Week 2
The 80x86 Microprocessor Architecture

OBJECTIVES

this chapter enables the student to:

- Describe the Intel family of microprocessors from 8085 to Pentium®.
  - In terms of bus size, physical memory & special features.
- Explain the function of the EU (execution unit) and BIU (bus interface unit).
- Describe pipelining and how it enables the CPU to work faster.
- List the registers of the 8086.
- Code simple MOV and ADD instructions.
  - Describe the effect of these instructions on their operands.

OBJECTIVES
this chapter enables the student to:

- State the purpose of the code segment,data segment,stack segment, and extra segment.
- Explain the difference between a logical address and a physical address.
- Describe the "*little endian*" storage convention of x86 microprocessors.
- State the purpose of the stack.
- Explain the function of PUSH and POP instructions.
- List the bits of the flag register and briefly state the purpose of each bit.

Brief History of the 80x86 Family

- Evolution from 8080/8085 to 8086
  - In 1987, Intel introduced a 16-bit microprocessor called the 8086
  - It was a major improvement over the previous generation 8080/8085 microprocessors
    - 1 Mbyte memory (20 address lines) vs 8080/8085's capability of 64 Kbytes
    - 8080/8085 was an 8 bit system, meaning that the data larger than 8 bits should be broken into 8-bit pieces to be processed by the CPU; in contrast 8086 is a 16 bit microprocessor
    - 8086 is **pipelined** vs nonpipelined 8080/8085; in a system with pipelining the data and address busses are busy transferring data while the CPU is processing information
- Evolution from 8086 to 8088
  - 8086 is a microprocessor with a 16-bit data bus internally and externally
  - Internally all registers are 16 bits wide
  - External the data bus is16 bits to transfer data in and out of the CPU
  - There was a resistance in using the 16 bit external data bus since at that time peripherals were designed around 8-bit microprocessors
  - Intel then came out with the 8088 version with 8-bit data bus

Brief History - Continued

- Success of 8088
  - IBM picked up the 8088 as their microprocessor of choice in designing the IBM PC
  - All specification of the hardware and software of the PC are made public by IBM and Microsoft (in contrast with Apple computers)
- Intel introduced 80286 in 1982
  - 16 bit internal and external data buses
  - 24 address lines (16 Mbyte main memory)
  - **Virtual memory:** a way of fooling the microprocessor into thinking that it has access to almost **unlimited** amount of memory by swapping data between disk storage and RAM
  - **Real mode vs protected mode with 80286**
- Intel unveiled the 80386 (sometimes called the 80386DX) in 1985; internally and externally a 32 bit microprocessor with a 32 bit address bus (4 Gbyte physical memory)
  - Numeric data processing chips were made available: 8087, 80287, 80387 etc.
- On the 80486, in 1989, Intel put a greatly enhanced 80386 & math coprocessor on a single chip.
  - Plus additional features such as *cache memory.*
    - Cache memory is static RAM with a very fast access time.
- All programs written for the 8088/86 will run on 286, 386, and 486 computers.

The 80286 and above - Modes of Operation

- **Real Mode (We will use this in our course)**
  - The address space is limited to 1MB using address lines A0-19; the high address lines are inactive
  - The segmented memory addressing mechanism of the 8086 is retained with each segment limited to 64KB
  - Two new features are available to the programmer
    - –Access to the 32 bit registers
    - –Addition of two new segments F and G

- **Protected Mode**
  - –Difference is in the new addressing mechanism and protection levels
  - –Each memory segment may range from a single byte to 4GB
  - –The addresses stored in the segment registers are now interpreted as pointers into a descriptor table
  - –Each segment's entry in this table is eight bytes long and identifies the base address of the segment, the segment size, and access rights
  - –In 8088/8086 any program can access the core of the OS hence crash the system. Access Rights are added in descriptor tables.

Brey 59

Virtual Memory

- 286 onward supported Virtual Memory Management and Protection™
- Unlimited amount of main memory assumed
- Two methods are used:
  - Segmentation
  - Paging
- Both techniques involve swapping blocks of user memory with hard disk space as necessary
  - If the program needs to access a block of memory that is indicated to be stored in the disk, the OS searches for an available memory block (typically using a least recently used algorithm) and swaps that block with the desired data on the hard drive
  - Memory swapping is invisible to the user
  - Segmentation: the block size is variable ranging up to 4GB
  - Paging: Block sizes are always 4 KB at a time.
- A final protected mode feature is the ability to assign a privilege level to individual tasks (programs). Tasks of lower privilege level cannot access programs or data with a higher privilege level. The OS can run multiple programs each protected from each other.

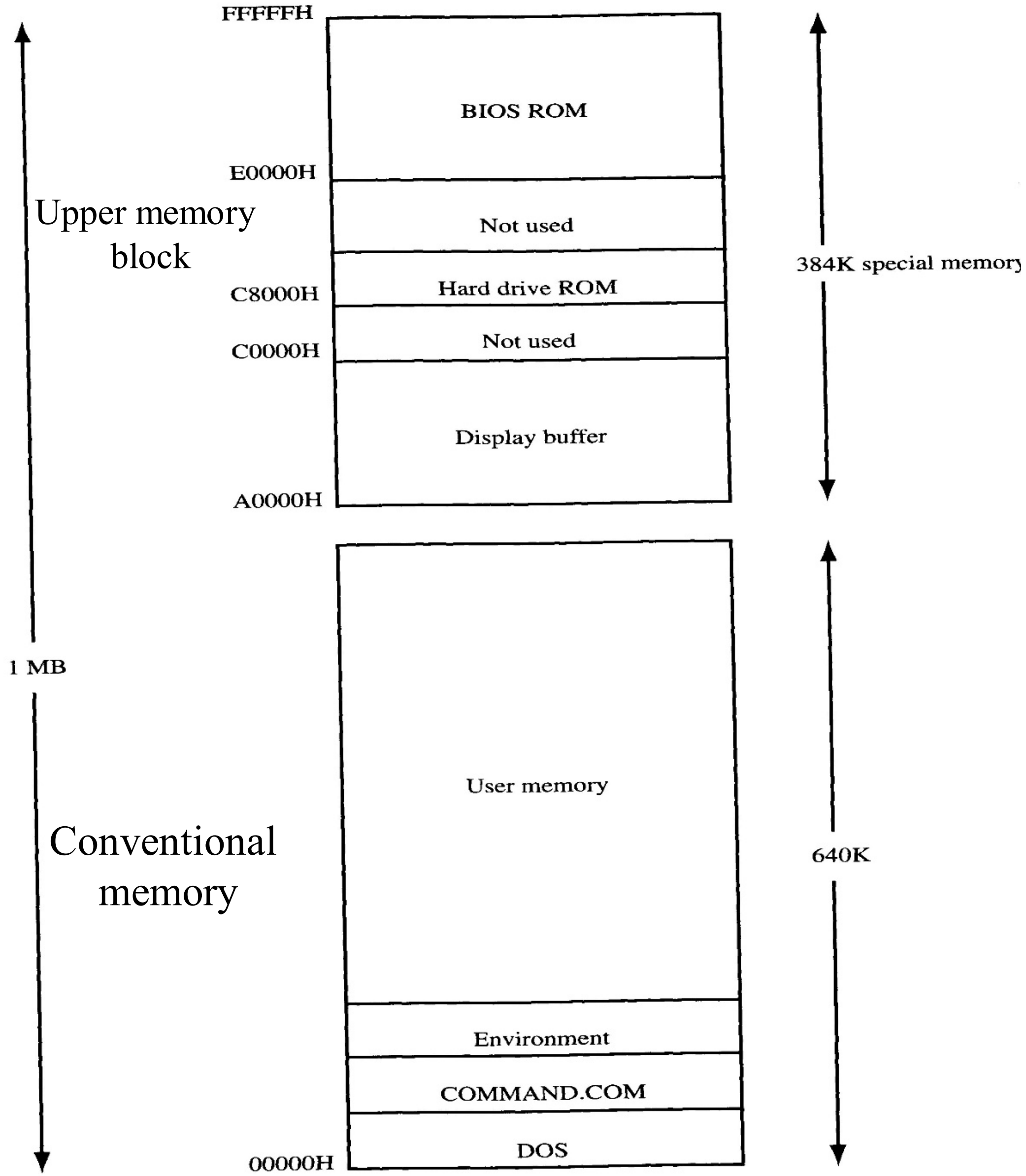
Mazidi 648

Memory Map of a PC

The 640 K Barrier:
DOS was designed to run on the original IBM PC - 8088 microprocessor - with 1Mbytes of main memory

IBM divided this 1Mb address space into specific blocks

- 640 K of RAM (user RAM)
- 384 K reserved for ROM functions (control programs for the video system, hard drive controller, and the basic input/output system)



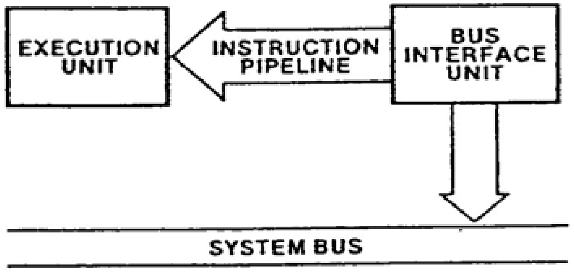
Read Mode vs. Virtual 8086 Mode

- Real Mode (repeat of previous definition)
  - Only one program can be run one time
  - All of the protection and memory management functions are turned off
  - Memory space is limited to 1MB
- Virtual 8086 Mode
  - The processor hands each real mode program its own 1MB chunk of memory
  - Multiple 8086 programs to be run simultaneously but protected from each other (multiple MSDOS prompts)
  - Due to time sharing, the response becomes much slower as each new program is launched
  - The processor can be operated in Protected Mode and Virtual 8086 mode at the same time.
  - Because each 8086 task is assigned the lowest privilege level, access to programs or data in other segments is not allowed thus protecting each task.
  - We'll be using the virtual 8086 mode in the lab experiments on PCs that do have either Pentiums processors or 486s etc.

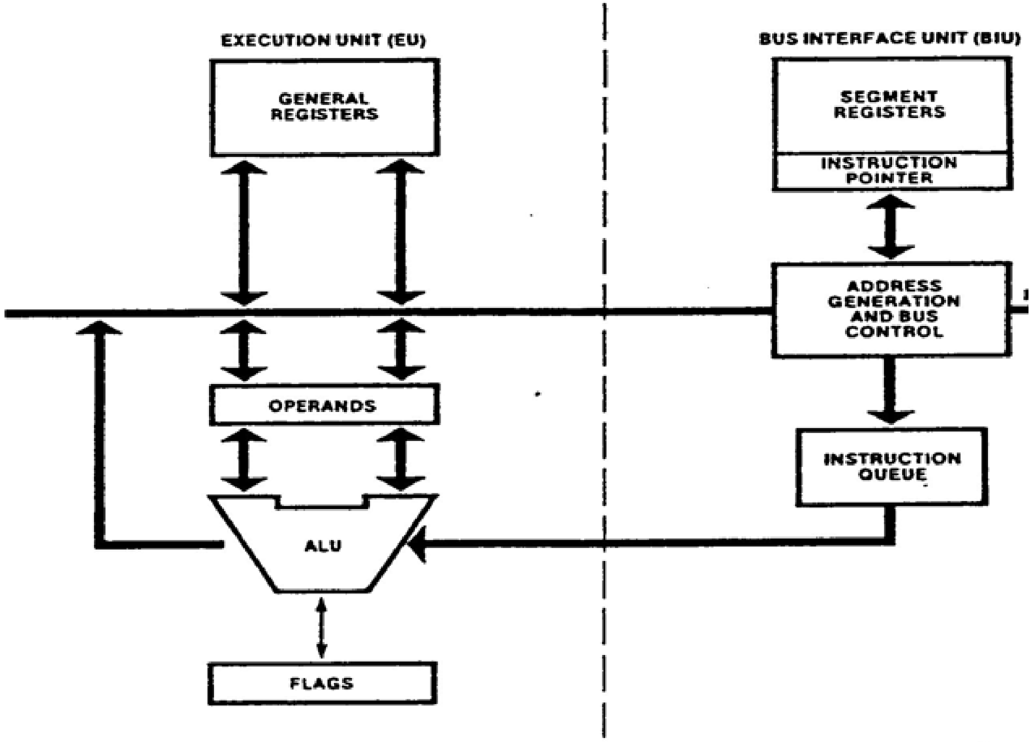
The 8086 and 8088

- The 8086 microprocessor represents the foundation upon which all the 80x86 family of processors have been built
- Intel has made the commitment that as new generations of microprocessors are developed, each will maintain software compatibility with this first generation part.
  - For example, a program designed to run on an **Intel** 386 microprocessor, which also runs on a Pentium, is **upward** compatible.
- Intel implemented pipelining in 8088/86 by splitting the internal structure of the into two sections:
- The execution unit (EU) and the bus interface unit (BIU).
  - These two sections work simultaneously.
- **Processor model**
  - BIU (Bus Interface Unit) provides hardware functions including generation of the memory and I/O addresses for the transfer of data between itself and the outside world
  - EU (Execution Unit) receives program instruction codes and data from the BIU executes these instructions and stores the results in the general registers.
  - EU has no connection to the system buses; it receives and outputs all its data through the BIU.

Execution and Bus Interface Units

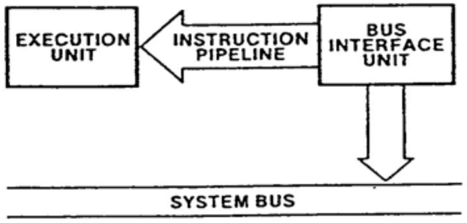


(a)

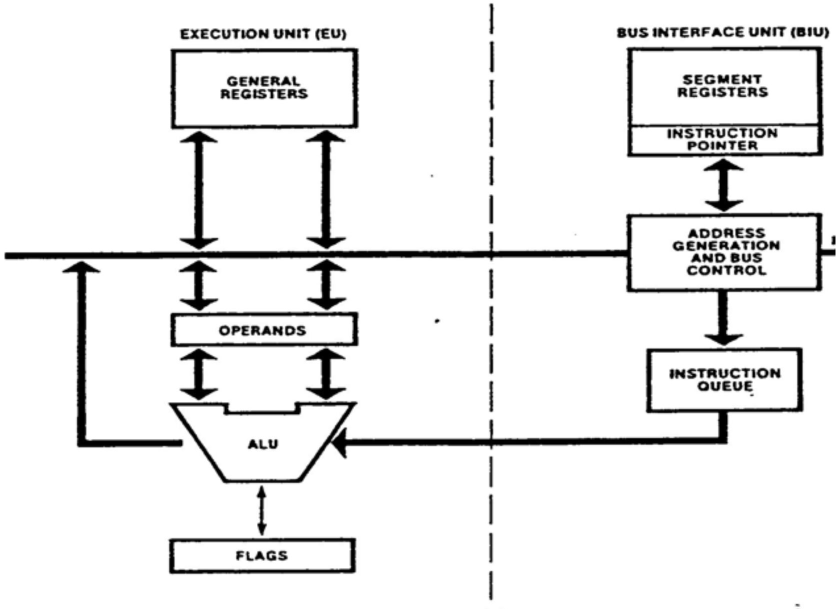


(b)

Fetch and Execute Cycle



(a)



(b)

- Fetch and execute cycles overlap
  - BIU outputs the contents of the IP onto the address bus

    Register IP is incremented by one or more than one for the next instruction fetch

    Once inside the BIU, the instruction is passed to the queue; this queue is a first-in-first-out register sometimes likened to a pipeline

    Assuming that the queue is initially empty the EU immediately draws this instruction from the queue and begins execution

    While the EU is executing this instruction, the BIU proceeds to fetch a new instruction.

    - BIU will fill the queue with several new instructions before the EU is ready to draw its next instruction

    The cycle continues with the BIU filling the queue with instructions and the EU fetching and executing these instructions

Pipelined Architecture

- Three conditions that will cause the EU to enter a wait mode
  - when the instruction requires access to a memory location not in the queue
  - when the instruction to be executed is a jump instruction; the instruction queue should be flushed out (known as branch penalty too much jumping around reduces the efficiency of the program)
  - during the execution of slow instructions
    - for example the instruction AAM (ASCII Adjust for Multiplication) requires 83 clock cycles to complete for an 8086

8088/86 pipelining

- The idea of pipelining in its simplest form is to allow the CPU to fetch *and* execute at the same time.
- 8088/86 pipelining has two stages, *fetch* & *execute.* In more powerful computers, it can have many stages.

