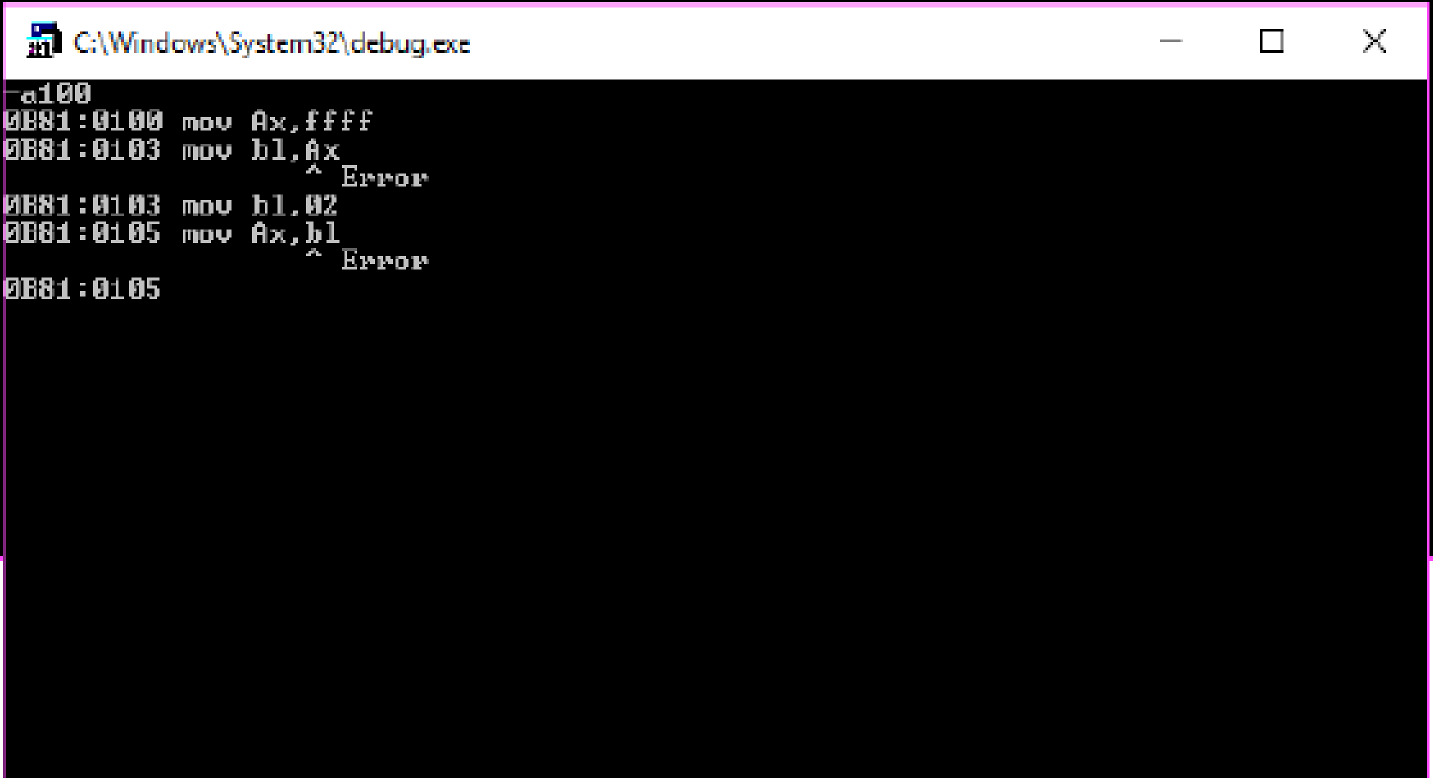
```
    MOV
```

Logical address and Physical address



C:\Windows\System32\debug.exe

```
-a100
0B81:0100 mov Ax,ffff
0B81:0103 mov bl,Ax
                  ^ Error
0B81:0103 mov _
```



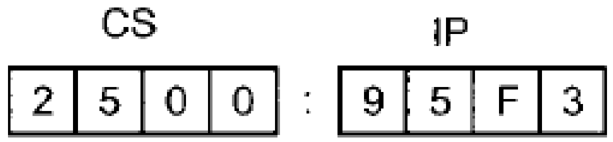
C:\Windows\System32\debug.exe

```
-a100
0B81:0100 mov Ax,ffff
0B81:0103 mov bl,Ax
                  ^ Error
0B81:0103 mov bl,02
0B81:0105 mov Ax,bl
                  ^ Error
0B81:0105
```

Logical address and physical address

In Intel literature concerning the 8086, there are three types of addresses mentioned frequently: the physical address, the offset address, and the logical address. The *physical address* is the 20-bit address that is actually put on the address pins of the 8086 microprocessor and decoded by the memory interfacing circuitry. This address can have a range of 00000H to FFFFFH for the 8086 and real-mode 286, 386, and 486 CPUs. This is an actual physical location in RAM or ROM within the 1 megabyte memory range. The *offset address* is a location within a 64K-byte segment range. Therefore, an offset address can range from 0000H to FFFFH. The *logical address* consists of a segment value and an offset address. The differences among these addresses and the process of converting from one to another is best understood in the context of some examples, as shown next.

