The x86 PC

assembly language, design, and interfacing

fifth edition

Prentice Hall

```
Dec   Hex   Bin
1     1     00000001
```

```
ORG  ; ONE
```

The x86 Microprocessor

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY

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 (cont)

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.

OBJECTIVES
this chapter enables the student to: (cont)

- Demonstrate the effect of ADD instructions on the flag register.
- List the addressing modes of the 8086 and recognize examples of each mode.
- Know how to use flowcharts and pseudocode in program development.

1.1 BRIEF HISTORY OF THE x86 FAMILY
evolution from 8080/8085 to 8086

- In 1978, Intel Corporation introduced the 16-bit 8086 microprocessor, a major improvement over the previous generation 8080/8085 series.
  - The 8086 capacity of 1 megabyte of memory exceeded the 8080/8085 maximum of 64K bytes of memory.
  - 8080/8085 was an 8-bit system, which 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.
  - 8086 was a *pipelined* processor, as opposed to the *nonpipelined* 8080/8085.

1.1 BRIEF HISTORY OF THE x86 FAMILY
evolution from 8080/8085 to 8086

Table 1-1: Evolution of Intel's Microprocessors (from the 8008 to the 8088)

Product	8008	8080	8085	8086	8088
Year introduced	1972	1974	1976	1978	1979
Technology	PMOS	NMOS	NMOS	NMOS	NMOS
Number of pins	18	40	40	40	40
Number of transistors	3000	4500	6500	29,000	29,000
Number of instructions	66	111	113	133	133
Physical memory	16K	64K	64K	1M	1M
Virtual memory	None	None	None	None	None
Internal data bus	8	8	8	16	16
External data bus	8	8	8	16	8
Address bus	8	16	16	20	20
Data types	8	8	8	8/16	8/16

1.1 BRIEF HISTORY OF THE x86 FAMILY
evolution from 8086 to 8088

- The 8086 microprocessor has a 16-bit data bus, internally and externally.
  - All registers are 16 bits wide, and there is a 16-bit data bus to transfer data in and out of the CPU
    - There was resistance to a 16-bit external bus as peripherals were designed around 8-bit processors.
    - A printed circuit board with a 16-bit data also bus cost more.
- As a result, Intel came out with the 8088 version.
  - Identical to the 8086, but with an 8-bit data bus.
    - Picked up by IBM as the microprocessor in designing the PC.

1.1 BRIEF HISTORY OF THE x86 FAMILY
success of the 8088

- The 8088-based IBM PC was an great success, because IBM & Microsoft made it an open system.
  - Documentation and specifications of the hardware and software of the PC were made public
    - Making it possible for many vendors to clone the hardware successfully & spawn a major growth in both hardware and software designs based on the IBM PC.

1.1 BRIEF HISTORY OF THE x86 FAMILY 80286, 80386, and 80486

- Intel introduced the 80286 in 1982, which IBM picked up for the design of the PC AT.
  - 16-bit internal & external data buses.
  - 24 address lines, for 16mb memory. (2^{24} = 16mb)
  - Virtual memory.
- 80286 can operate in one of two modes:
  - **Real mode** - a faster 8088/8086 with the same maximum of 1 megabyte of memory.
  - **Protected mode** - which allows for 16M of memory.
    - Also capable of protecting the operating system & programs from accidental or deliberate destruction by a user.

1.1 BRIEF HISTORY OF THE x86 FAMILY 80286, 80386, and 80486

- *Virtual memory* is a way of fooling the processor into thinking it has access to an almost unlimited amount of memory.
  - By swapping data between disk storage and RAM.

1.1 BRIEF HISTORY OF THE x86 FAMILY 80286, 80386, and 80486

- In 1985 Intel introduced 80386 (or 80386DX).
  - 32-bit internally/externally, with a 32-bit address bus.
  - Capable of handling memory of up to 4 gigabytes. (2^{32})
  - Virtual memory increased to 64 terabytes. (2^{46})
- Later Intel introduced 386SX, internally identical, but with a 16-bit external data bus & 24-bit address bus.
  - This makes the 386SX system much cheaper.
- Since general-purpose processors could not handle mathematical calculations rapidly, Intel introduced numeric data processing chips.
  - Math *coprocessors*, such as 8087, 80287, 80387.