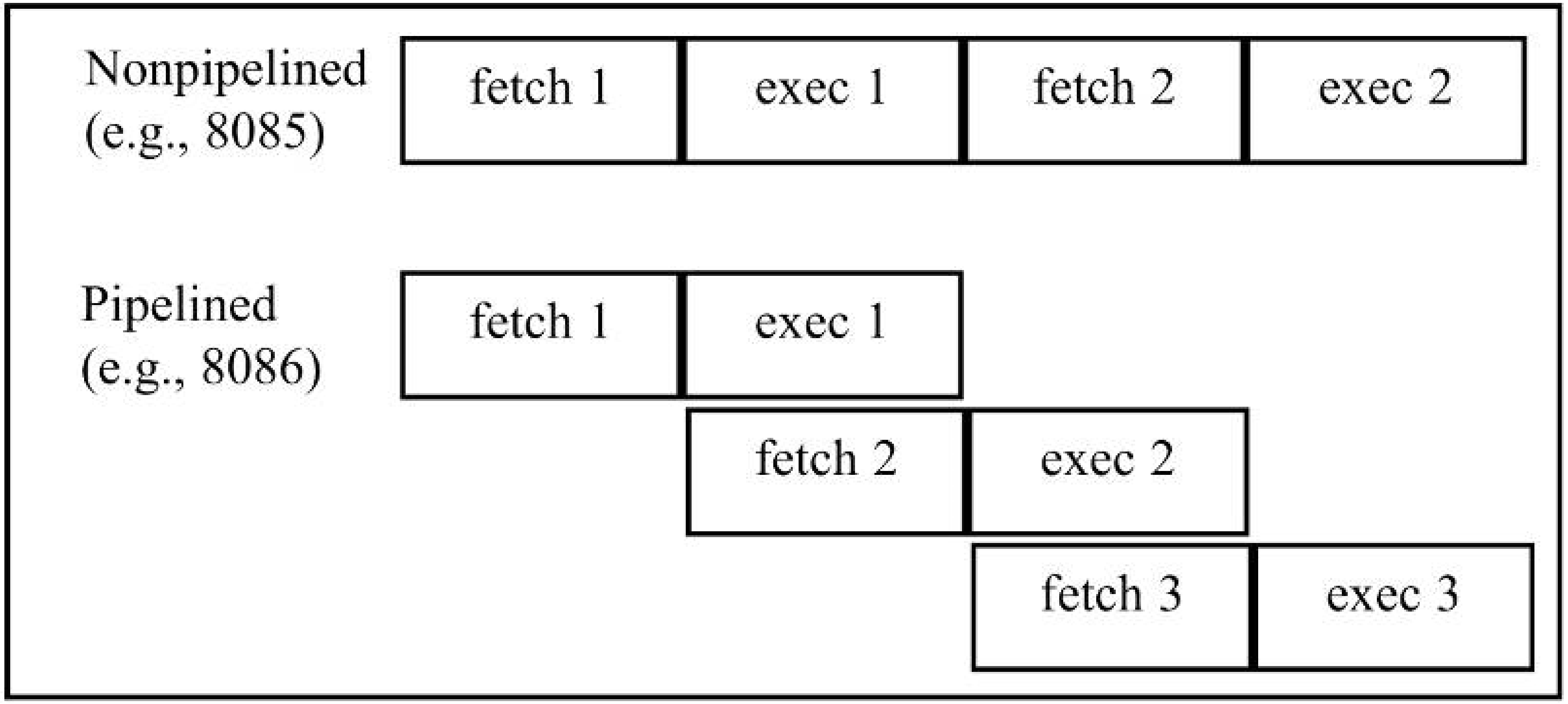


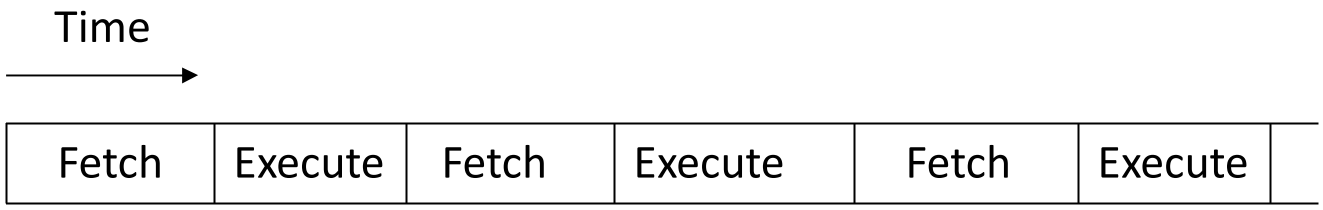
Figure 1-2 Pipelined vs Nonpipelined Execution

Pipelined Architecture

- The BIU accesses memory and peripherals, while the EU executes instructions previously fetched.
  - This works only if the BIU keeps ahead of the EU, so the BIU of the 8088/86 has a buffer, or queue
    - The buffer is 4 bytes long in 8088 and 6 bytes in 8086.

- 8086 vs 8088
  - BIU data bus width 8 bits for 8088, BIU data bus width 16 bits for 8086
  - 8088 instruction queue is four bytes instead of six
  - 8088 is found to be 30% slower than 8086
    - WHY
  - Long instructions provide more time for the BIU to fill the queue

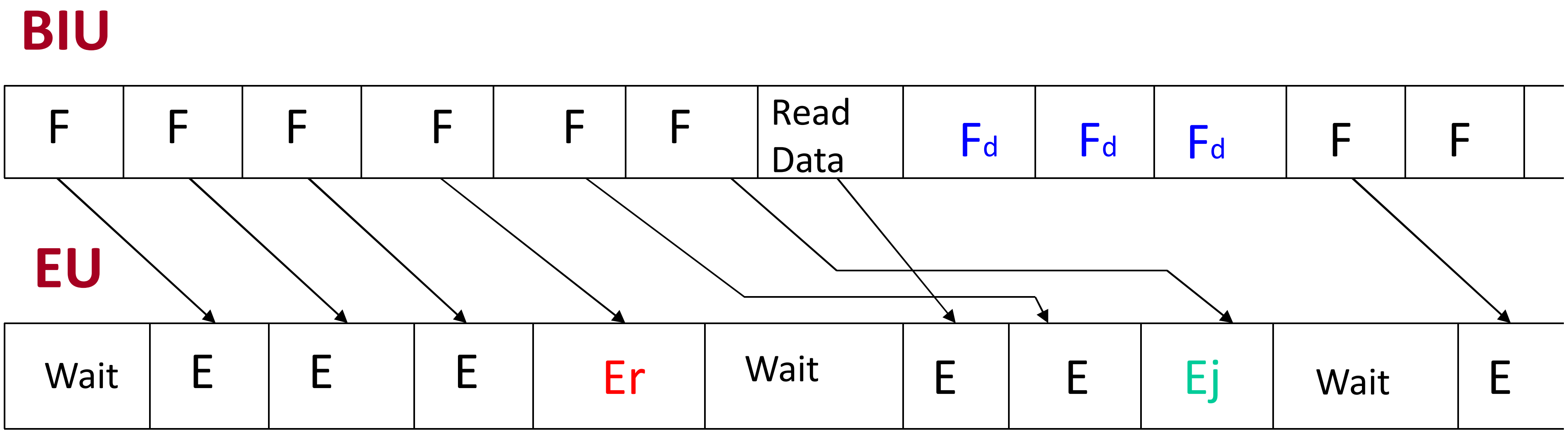
Nonpipelined vs pipelined architecture



Time

Fetch | Execute | Fetch | Execute | Fetch | Execute

Non-pipelined architecturea



BIU

F F F F F F Read Data F_d F_d F_d F F

EU

Wait E E E Er Wait E E Ej Wait E

Pipelined architecture

Er: a request for data not in the queue

Ej: jump instruction occurs

Fd: Discarded

INSIDE THE 8088/86 pipelining

- Three conditions for failure!
  - If an instruction **takes too long to execute**, the queue is filled to capacity and the buses will sit idle AAM (ASCII Adjust for Multiplication) requires 83 clock cycles to complete for an 8086
  - **In some circumstances, the microprocessor must flush out the queue.**
    - When a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information fetched previously is discarded.
    - The EU must wait until the BIU fetches the new instruction
      - In computer science terminology, a branch penalty.
    - In a pipelined CPU, too much jumping around reduces the efficiency of a program.
  - when the instruction requires **access to a memory location** not in the queue

Registers of the 8086/80286 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), Base Pointer (BP)
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)

General Purpose Registers

15 H(HIGH) 8	7 L(LOW) 0
AX (Accumulator)	
AH	AL
BX (Base Register)	
BH	BL
CX (Used as a counter)	
CH	CL
DX (Used to point to data in I/O operations)	
DH	DL

- **Data Registers are normally used for storing temporary results that will be acted upon by subsequent instructions**
- **Each of the registers is 16 bits wide (AX, BX, CX, DX)**
- **General purpose registers can be accessed as either 16 or 8 bits e.g., AH: upper half of AX, AL: lower half of AX**

Data Registers

Register	Operations
AX	Word multiply, word divide, word I/O
AL	Byte multiply, byte divide, byte I/O, decimal arithmetic
AH	Byte multiply, byte divide
BX	Store address information
CX	String operations, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O

Pointer and Index Registers

SP	**Stack Pointer**
BP	**Base Pointer**
SI	**Source Index**
DI	**Destination Index**
IP	**Instruction Pointer**

The registers in this group are all 16 bits wide

Low and high bytes are not accessible

These registers are used as memory pointers

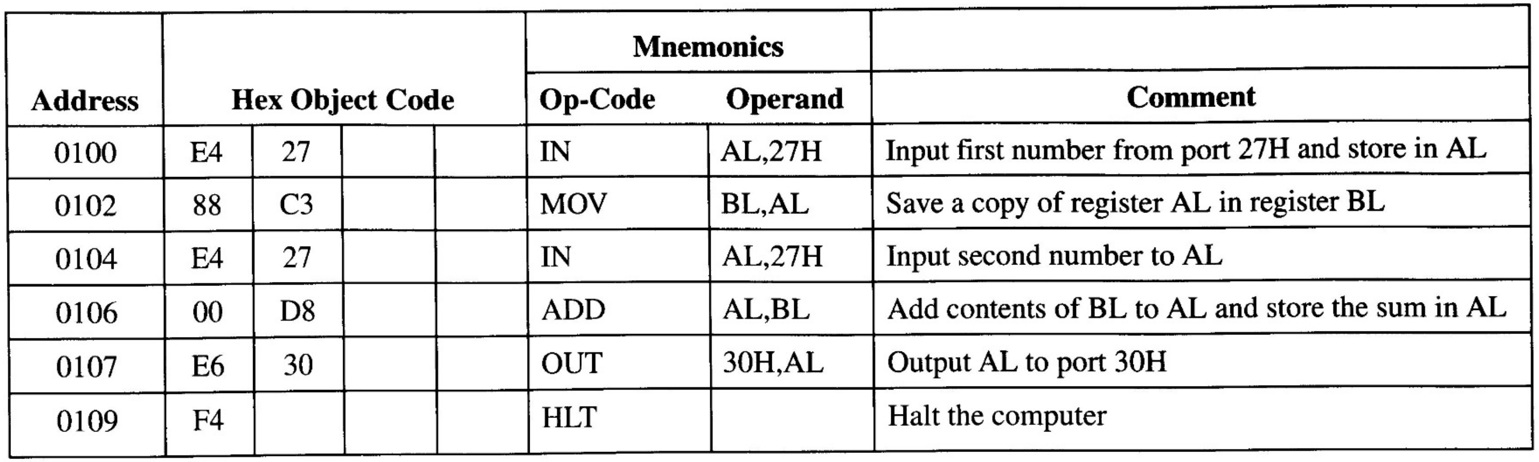
- Example: MOV AH, [SI]

Move the byte stored in memory location
whose address is contained in register SI to register AH

IP is not under direct control of the programmer

Computer Programming

- Machine Language vs Assembly Language
  - Machine language or object code is the only code a computer can execute but it is nearly impossible for a human to work with
  - E4 27 88 C3 E4 27 00 D8 E6 30 F4 the object code for adding two numbers input from the keyboard
- When programming a microprocessor, programmers often use assembly language
  - This involves 3-5 letter abbreviations for the instruction codes (mnemonics) rather than the binary or hex object codes



					Mnemonics		
Address	Hex Object Code				Op-Code	Operand	Comment
0100	E4	27			IN	AL,27H	Input first number from port 27H and store in AL
0102	88	C3			MOV	BL,AL	Save a copy of register AL in register BL
0104	E4	27			IN	AL,27H	Input second number to AL
0106	00	D8			ADD	AL,BL	Add contents of BL to AL and store the sum in AL
0107	E6	30			OUT	30H,AL	Output AL to port 30H
0109	F4				HLT		Halt the computer

Source code

Edit, Assemble, Test, and Debug Cycle

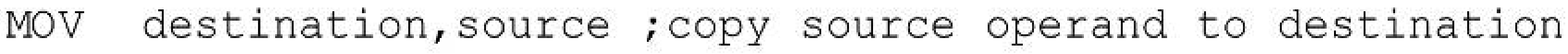
- Using an *editor*, the source code of the program is created. This means selecting the appropriate instruction mnemonics to accomplish the task
- A compiler program which examines the source code file generated by the editor and determines the object code for each instruction in the program, is then run. In assembly language programming, this is called an *assembler* (MASM (Chapter 2 of the textbook, DEBUG: Appendix A of the textbook, etc.,)
- The object code produced by the computer is loaded into the target computer's memory and is then *run.*
- *Debugging:* locating and fixing the source of error
- High-level programming Languages
  - Basic, Pascal, C, C++

assembly language programming

- An Assembly language program consists of a series of lines of Assembly language instructions.
- An Assembly language instruction consists of a mnemonic, optionally followed by one or two *operands*.
  - Operands are the data items being manipulated.
  - Mnemonics are commands to the CPU, telling it what to do with those items.
- Two widely used instructions are *move* & *add.*

MOV instruction

- The MOV instruction copies data from one location to another, using this format:



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

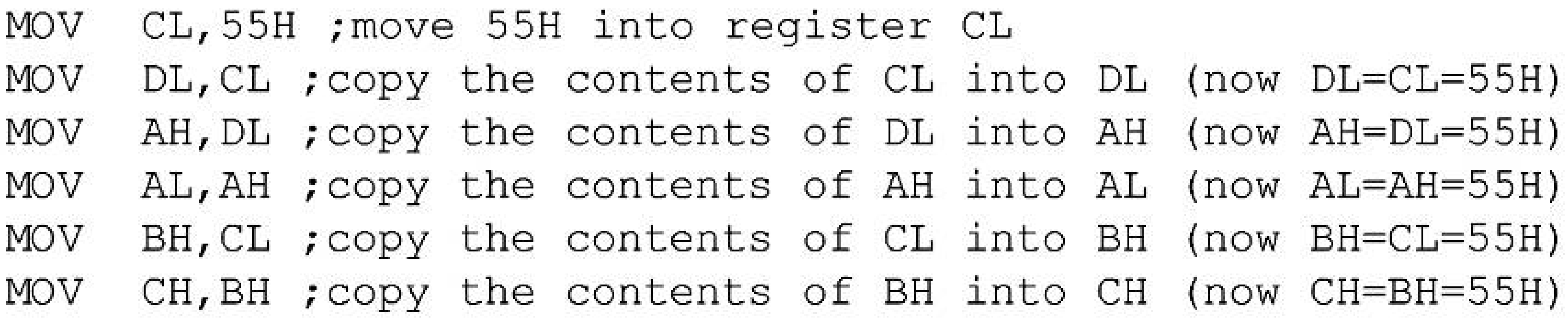
- This instruction tells the CPU to move (in reality, copy) the source operand to the destination operand.
  - For example, the instruction "**MOV DX,CX**" copies the contents of register CX to register DX.
  - After this instruction is executed, register DX will have the **same** value as register CX.

MOV Instruction

- MOV destination,source
  - **8 bit moves**
    - MOV CL,55h
    - MOV DL,CL
    - MOV BH,CL
    - Etc.
  - **16 bit moves**
    - MOV CX,468Fh
    - MOV AX,CX
    - MOV BP,DI
    - Etc.