Code segment

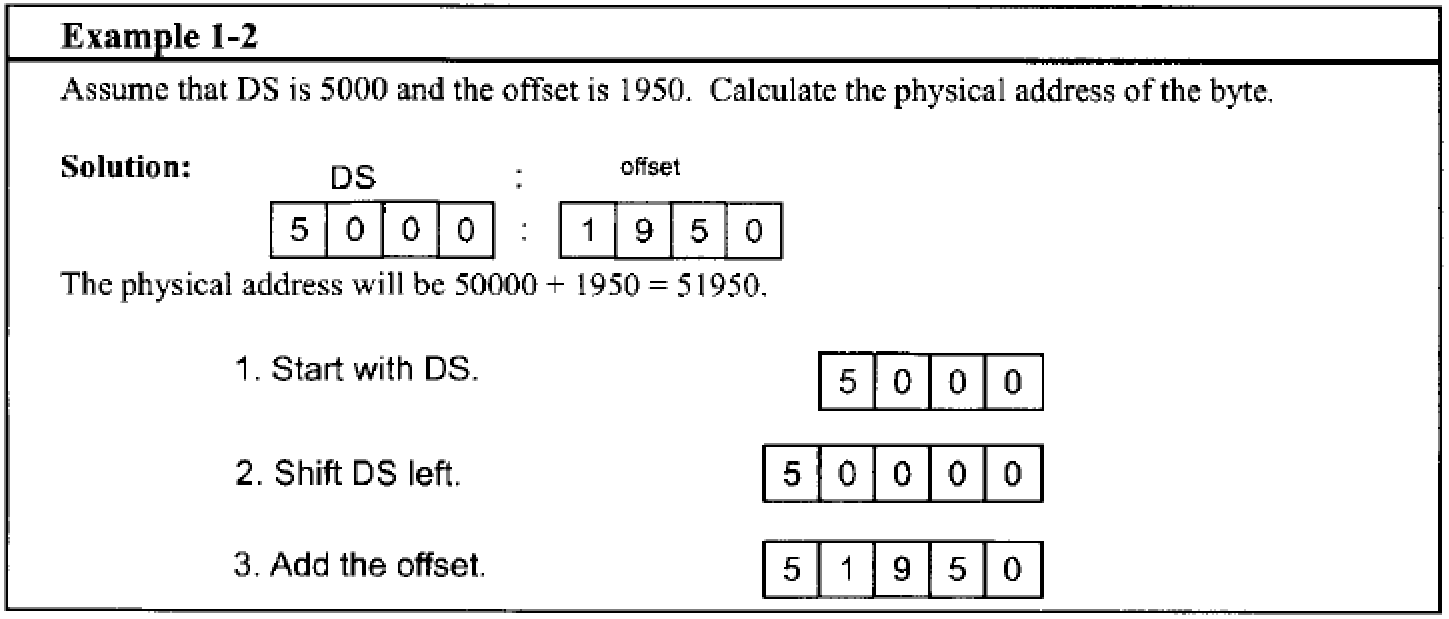


To execute a program, the 8086 fetches the instructions (opcodes and operands) from the code segment. The logical address of an instruction always consists of a CS (code segment) and an IP (instruction pointer), shown in CS:IP format. The physical address for the location of the instruction is generated by shifting the CS left one hex digit and then adding it to the IP. IP contains the offset address. The resulting 20-bit address is called the physical address since it is put on the external physical address bus pins to be decoded by the memory decoding circuitry. To clarify this important concept, assume values in CS and IP as shown in the diagram. The offset address is contained in IP; in this case it is 95F3H. The logical address is CS:IP, or 2500:95F3H. The physical address will be 25000 + 95F3 = 2E5F3H. The physical address of an instruction can be calculated as follows:

Physical add.=Segment*10+Offset

CS					IP			
2	5	0	0	:	9	5	F	3

1. Start with CS.	2 5 0 0
2. Shift left CS.	2 5 0 0 0
3. Add IP.	2 E 5 F 3


Example 1-2

Assume that DS is 5000 and the offset is 1950. Calculate the physical address of the byte.

Solution:

DS				:	offset			
5	0	0	0	:	1	9	5	0

The physical address will be 50000 + 1950 = 51950.

1. Start with DS.	5 0 0 0
2. Shift DS left.	5 0 0 0 0
3. Add the offset.	5 1 9 5 0

Example 1-1

If CS = 24F6H and IP = 634AH, show:
(a) The logical address
(b) The offset address
and calculate:
(c) The physical address
(d) The lower range
(e) The upper range of the code segment

Solution:

(a) 24F6:634A
(b) 634A
(c) 2B2AA (24F60 + 634A)
(d) 24F60 (24F60 + 0000)
(e) 34F5F (24F60 + FFFF)

Example 1-3

If DS = 7FA2H and the offset is 438EH,
(a) Calculate the physical address.
(b) Calculate the lower range.
(c) Calculate the upper range of the data segment.
(d) Show the logical address.

Solution:

(a) 83DAE (7FA20 + 438E)
(b) 7FA20 (7FA20 + 0000)
(c) 8FA1F (7FA20 + FFFF)
(d) 7FA2:438E

INC increment
Format: INC destination   dest.=dest.+1

```
 Example: inc ax
          inc si
          INC DL
```

DEC   Decrement
Format: DEC dest.   dest.=dest.-1

```
Example: dec ax
         dec di
         DEC CL
```

Q/ What is the AX value after executing the following instructions:

```
MOV AX,091B
DEC AL
DEC AL
INC AH
INC AH
INCAH
```

```
MOV AX,091B
DEC AX
DEC AX
INC AX
INC AX
INC AX
```

thank you