1.1 BRIEF HISTORY OF THE x86 FAMILY 80286, 80386, and 80486

- 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.

1.1 BRIEF HISTORY OF THE x86 FAMILY 80286, 80386, and 80486

Table 1-2: Evolution of Intel's Microprocessors (from the 8086 to the Pentium Pro)

Product	8086	80286	80386	80486	Pentium	Pentium Pro
Year Introduced	1978	1982	1985	1989	1993	1995
Technology	NMOS	NMOS	CMOS	CMOS	BICMOS	BICMOS
Clock rate (MHz)	3–10	10–16	16–33	25–33	60, 66	150
Number of pins	40	68	132	168	273	387
Number of transistors	29,000	134,000	275,000	1.2 mill.	3.1 mill.	5.5 mill.
Physical memory	1M	16M	4G	4G	4G	64G
Virtual memory	None	1G	64T	64T	64T	64T
Internal data bus	16	16	32	32	32	32
External data bus	16	16	32	32	64	64
Address bus	20	24	32	32	32	36
Data types	8/16	8/16	8/16/32	8/16/32	8/16/32	8/16/32

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® & Pentium® Pro

- In 1992, Intel released the Pentium®. (*not* 80586)
  - A name can be copyrighted, but numbers cannot.
- On release, Pentium® had speeds of 60 & 66 MHz.
  - Designers utilized over 3 million transistors on the Pentium® chip using submicron fabrication technology.
  - New design features made speed twice that of 80486/66.
    - Over 300 times faster than that of the original 8088.
- Pentium® is fully compatible with previous x86 processors but includes several new features.
  - Separate 8K cache memory for code and data.
  - 64-bit bus, and a vastly improved floating-point processor.

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® & Pentium® Pro

- The Pentium® is packaged in a 273-pin PGA chip
  - BICMOS technology, combines the speed of bipolar transistors with power efficiency of CMOS technology
  - 64-bit data bus, 32-bit registers & 32-bit address bus.
    - Capable of addressing 4gb of memory.
- In 1995 Intel Pentium® Pro was released—the sixth generation x86.
  - 5.5 million transistors.
  - Designed primarily for 32-bit servers & workstations.

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® & Pentium® Pro

Table 1-3: Evolution of Intel x86 Microprocessors: From Pentium II to Itanium

Product	Pentium II	Pentium III	Pentium 4	Itanium II
Year introduced	1997	1999	2000	2002
Technology	BICMOS	BICMOS	BICMOS	BICMOS
Number of transistors	7.5 mill.	9.5 mill.	42 mill.	220 mill.
Cache size	512K	512K	512K	3MB
Physical memory	64G	64G	64G	64G
Virtual memory	64T	64T	64T	64T
Internal data bus	32	32	32	64
External data bus	64	64	64	64
Address bus	36	36	36	64
Data types	8/16/32	8/16/32	8/16/32	8/16/32/64

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® II

- In 1997 Intel introduced the Pentium® II processor
  - 7.5-million-transistor processor featured MMX (MultiMedia Extension) technology incorporated into the CPU.
    - For fast graphics and audio processing.
- In 1998 the Pentium® II Xeon was released.
  - Primary market is for servers and workstations.
- In 1999, Celeron® was released.
  - Lower cost & good performance make it ideal for PCs used to meet educational and home business needs.

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® III

- In 1999 Intel released Pentium® III.
  - 9.5-million-transistor processor.
  - 70 new instructions called SIMD.
    - Enhance video/audio performance in 3-D imaging, and streaming audio.
- In 1999 Intel introduced the Pentium® III Xeon.
  - Designed more for servers and business workstations with multiprocessor configurations.

1.1 BRIEF HISTORY OF THE x86 FAMILY
Pentium® 4

- The Pentium® 4 debuted late in 1999.
  - Speeds of 1.4 to 1.5 GHz.
  - System bus operates at 400 MHz
- Completely new 32-bit architecture, called NetBurst.
  - Designed for heavy multimedia processing.
    - Video, music, and graphic file manipulation on the Internet.
  - New cache and pipelining technology & expansion of the multimedia instruction set make the P4 a high-end media processing microprocessor.