MOV instruction

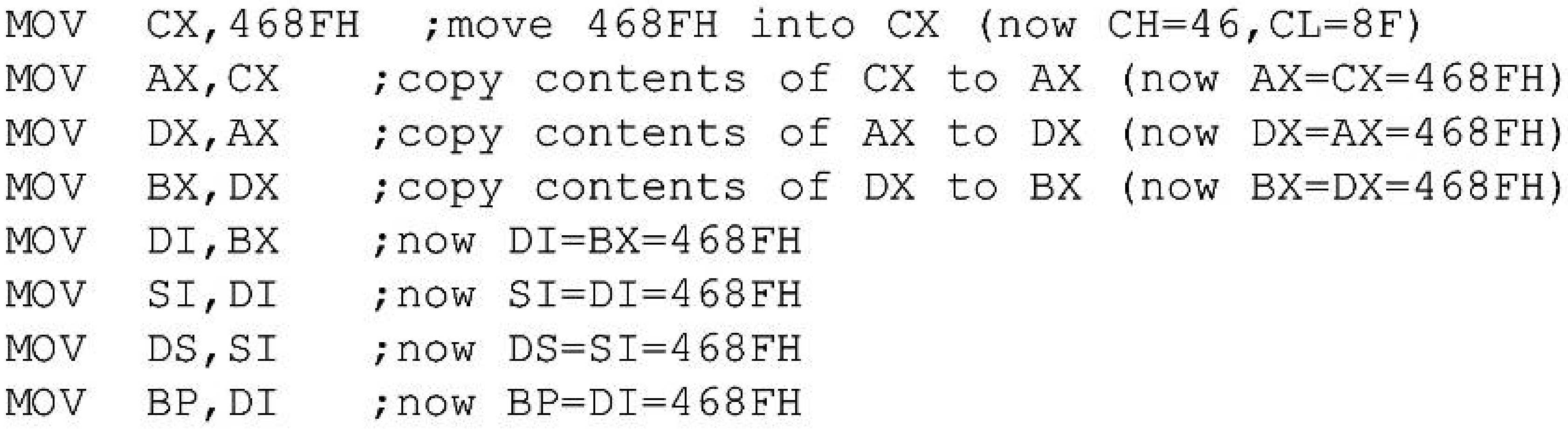
- This program first loads CL with value 55H, then moves this value around to various registers inside the CPU.



```
MOV   CL,55H ;move 55H into register CL
MOV   DL,CL ;copy the contents of CL into DL (now DL=CL=55H)
MOV   AH,DL ;copy the contents of DL into AH (now AH=DL=55H)
MOV   AL,AH ;copy the contents of AH into AL (now AL=AH=55H)
MOV   BH,CL ;copy the contents of CL into BH (now BH=CL=55H)
MOV   CH,BH ;copy the contents of BH into CH (now CH=BH=55H)
```

MOV instruction

- The use of 16-bit registers is shown here:



```
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
```

MOV Instruction

- **Data can be moved among all registers but data cannot be moved directly into the segment registers** (CS,DS,ES,SS).
  - To load as such, first load a value into a non-segment register and then move it to the segment register

```
MOV AX,2345h
MOV DS,AX
```

- **Moving a value that is too large into a register will cause an error**

```
MOV BL,7F2h       ; illegal !
MOV AX,2FE456h  ; illegal !
```

- **If a value less than than FFh is moved into a 16 bit register. The rest of the bits are assumed to be all zeros.**

```
MOV BX,5        ; BX = 0005 with BH  = 00 and BL = 05
```

MOV Instruction

• MOV AX,58FCH	√
• MOV DX,6678H	√
• MOV SI,924BH	√
• MOV BP,2459H	√
• MOV DS,2341H	x
• MOV CX,8876H	√
• MOV CS,3F47H	x
• MOV BH,99H	√

ADD Instruction

- ADD destination,source
- The ADD instruction tells the CPU to add the source and destination operands and put out the results in the destination

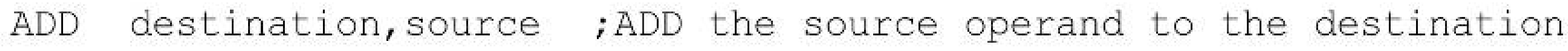
DESTINATION = DESTINATION + SOURCE

```
MOV AL,25H
MOV BL,34h
ADD AL,BL   ; (AL should read 59h once the instruction is executed)
MOV DH,25H
ADD DH,34h   ; (AL should read 59h once the instruction is executed)
```

Immediate operand

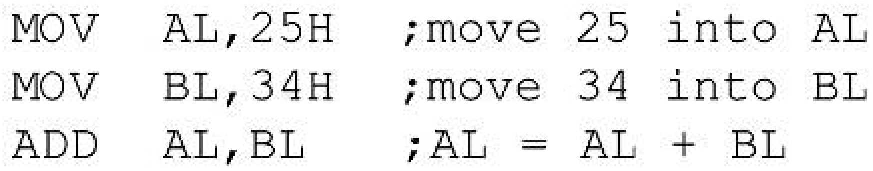
ADD instruction

- The ADD instruction has the following format:



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

- ADD tells the CPU to add the source & 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, then added together:

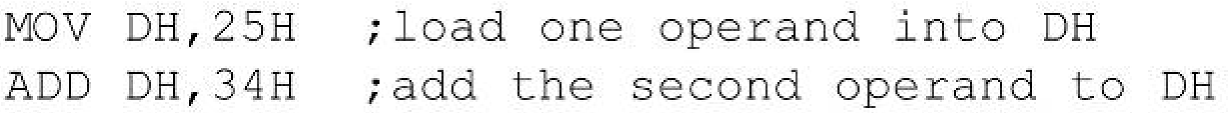


```
MOV   AL,25H   ;move 25 into AL
MOV   BL,34H   ;move 34 into BL
ADD   AL,BL    ;AL = AL + BL
```

  - Executing the program above results in:
    AL = 59H (25H + 34H = 59H) and BL = 34H.
    - The contents of BL do not change.

ADD instruction

- *Is it necessary to move both data items into registers before adding them together?*
  - No, it is not necessary.

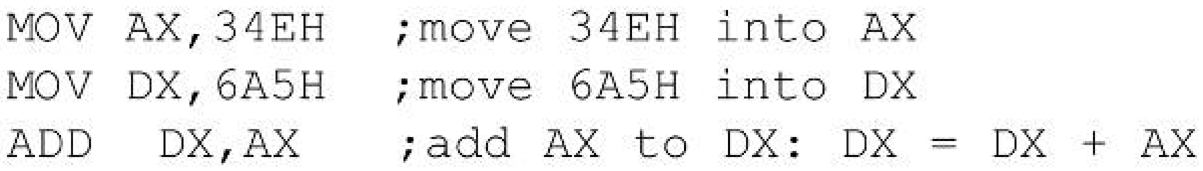


```
MOV DH,25H   ;load one operand into DH
ADD DH,34H   ;add the second operand to DH
```

  - In the case above, while one register contained one value, the second value followed the instruction as
an operand.
    - This is called an immediate operand.

ADD instruction

- An 8-bit register can hold numbers up to FFH.
  - For numbers larger than FFH (255 decimal), a 16-bit register such as AX, BX, CX, or DX must be used.
- The following program can add 34EH & 6A5H:

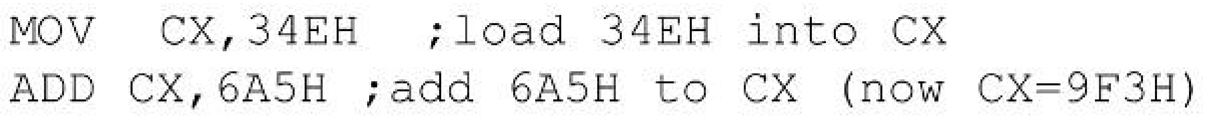


```
MOV AX,34EH    ;move 34EH into AX
MOV DX,6A5H    ;move 6A5H into DX
ADD  DX,AX     ;add AX to DX: DX = DX + AX
```

– Running the program gives DX = 9F3H.
  - (34E + 6A5 = 9F3) and AX = 34E.

ADD instruction

- 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.*

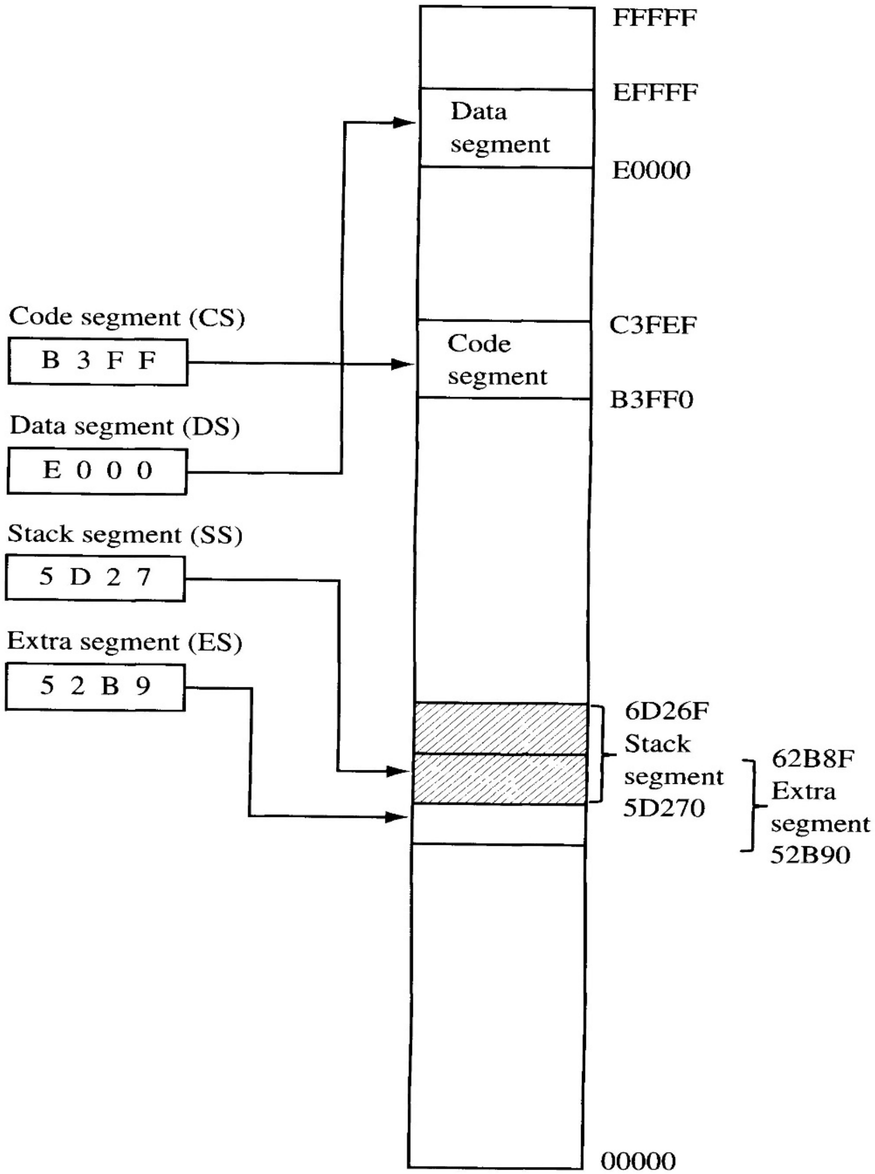
Origin and Definition of a Segment

- A segment is an area of memory that includes up to 64 Kbytes and begins on an address divisible by 16 (such an address ends with an hex digit 0h or 0000b)
  - 8085 could address 64Kbytes 16 address lines
- In the 8085, 64 K is for code, data, and stack
- In the 8086/88, 64 K is assigned to each category
  - Code segment
  - Data segment
  - Stack Segment
  - Extra Segment

Advantages of Segmented Memory

- One program can work on several different sets of data. This is done by reloading register DS to a new value.
- Programs that reference logical addresses can be loaded and run anywhere in the memory: relocatable
- Segmented memory introduces extra complexity in both hardware in that memory addresses require two registers.
- They also require complexity in software in that programs are limited to the segment size
- Programs greater than 64 KB can be run on 8086 but the software needed is more complex as it must switch to a new segment.
- Protection among segments is provided (when available).

Segment Registers

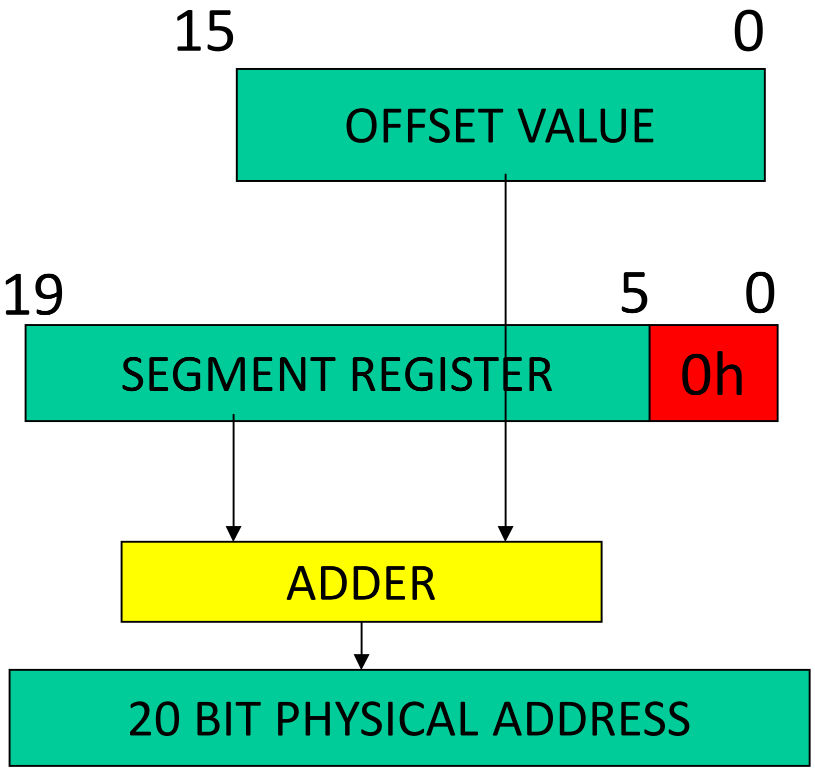


logical address and physical address

- In literature concerning 8086, there are three types of addresses mentioned frequently:
  - **The physical address** - the 20-bit address actually on the address pins of the 8086 processor, decoded by the memory interfacing circuitry.
    - This address can have a range of 00000H to FFFFFH.
    - An actual physical location in RAM or ROM within the 1 mb memory range.
  - **The offset address** - a location in a 64K-byte segment range, which can can range from 0000H to FFFFH.
  - **The logical address** - which consists of a segment value and an offset address.

Logical and Physical Addresses

- Addresses within a segment can range from address 0 to address FFFFh. This corresponds to the 64Kbyte length of the segment called an *offset*
- An address within a segment is called the *logical address*
- *Ex.* Logical address 0005h in the code segment actually corresponds to B3FF0h + 5 = B3FF5h.



Example 1:
Segment base value: 1234h
Offset: 0022h

$$\begin{array}{r} 12340h \\ +\ 0022h \\ \hline \end{array}$$

12362h is the physical 20 bit address

Two different logical addresses may correspond to the same physical address.

D470h in ES	2D90h in SS
ES:D470h	SS:2D90h

Example

- If DS=7FA2H and the offset is 438EH

a) Calculate the physical address

7FA20 + 438E = 83DAE

b) calculate the lower range

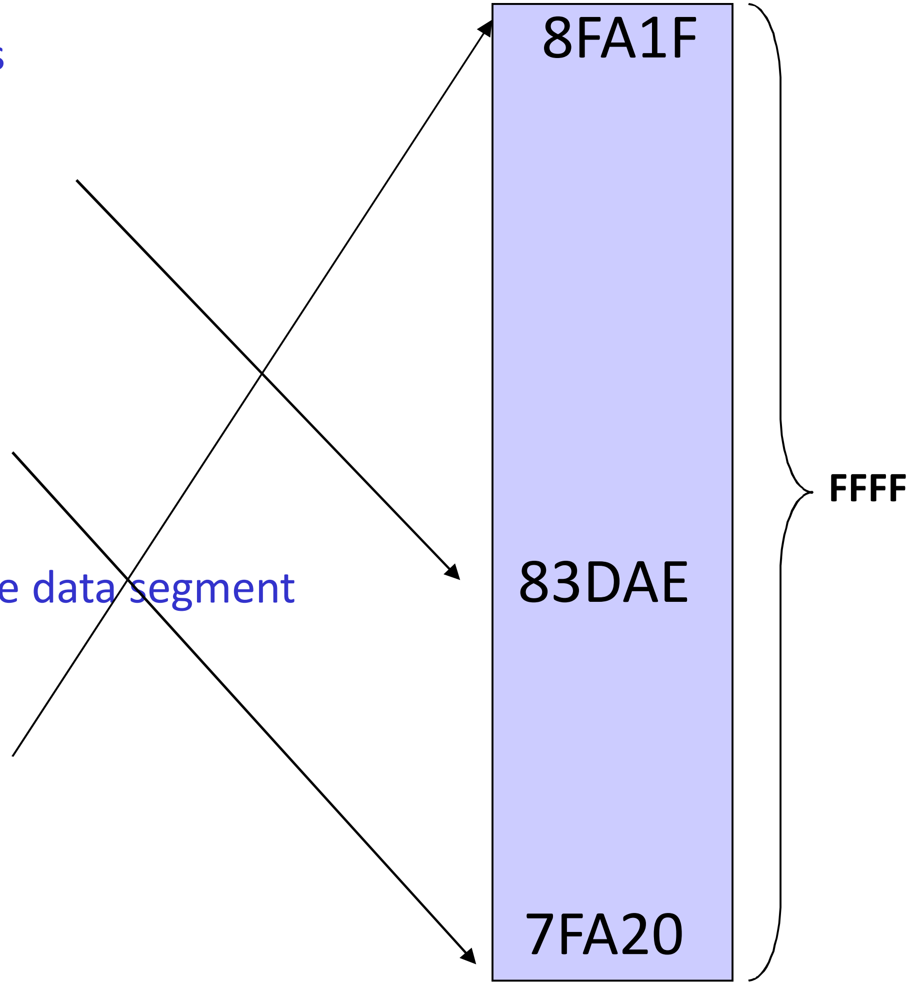
7FA20 + 0000 = 7FA20

c) Calculate the upper range of the data segment

7FA20 + FFFF = 8FA1F

d) Show the logical Address

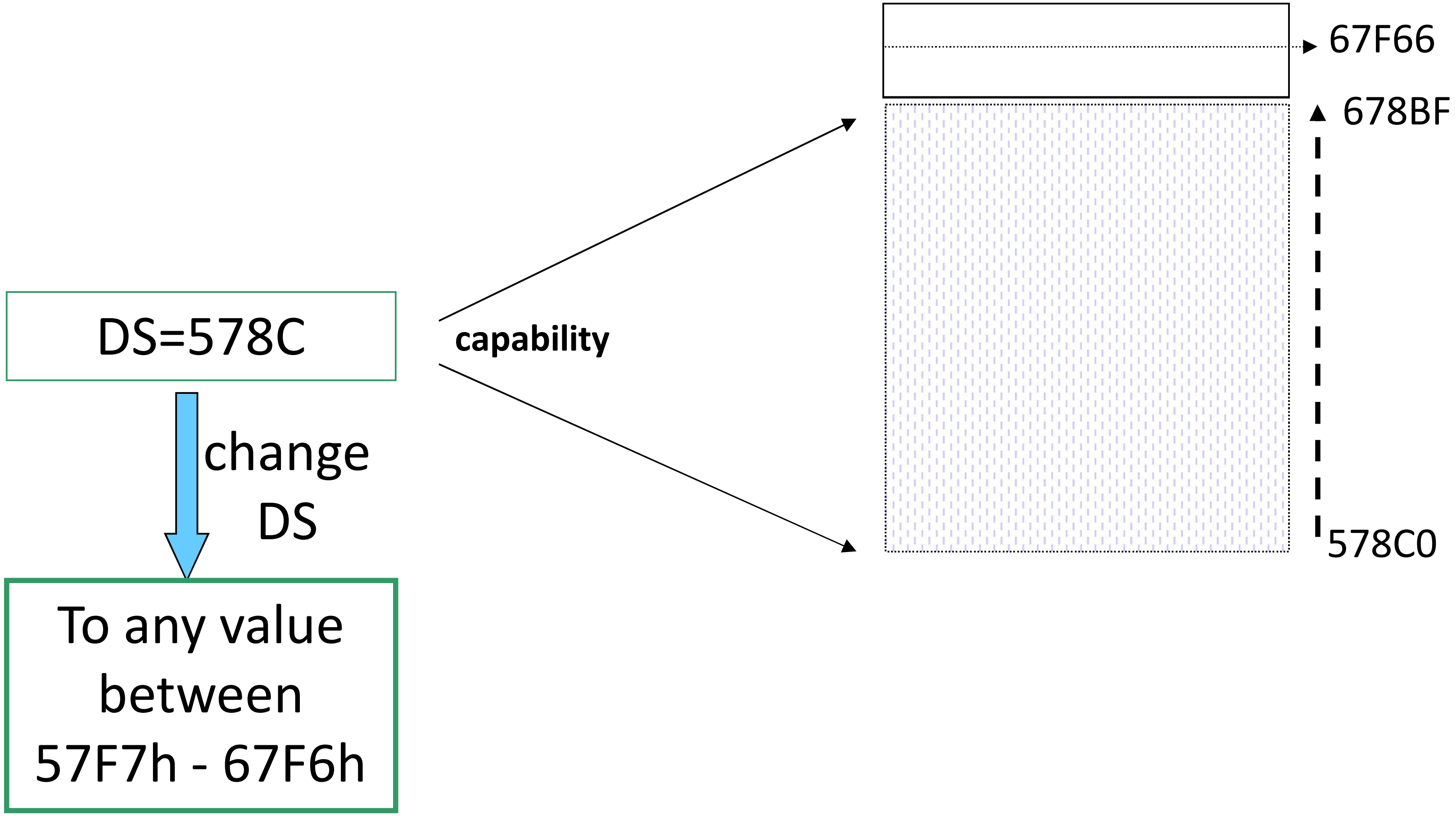
7FA2:438E



Example

Question:

Assume DS=578C. To access a Data in 67F66 what should we do?



Another Example

- What would be the offset required to map to physical address location 002C3H if the contents of the corresponding segment register is 002AH?

- Solution: the offset value can be obtained by shifting the contents of the segment register left by four bit positions and then subtracting from the physical address. Shifting gives:

  002A0H

  Now subtracting, we get the value of the offset:

  002C3H-002A0H=0023H

Code Segment

- To execute a program, the 8086 fetches the instructions (opcodes and operands) from the code segment
- The logical address is in the form CS:IP

- Example: If CS = 24F6h and IP = 634Ah, show
  - The logical address
  - The offset address

  and calculate

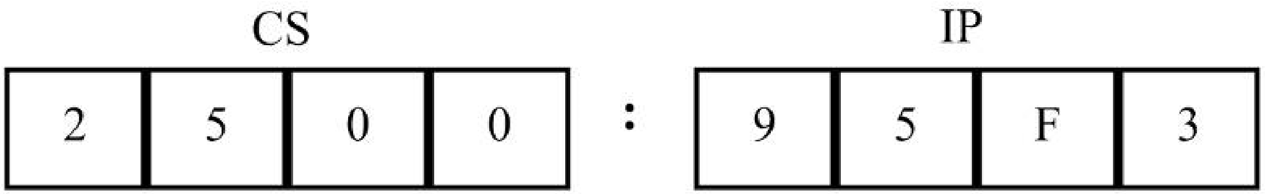
  - The physical address
  - The lower range
  - The upper range

code segment

- To execute a program, 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, 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.

code segment

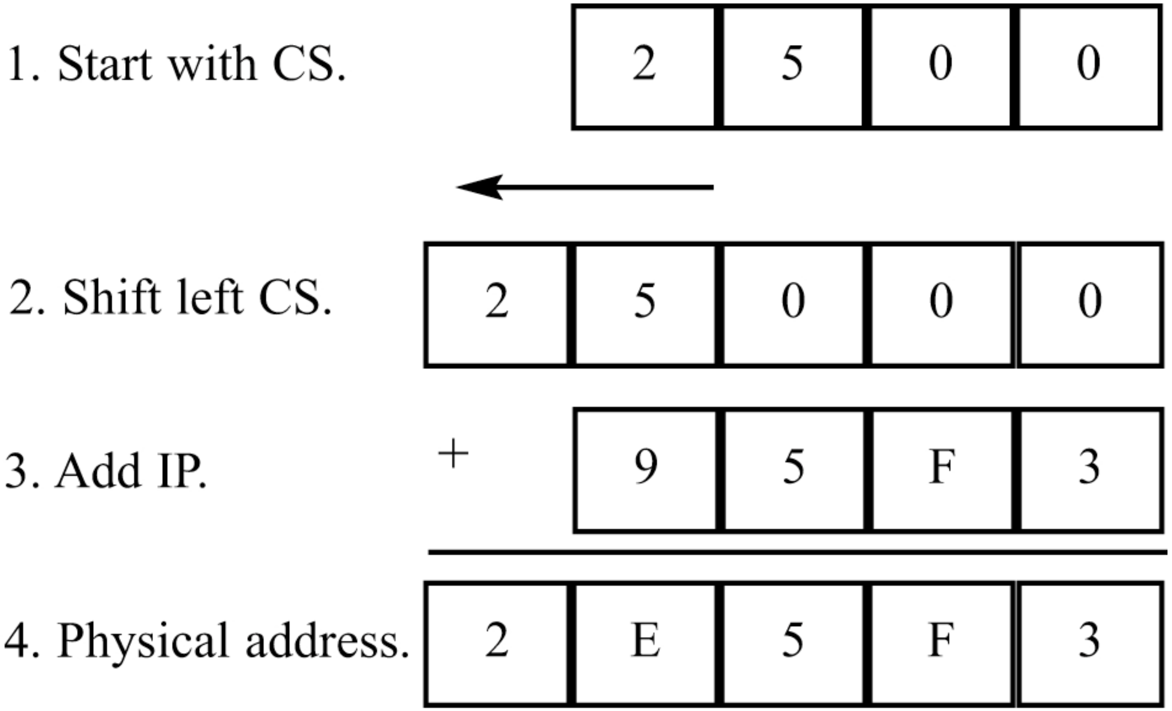
- Assume values in CS & IP as shown in the diagram:



  - The offset address contained in IP, is 95F3H.
  - The logical address is **CS:IP**, or **2500:95F3H.**
  - The physical address will be 25000 + 95F3 = 2E5F3H

code segment

- Calculate the physical address of an instruction:

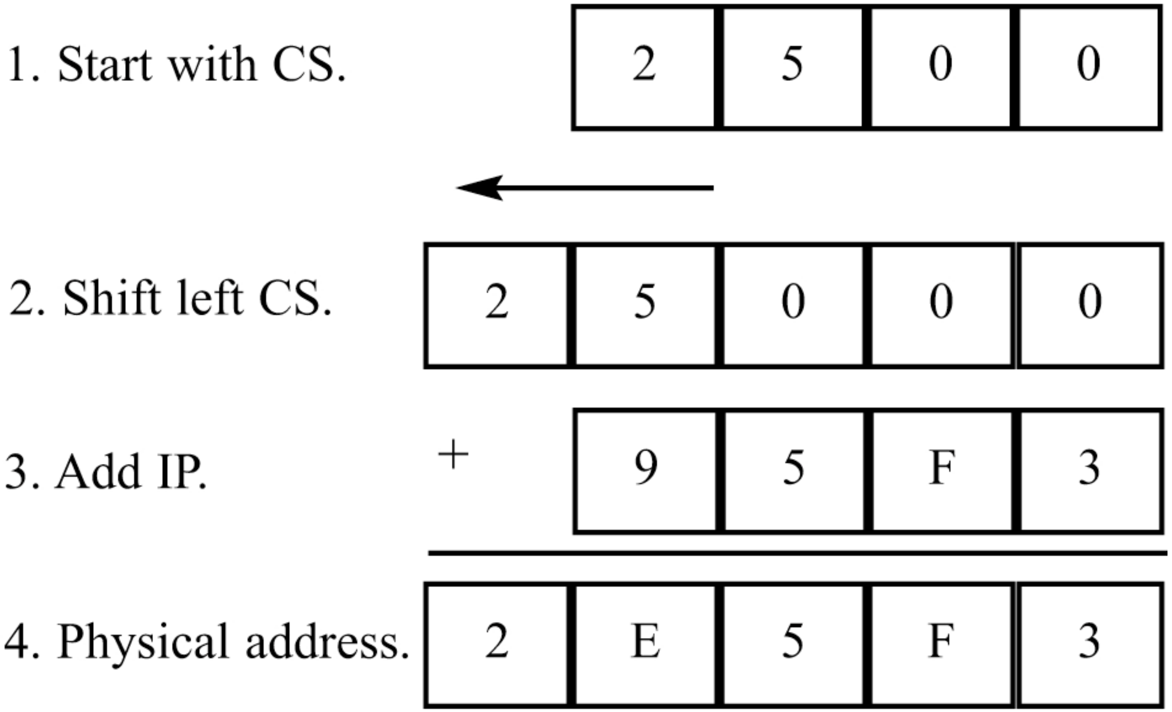


1. Start with CS.		2	5	0	0
2. Shift left CS.	2	5	0	0	0
3. Add IP.	+	9	5	F	3
4. Physical address.	2	E	5	F	3

  – The microprocessor will retrieve the instruction from memory locations starting at 2E5F3.

code segment

- Calculate the physical address of an instruction:



1. Start with CS. 2 5 0 0

2. Shift left CS. 2 5 0 0 0

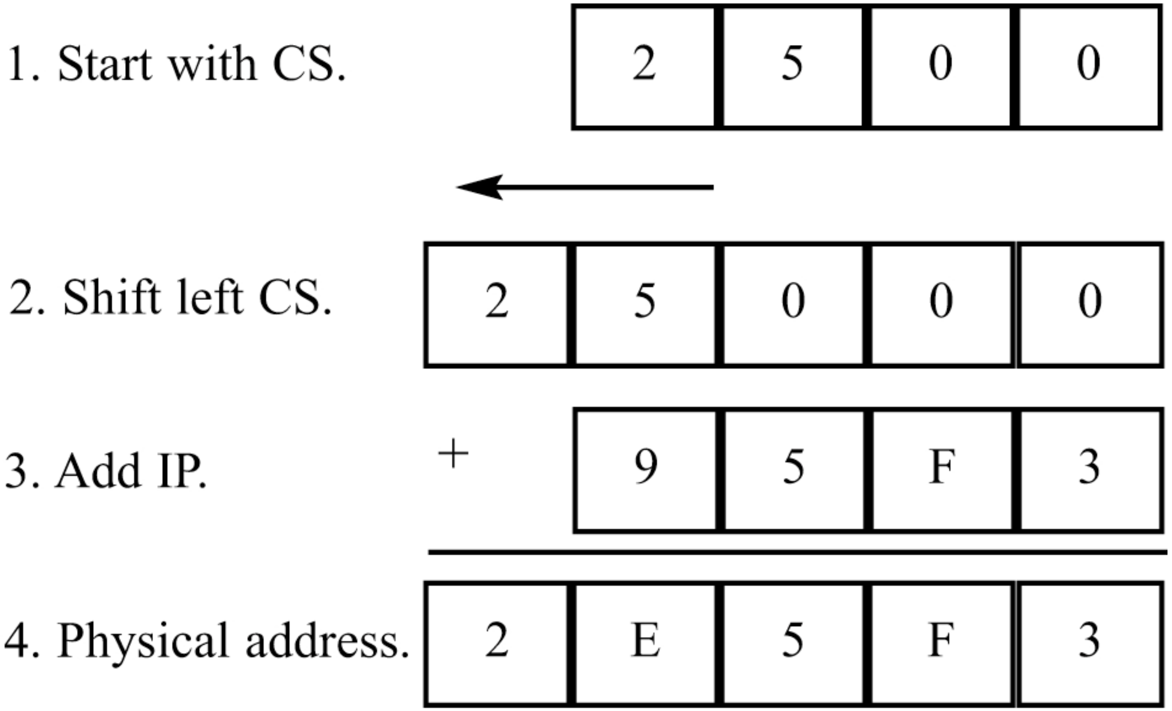
3. Add IP. + 9 5 F 3

4. Physical address. 2 E 5 F 3

  - Since IP can have a minimum value of 0000H and a maximum of FFFFH, the logical address range in this example is 2500:0000 to 2500:FFFF.

code segment

- Calculate the physical address of an instruction:



1. Start with CS. 2 5 0 0

2. Shift left CS. 2 5 0 0 0

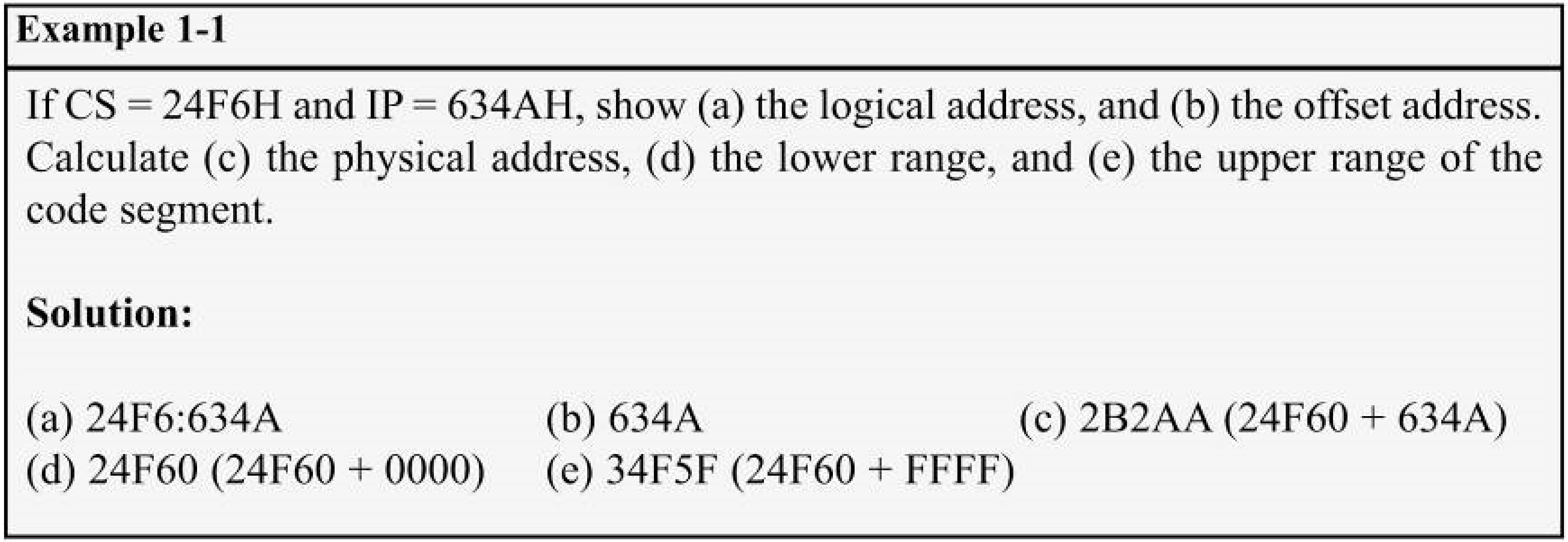
3. Add IP. + 9 5 F 3

4. Physical address. 2 E 5 F 3

  - This means that the lowest memory location of the code segment above will be 25000H (25000 + 0000) and the highest memory location will be 34FFFH (25000 + FFFF).

code segment

- *What happens if the desired instructions are located beyond these two limits?*
  - The value of CS must be changed to access those instructions.



Example 1-1

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

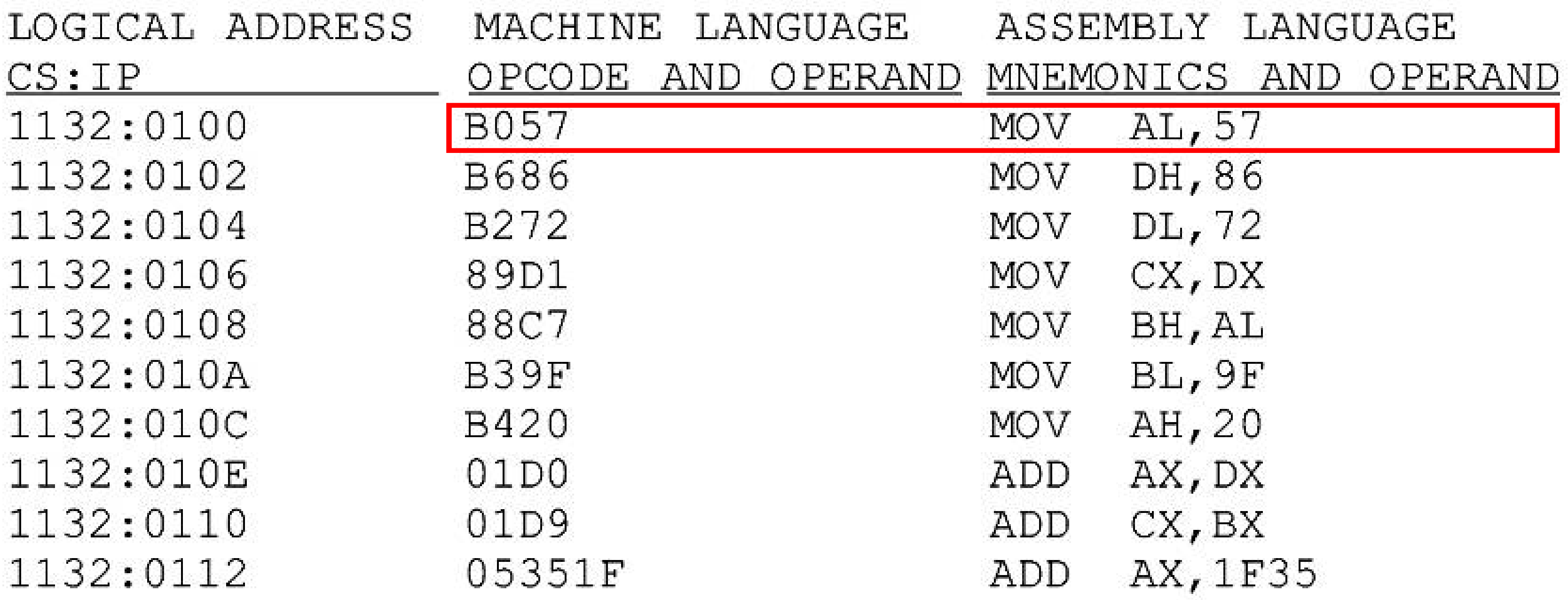
Solution:

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

code segment logical/physical address

- In the next code segment, CS and IP hold the logical address of the instructions to be executed.
  - The following Assembly language instructions have been assembled (translated into machine code) and stored in memory.
  - The three columns show the logical address of **CS:IP**, the machine code stored at that address, and the corresponding Assembly language code.
  - The physical address is put on the address bus by the CPU to be decoded by the memory circuitry.

code segment logical/physical address



LOGICAL ADDRESS	MACHINE LANGUAGE	ASSEMBLY LANGUAGE
CS:IP	OPCODE AND OPERAND	MNEMONICS AND OPERAND
1132:0100	B057	MOV AL,57
1132:0102	B686	MOV DH,86
1132:0104	B272	MOV DL,72
1132:0106	89D1	MOV CX,DX
1132:0108	88C7	MOV BH,AL
1132:010A	B39F	MOV BL,9F
1132:010C	B420	MOV AH,20
1132:010E	01D0	ADD AX,DX
1132:0110	01D9	ADD CX,BX
1132:0112	05351F	ADD AX,1F35

Instruction "**MOV AL,57**" has a machine code of B057.

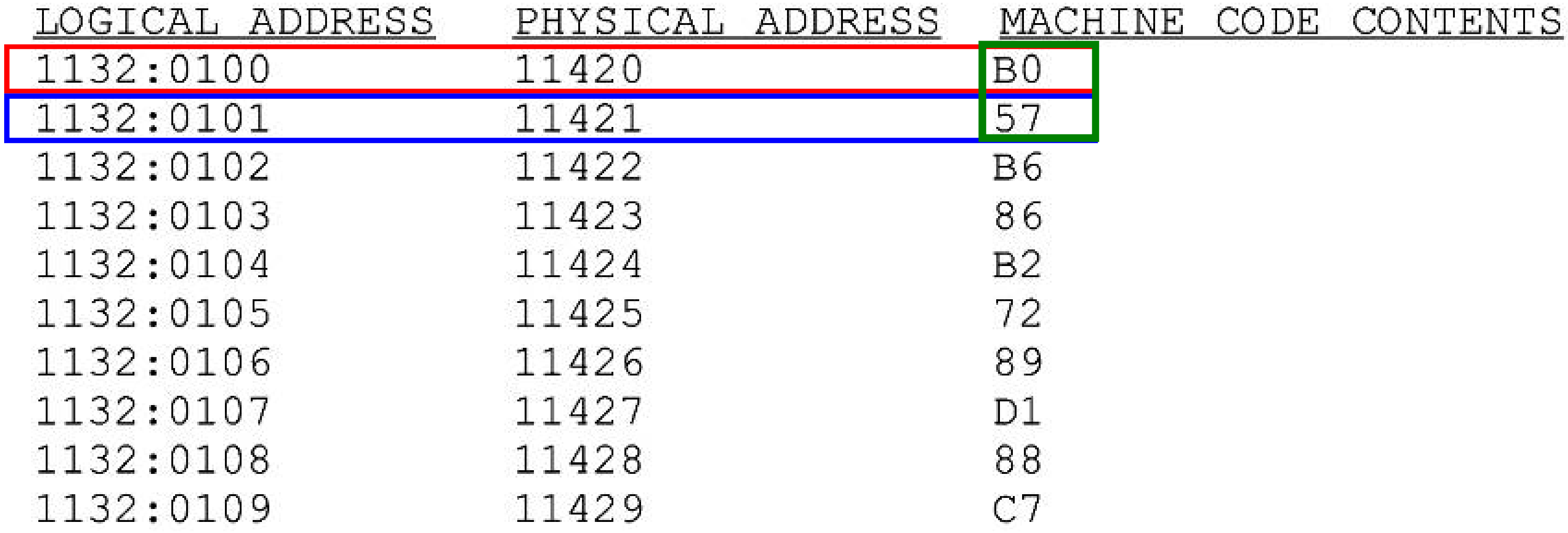
B0 is the opcode and 57 is the operand.

Logical Address vs Physical Address in the CS

CS:IP	Machine Language	Mnemonics
1132:0100	B057	MOV AL,57h
1132:0102	B686	MOV DH,86h
1132:0104	B272	MOV DL,72h
1132:0106	89D1	MOV CX,DX
1132:0108	88C7	MOV BH,AL
1132:010A	B39F	MOV BL,9F
1132:010C	B420	MOV AH,20h
1132:010E	01D0	ADD AX,DX
1132:0110	01D9	ADD CX,BX
1132:0112	05351F	ADD AX, 1F35h

- Show how the code resides physically in the memory

code segment logical/physical address



LOGICAL ADDRESS	PHYSICAL ADDRESS	MACHINE CODE CONTENTS
1132:0100	11420	B0
1132:0101	11421	57
1132:0102	11422	B6
1132:0103	11423	86
1132:0104	11424	B2
1132:0105	11425	72
1132:0106	11426	89
1132:0107	11427	D1
1132:0108	11428	88
1132:0109	11429	C7

Instruction "**MOV AL,57**" has a machine code of B057.

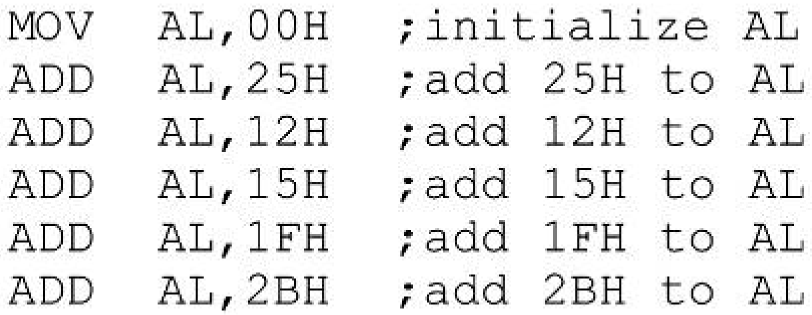
B0 is the opcode and 57 is the operand.

The byte at address 1132:0100 contains B0, the opcode for moving a value into register AL.

Address 1132:0101 contains the operand to be moved to AL.

data segment

- Assume a program to add 5 bytes of data, such as 25H, 12H, 15H, 1FH, and 2BH.
  - One way to add them is as follows:



```
MOV   AL,00H   ;initialize AL
ADD   AL,25H   ;add 25H to AL
ADD   AL,12H   ;add 12H to AL
ADD   AL,15H   ;add 15H to AL
ADD   AL,1FH   ;add 1FH to AL
ADD   AL,2BH   ;add 2BH to AL
```

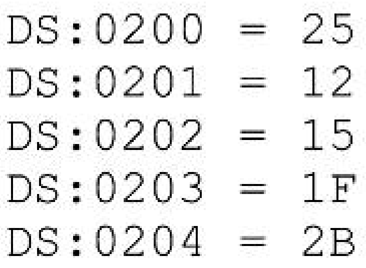
- In the program above, the data & code are mixed together in the instructions.
  - If the data changes, the code must be searched for every place it is included, and the data retyped
  - From this arose the idea of an area of memory strictly for data

data segment

- In x86 microprocessors, the area of memory set aside for data is called the *data segment.*
  - The data segment uses register DS and an offset value.
  - DEBUG assumes that all numbers are in hex.
    - No "H" suffix is required.
  - MASM assumes that they are in decimal.
    - The "H" *must* be included for hex data.
- The next program demonstrates how data can be stored in the data segment and the program rewritten so that it can be used for any set of data.

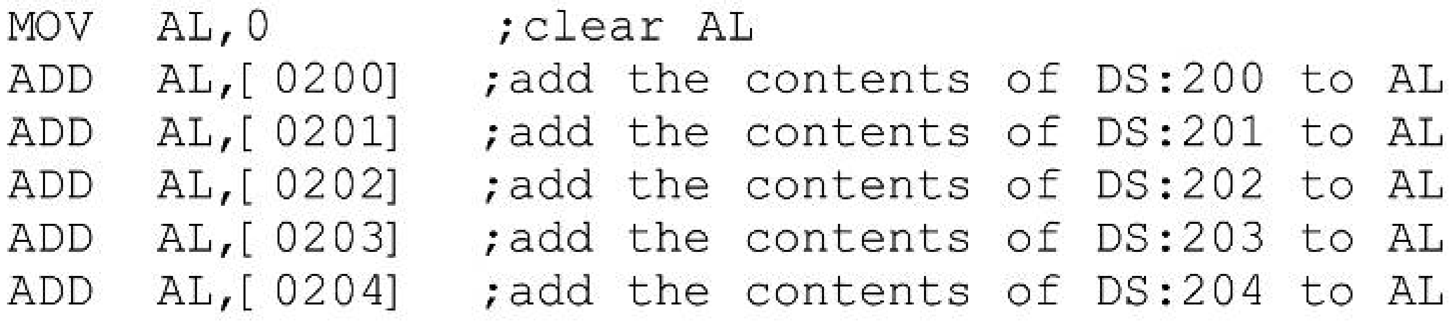
data segment

- Assume data segment offset begins at 200H.
  - The data is placed in memory locations:



```
DS:0200 = 25
DS:0201 = 12
DS:0202 = 15
DS:0203 = 1F
DS:0204 = 2B
```

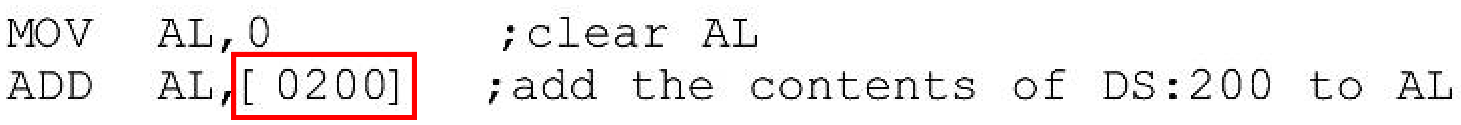
  - The program can be rewritten as follows:



```
MOV   AL,0          ;clear AL
ADD   AL,[ 0200]   ;add the contents of DS:200 to AL
ADD   AL,[ 0201]   ;add the contents of DS:201 to AL
ADD   AL,[ 0202]   ;add the contents of DS:202 to AL
ADD   AL,[ 0203]   ;add the contents of DS:203 to AL
ADD   AL,[ 0204]   ;add the contents of DS:204 to AL
```

data segment

- The offset address is enclosed in **brackets**, which indicate that the operand represents the address of the data and not the data itself.



```
MOV   AL,0          ;clear AL
ADD   AL,[ 0200]   ;add the contents of DS:200 to AL
```

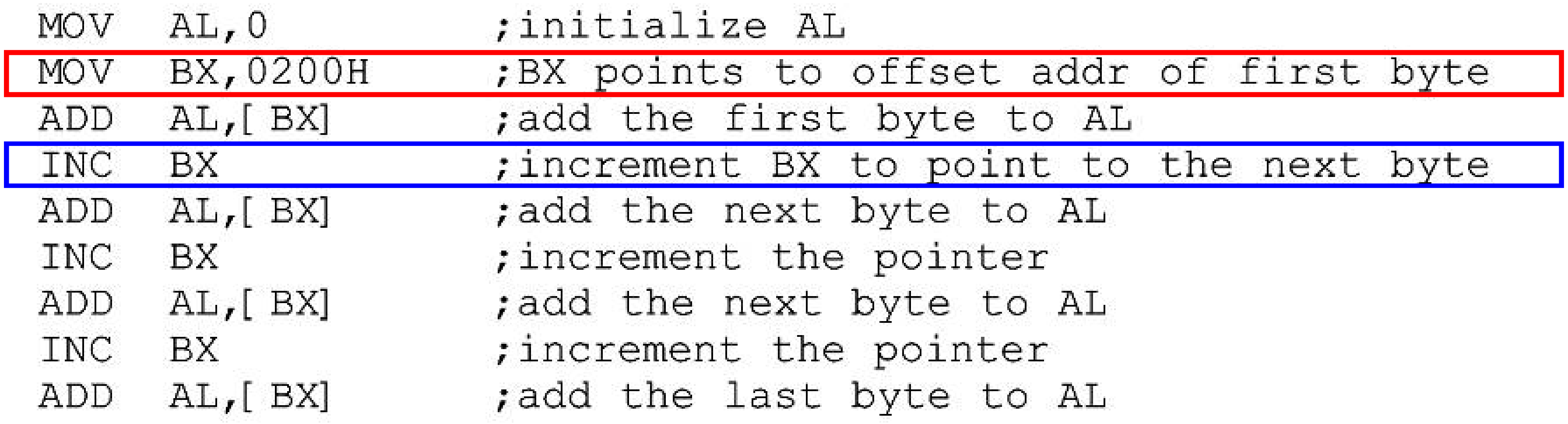
- If the brackets were not included, as in "**MOV AL,0200**", the CPU would attempt to move 200 into AL instead of the contents of offset address 200. decimal.
  - This program will run with any set of data.
  - Changing the data has no effect on the code.

data segment

- If the data had to be stored at a different offset address the program would have to be rewritten
  - A way to solve this problem is to use a register to hold the offset address, and before each ADD, increment the register to access the next byte.
- 8088/86 allows only the use of registers BX, SI, and DI as offset registers for the data segment
  - The term *pointer* is often used for a register holding an offset address.

data segment

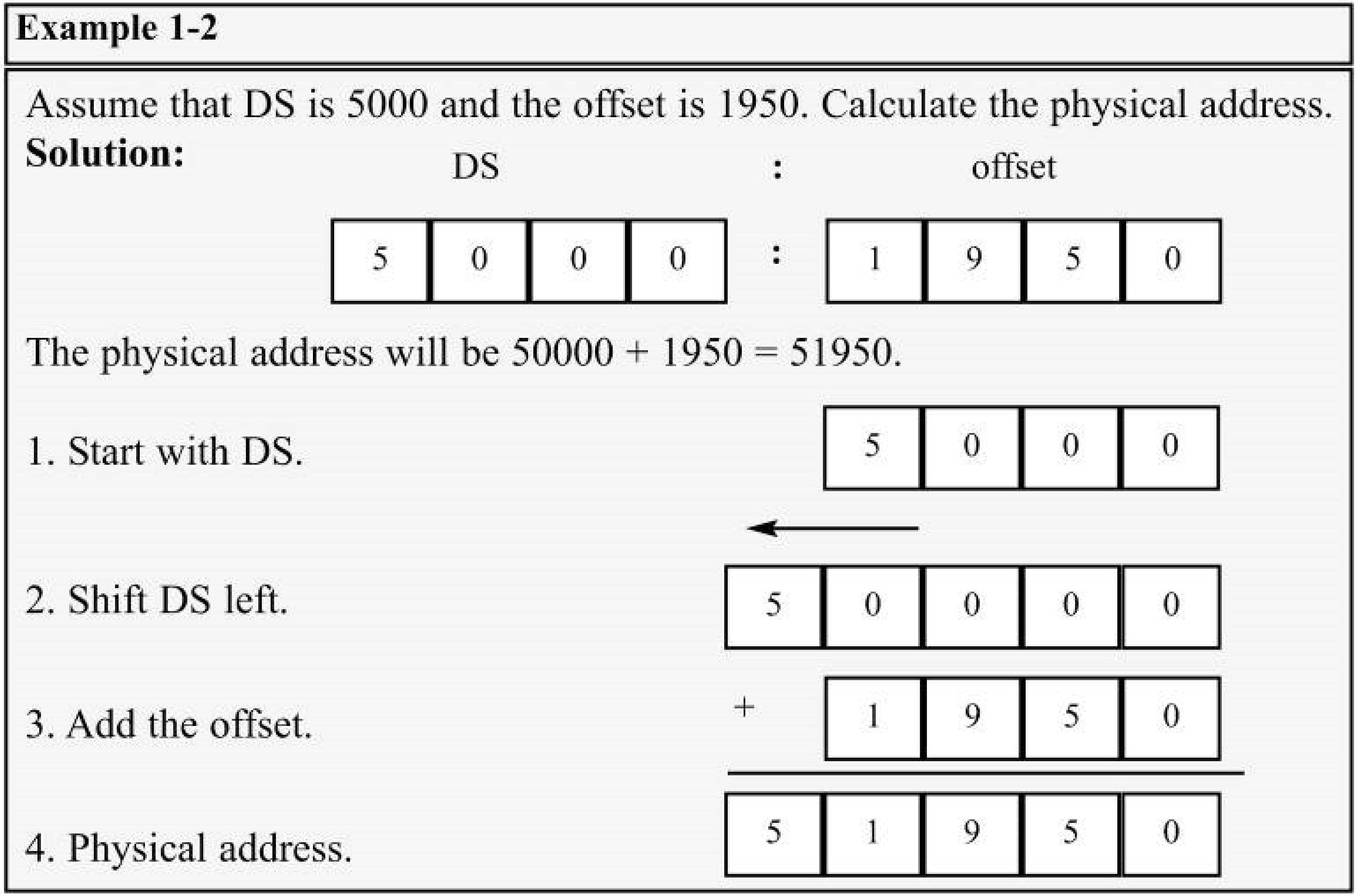
- In the following example, **BX** is used as a pointer:



```
MOV   AL,0             ;initialize AL
MOV   BX,0200H         ;BX points to offset addr of first byte
ADD   AL,[ BX]         ;add the first byte to AL
INC   BX               ;increment BX to point to the next byte
ADD   AL,[ BX]         ;add the next byte to AL
INC   BX               ;increment the pointer
ADD   AL,[ BX]         ;add the next byte to AL
INC   BX               ;increment the pointer
ADD   AL,[ BX]         ;add the last byte to AL
```

- The **INC** instruction adds 1 to (increments) its operand.
  - "**INC BX**" achieves the same result as "**ADD BX,1**“
  - If the offset address where data is located is changed, only one instruction will need to be modified.

data segment logical/physical address



Example 1-2

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

Solution:

DS : offset

5 0 0 0 : 1 9 5 0

The physical address will be 50000 + 1950 = 51950.

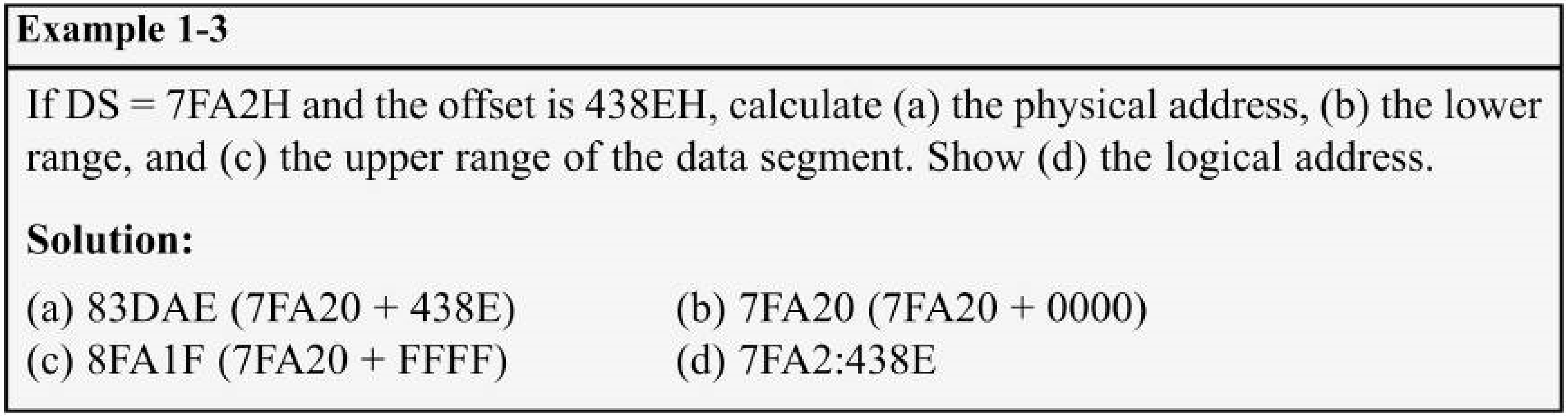
1. Start with DS. 5 0 0 0

2. Shift DS left. 5 0 0 0 0

3. Add the offset. + 1 9 5 0

4. Physical address. 5 1 9 5 0

data segment logical/physical address

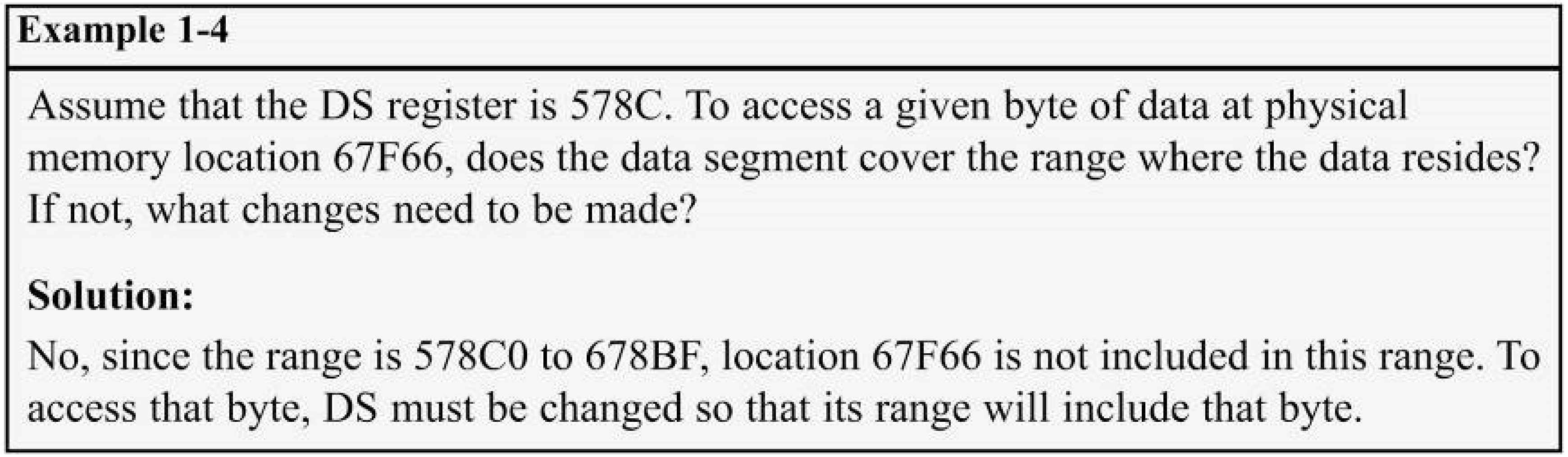


Example 1-3

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

Solution:

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



Example 1-4

Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data resides? If not, what changes need to be made?

Solution:

No, since the range is 578C0 to 678BF, location 67F66 is not included in this range. To access that byte, DS must be changed so that its range will include that byte.

extra segment (ES)

- ES is a segment register used as an extra data segment.
  - In many normal programs this segment is not used.
  - Use is essential for string operations.

16 bit Segment Register Assignments

Type of Memory Reference	Default Segment	Alternate Segment	Offset
Instruction Fetch	CS	none	IP
Stack Operations	SS	none	SP,BP
General Data	DS	CS,ES,SS	BX, address
String Source	DS	CS,ES,SS	SI, DI, address
String Destination	ES	None	DI

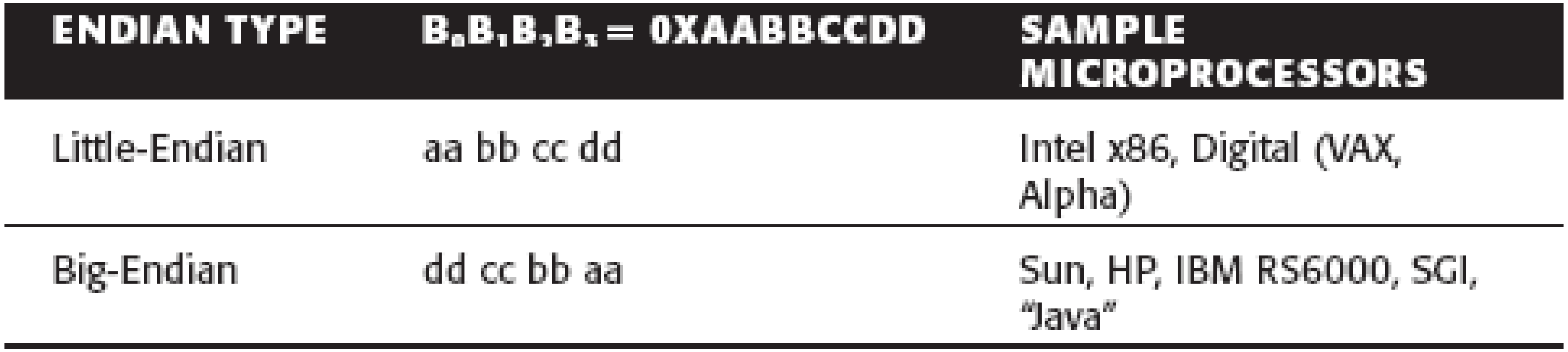
Brey

Little Endian Convention

"Little Endian" means that the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.)
Intel uses Little Endian Convention.
For example, a 4 byte LongInt

Byte3 | Byte2 | Byte1 |Byte0 will be arranged in memory as follows:
Base Address+0 Byte0
Base Address+1 Byte1
Base Address+2 Byte2
Base Address+3 Byte3

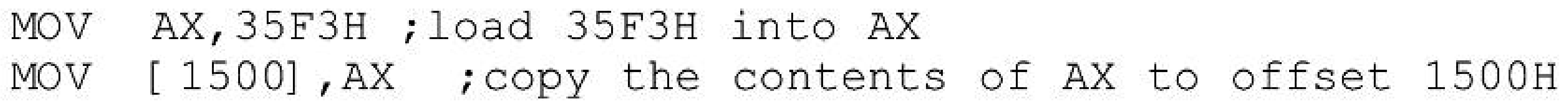
- **Adobe Photoshop** -- Big Endian
- **BMP (Windows and OS/2 Bitmaps)** – little Endian
- **GIF** -- Little Endian
- **IMG (GEM Raster)** -- Big Endian
- **JPEG** -- Big Endian



ENDIAN TYPE	$B_0B_1B_2B_3$ = 0XAABBCCDD	SAMPLE MICROPROCESSORS
Little-Endian	aa bb cc dd	Intel x86, Digital (VAX, Alpha)
Big-Endian	dd cc bb aa	Sun, HP, IBM RS6000, SGI, "Java"

little endian convention

- Previous examples used 8-bit or 1-byte data.
  - *What happens when 16-bit data is used?*

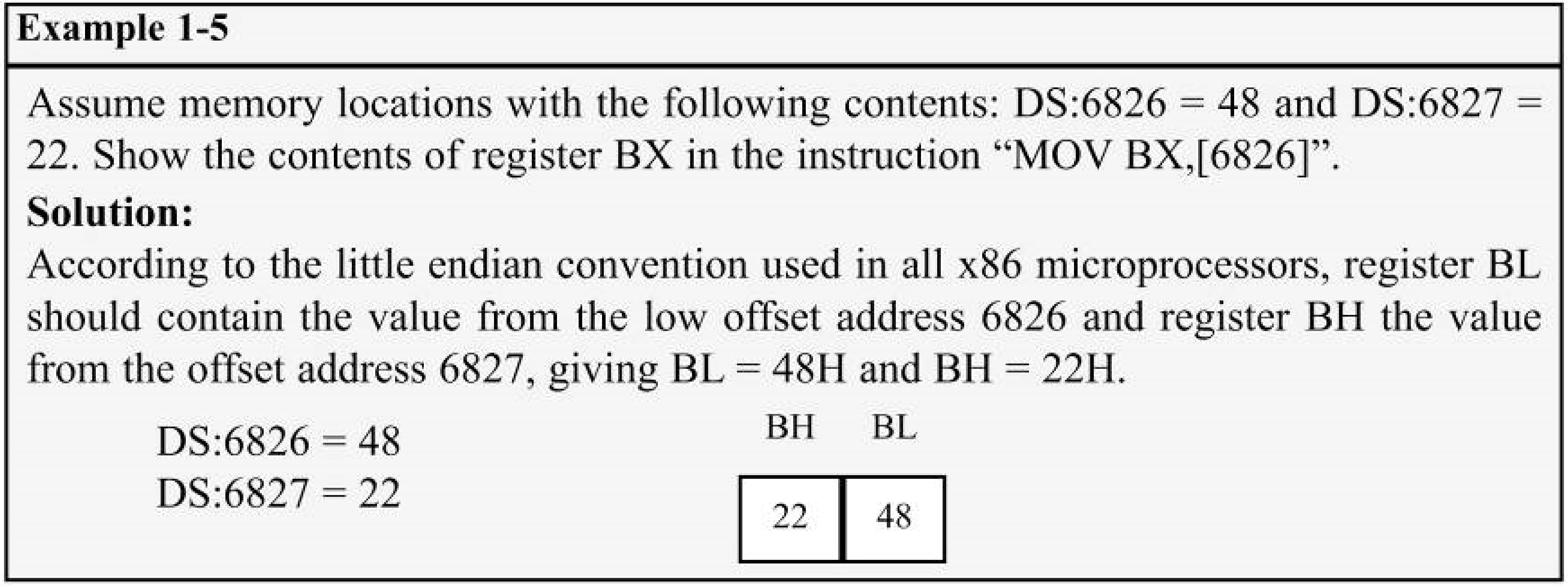


```
MOV   AX,35F3H ;load 35F3H into AX
MOV  [1500],AX   ;copy the contents of AX to offset 1500H
```

- The low byte goes to the low memory location and the high byte goes to the high memory address.
  - Memory location DS:1500 contains F3H.
  - Memory location DS:1501 contains 35H.
    - (DS:1500 = F3 DS:1501 = 35)
  - This convention is called *little endian* vs *big endian*.
    - From a Gulliver’s Travels story about how an egg should be opened—from the little end, or the big end.

little endian convention

- In the big endian method, the high byte goes to the low address.
  - In the little endian method, the high byte goes to the high address and the low byte to the low address.



Example 1-5

Assume memory locations with the following contents: DS:6826 = 48 and DS:6827 = 22. Show the contents of register BX in the instruction "MOV BX,[6826]".

Solution:

According to the little endian convention used in all x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH the value from the offset address 6827, giving BL = 48H and BH = 22H.

DS:6826 = 48
DS:6827 = 22

BH	BL
22	48

what is a stack? why is it needed?

- The stack is a section of read/write memory (RAM) used by the CPU to store information temporarily.
  - The CPU needs this storage area since there are only a limited number of registers.
    - There must be some place for the CPU to store information safely and temporarily.
- The main disadvantage of the stack is access time.
  - Since the stack is in RAM, it takes much longer to access compared to the access time of registers.
- Some very powerful (expensive) computers do not have a stack.
  - The CPU has a large number of registers to work with.

how stacks are accessed

- The stack is a section of RAM, so there must be registers inside the CPU to point to it.
  - The SS (*stack segment*) register.
  - The SP (*stack pointer*) register.
    - These registers must be loaded before any instructions accessing the stack are used.
- Every register inside the x86 can be stored in the stack, and brought back into the CPU from the
  stack memory, except segment registers and SP.
  - Storing a CPU register in the stack is called a *push*.
  - Loading the contents of the stack into the CPU register is called a *pop.*

how stacks are accessed

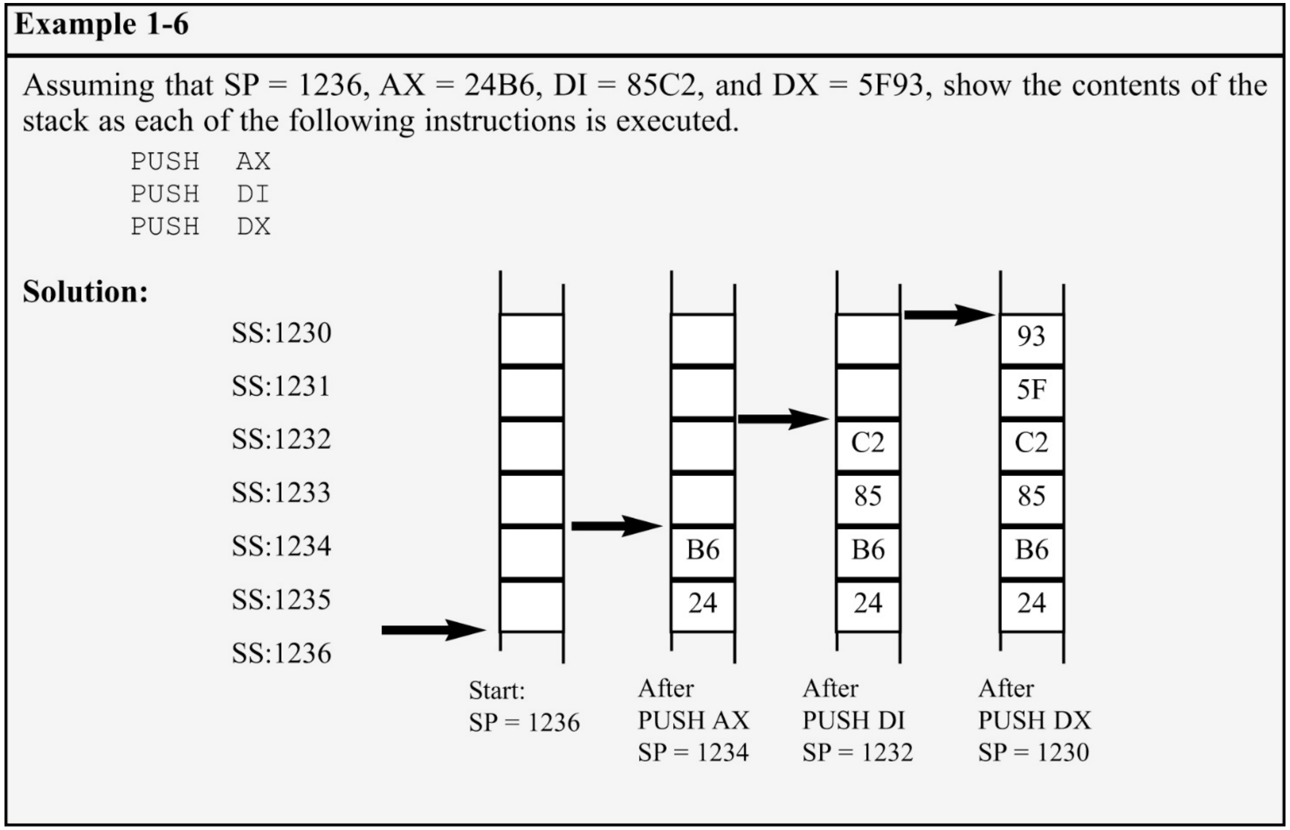
- The x86 stack pointer register (SP) points at the current memory location used as the top of the stack.
  - As data is *pushed* ***onto*** the stack it is ***decremented****.*
  - As data is *popped* ***off*** the stack into the CPU, it is ***incremented***.
- When an instruction pushes or pops a general-purpose register, it must be the *entire* 16-bit register.
  - One must code "**PUSH AX**".
    - There are no instructions such as "**PUSH AL**" or "**PUSH AH**".

how stacks are accessed

- The SP is decremented after the push is to make sure the stack is growing *downward* from upper addresses to lower addresses.
  - The opposite of the IP. (instruction pointer)
- To ensure the code section & stack section of the program never write over each other, they are
  located at opposite ends of the RAM set aside
  for the program.
  - They grow toward each other but must not meet.
    - If they meet, the program will crash.

pushing onto the stack

- As each PUSH is executed, the register contents are saved on the stack and SP is decremented by 2.



Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
        PUSH   AX
        PUSH   DI
        PUSH   DX
```

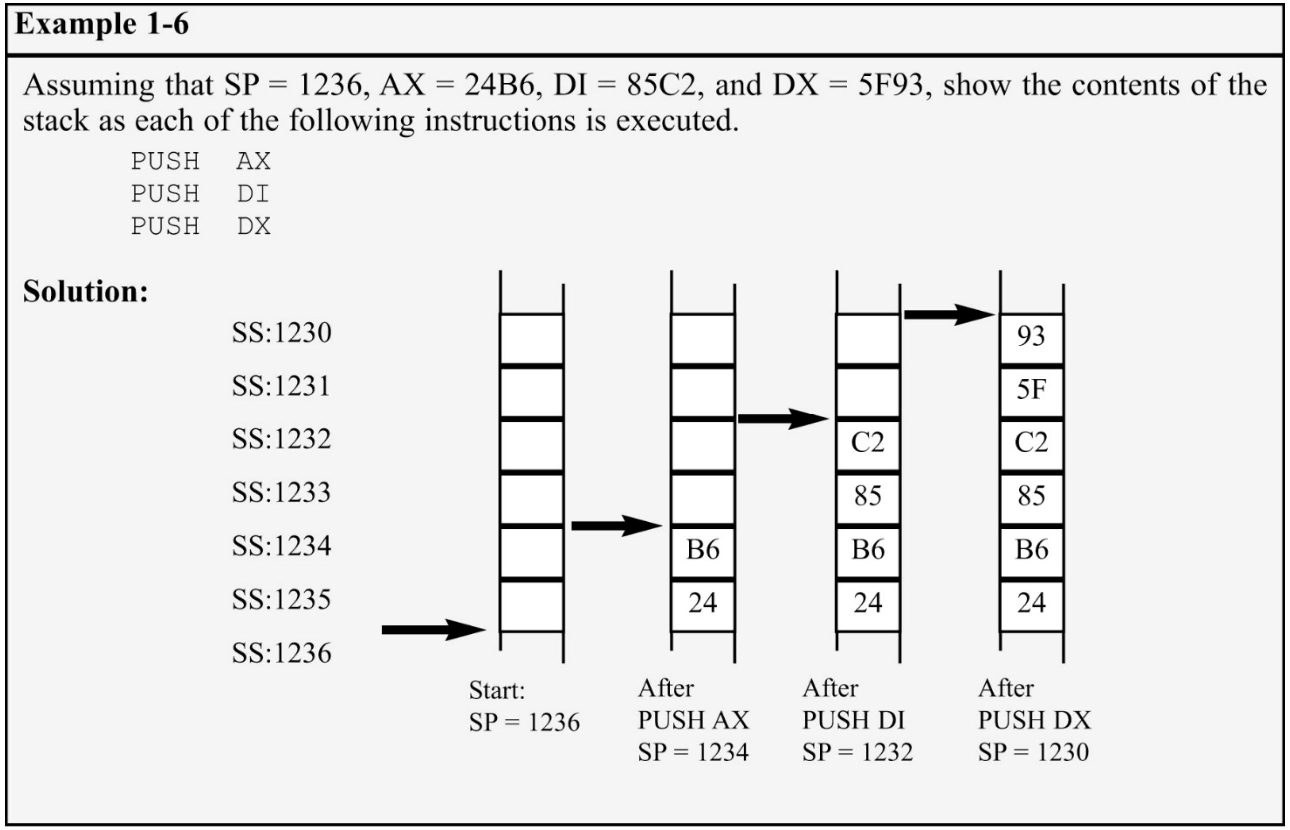
Solution:

	Start: SP = 1236	After PUSH AX SP = 1234	After PUSH DI SP = 1232	After PUSH DX SP = 1230
SS:1230				93
SS:1231				5F
SS:1232			C2	C2
SS:1233			85	85
SS:1234		B6	B6	B6
SS:1235		24	24	24
SS:1236				

pushing onto the stack

- For every byte of data saved on the stack, SP is decremented once.

Since the push is saving the contents of a 16-bit register, it decrements *twice.*



Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
PUSH   AX
PUSH   DI
PUSH   DX
```

Solution:

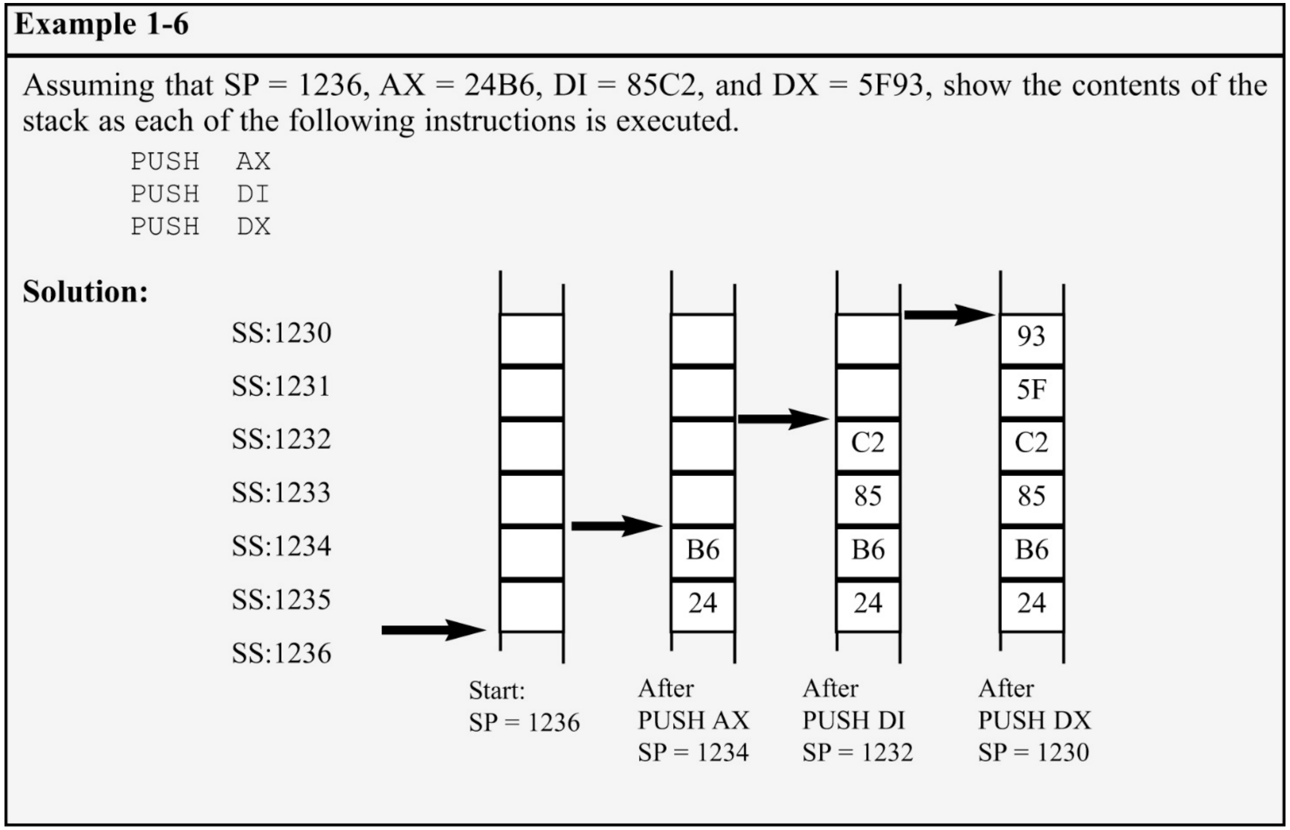
	Start: SP = 1236	After PUSH AX SP = 1234	After PUSH DI SP = 1232	After PUSH DX SP = 1230
SS:1230				93
SS:1231				5F
SS:1232			C2	C2
SS:1233			85	85
SS:1234		B6	B6	B6
SS:1235		24	24	24
SS:1236				

pushing onto the stack

- In the x86, the lower byte is always stored in the memory location with the *lower* address.

24H, the content of AH, is saved in the memory location with the address 1235.

AL is stored in location 1234.



Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
        PUSH   AX
        PUSH   DI
        PUSH   DX
```

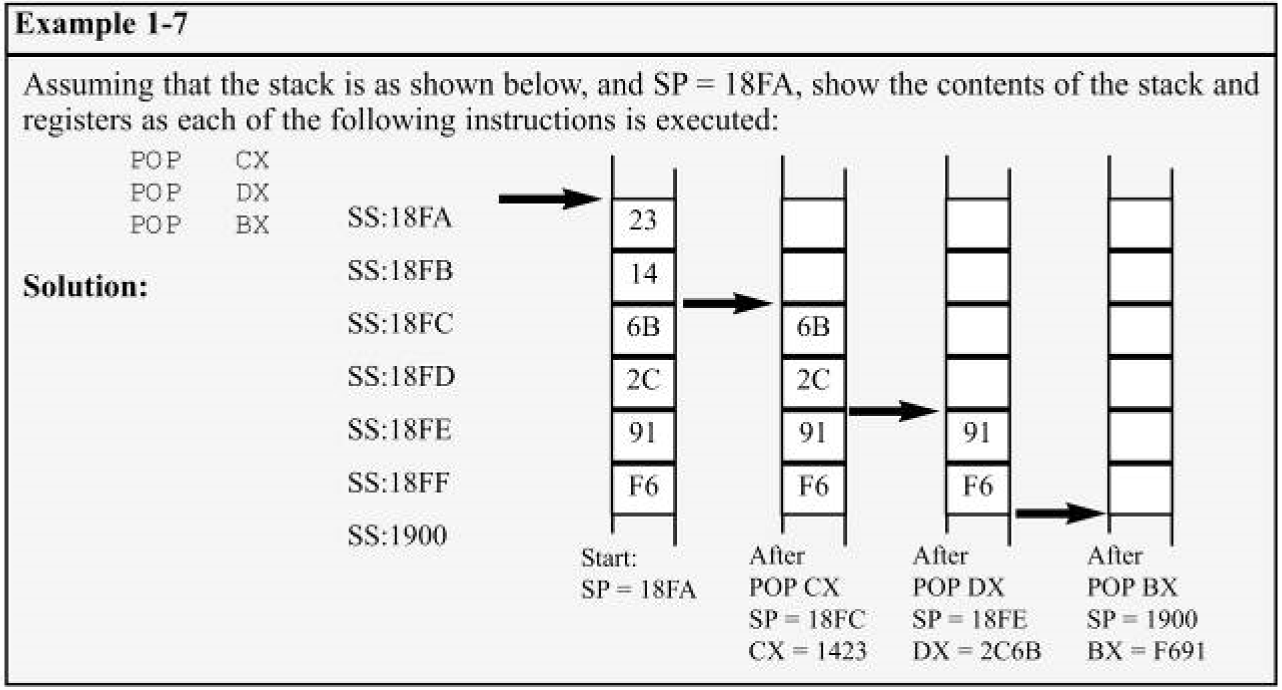
Solution:

	Start: SP = 1236	After PUSH AX SP = 1234	After PUSH DI SP = 1232	After PUSH DX SP = 1230
SS:1230				93
SS:1231				5F
SS:1232			C2	C2
SS:1233			85	85
SS:1234		B6	B6	B6
SS:1235		24	24	24
SS:1236				

popping the stack

- With every pop, the top 2 bytes of the stack are copied to the x86 CPU register specified by the instruction & the stack pointer is incremented twice.

While the data actually remains in memory, it is not accessible, since the stack pointer, SP is *beyond* that point.



Example 1-7

Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

```
POP   CX
POP   DX
POP   BX
```

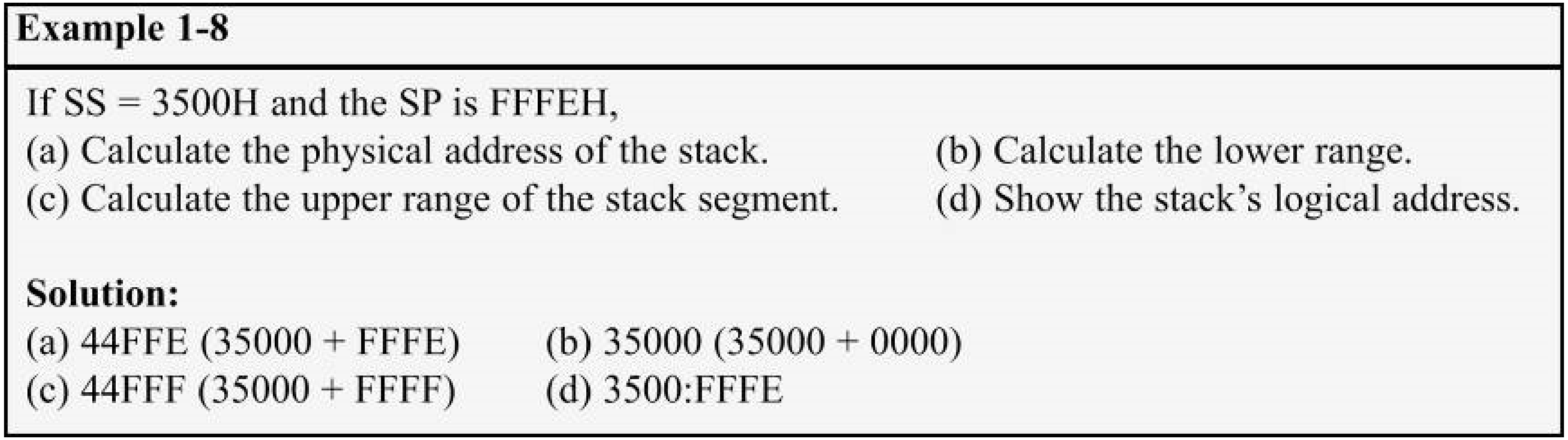
Solution:

	Start:
	SP = 18FA
		After
		POP CX
		SP = 18FC
		CX = 1423
			After
			POP DX
			SP = 18FE
			DX = 2C6B
				After
				POP BX
				SP = 1900
				BX = F691

SS:18FA	23			
SS:18FB	14			
SS:18FC	6B	6B		
SS:18FD	2C	2C		
SS:18FE	91	91	91	
SS:18FF	F6	F6	F6	
SS:1900				

logical vs physical stack address

- The exact physical location of the stack depends on the value of the stack segment (SS) register and
  SP, the stack pointer.
  - To compute physical addresses for the stack, shift left SS, then add offset SP, the stack pointer register.



Example 1-8

If SS = 3500H and the SP is FFFEH,

(a) Calculate the physical address of the stack. (b) Calculate the lower range.

(c) Calculate the upper range of the stack segment. (d) Show the stack's logical address.

Solution:

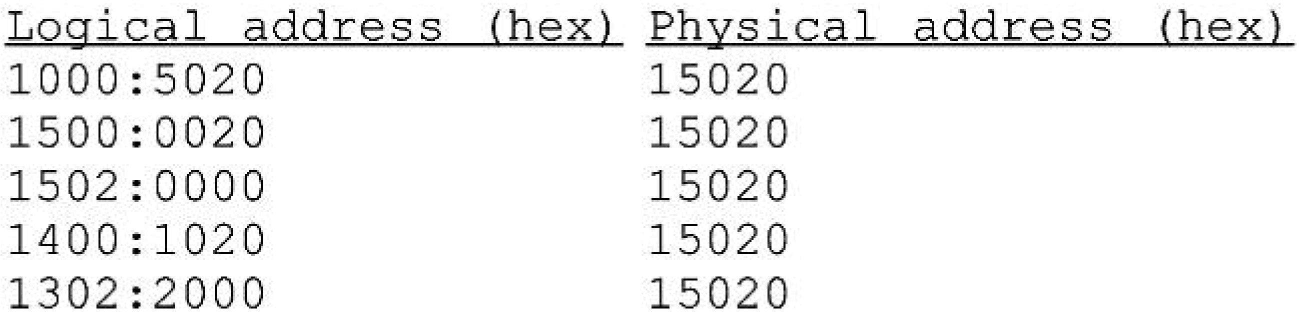
(a) 44FFE (35000 + FFFE) (b) 35000 (35000 + 0000)

(c) 44FFF (35000 + FFFF) (d) 3500:FFFE

- Usually Windows O/S assigns values for the SP and SS.

a few more words about x86 segments

- *Can a single physical address belong to many different logical addresses?*
  - Observe the physical address value of 15020H.
    - Many possible logical addresses represent this single physical address:

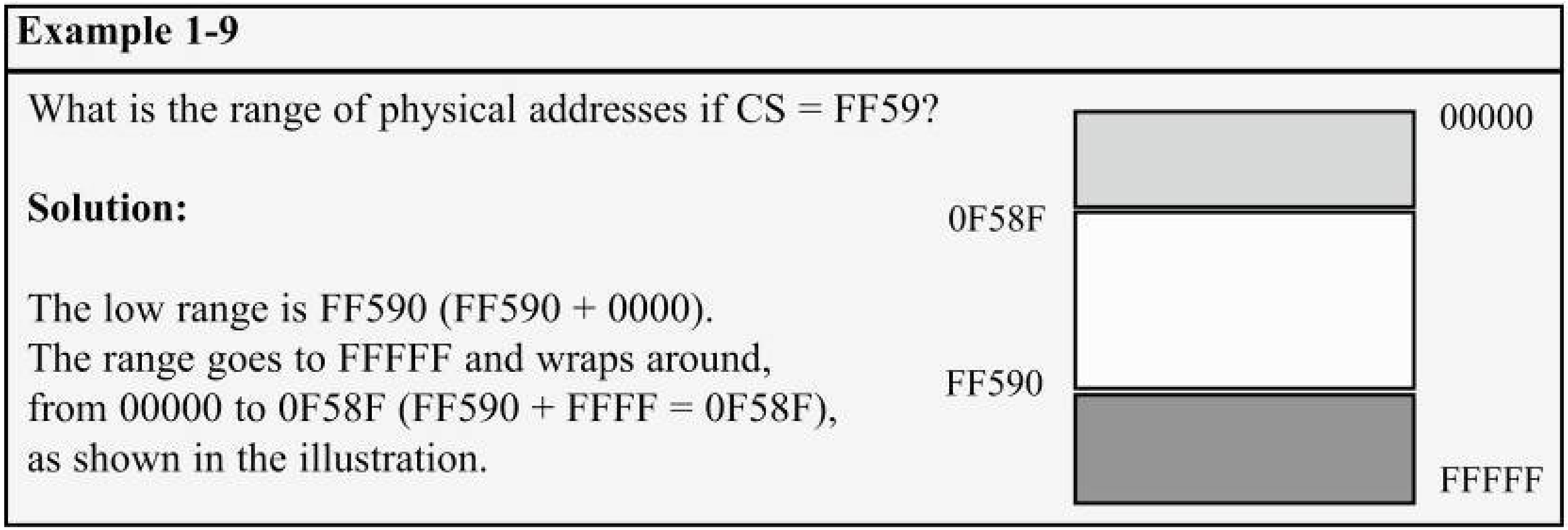


Logical address (hex)	Physical address (hex)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

  - An illustration of the dynamic behavior of the segment and offset concept in the 8086 CPU.

a few more words about x86 segments

- When adding the offset to the shifted segment register results in an address beyond the maximum allowed range of FFFFFH, *wrap-around* will occur.



Example 1-9

What is the range of physical addresses if CS = FF59?

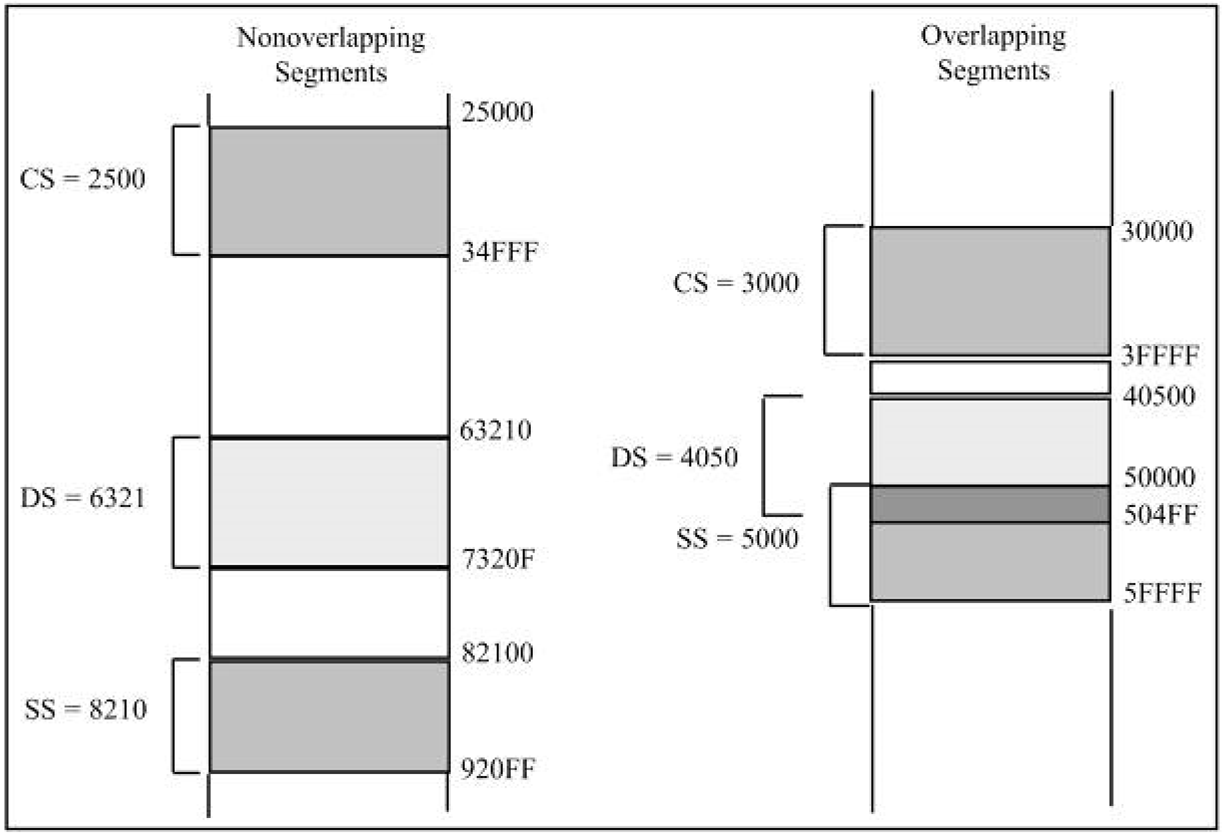
Solution:

The low range is FF590 (FF590 + 0000).
The range goes to FFFFF and wraps around,
from 00000 to 0F58F (FF590 + FFFF = 0F58F),
as shown in the illustration.

00000
0F58F
FF590
FFFFF

overlapping

- In calculating the physical address, it is possible that two segments can overlap.


Nonoverlapping Segments
CS = 2500
25000
34FFF
DS = 6321
63210
7320F
SS = 8210
82100
920FF
Overlapping Segments
CS = 3000
30000
3FFFF
40500
DS = 4050
50000
504FF
SS = 5000
5FFFF