1.1 BRIEF HISTORY OF THE x86 FAMILY
Intel 64 Architecture

- Intel has selected Itanium® as the new brand name for the first product in its 64-bit family of processors.
  - Formerly called Merced.
- The evolution of microprocessors is increasingly influenced by the evolution of the Internet.
  - Itanium® architecture is designed to meet Internet-driven needs for servers & high-performance workstations.
  - Itanium® will have the ability to execute many instructions simultaneously, plus extremely large memory capabilities.

1.2 INSIDE THE 8088/86

- There are two ways to make the CPU process information faster:
  - Increase the working frequency.
    - Using technology available, with cost considerations.
  - Change the internal architecture of the CPU.

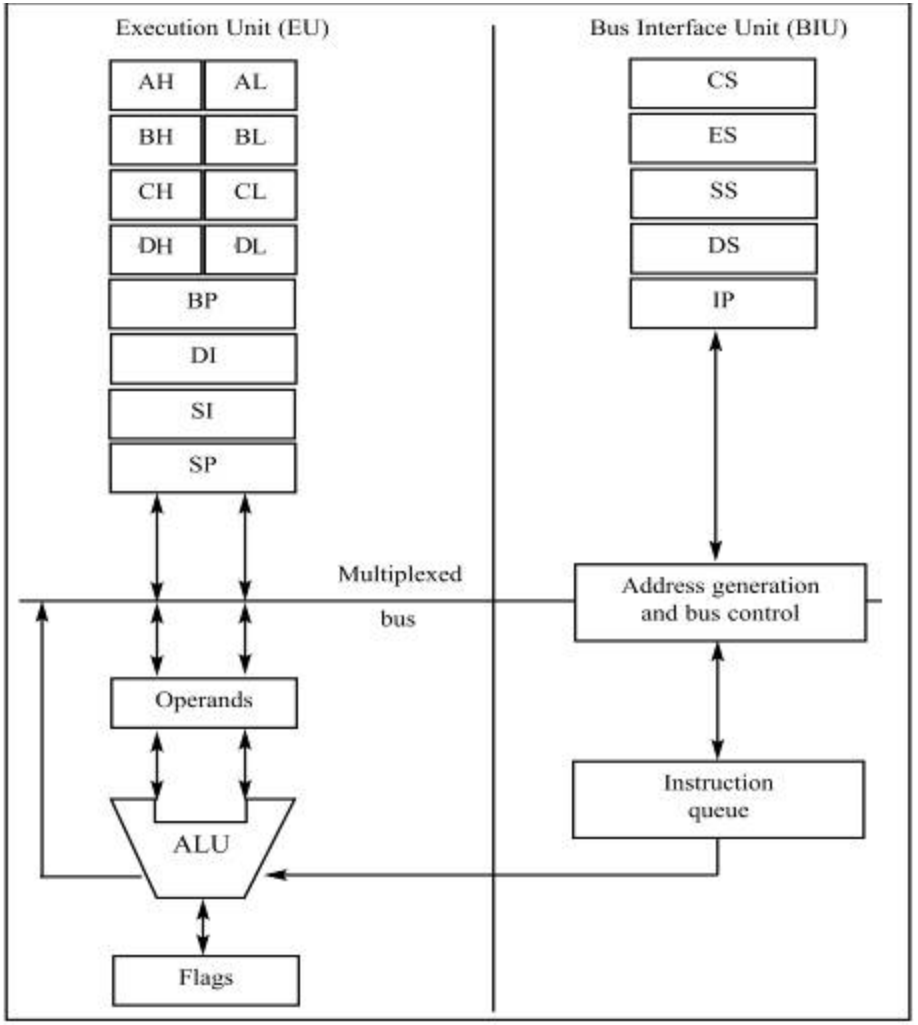


Figure 1-1
Internal Block Diagram of the 8088/86 CPU
(Reprinted by permission of Intel Corporation, Copyright Intel Corp.1989)

1.2 INSIDE THE 8088/86

pipelining

- 8085 could fetch *or* execute at any given time.
  - The idea of pipelining in its simplest form is to allow the CPU to fetch *and* execute at the same time.

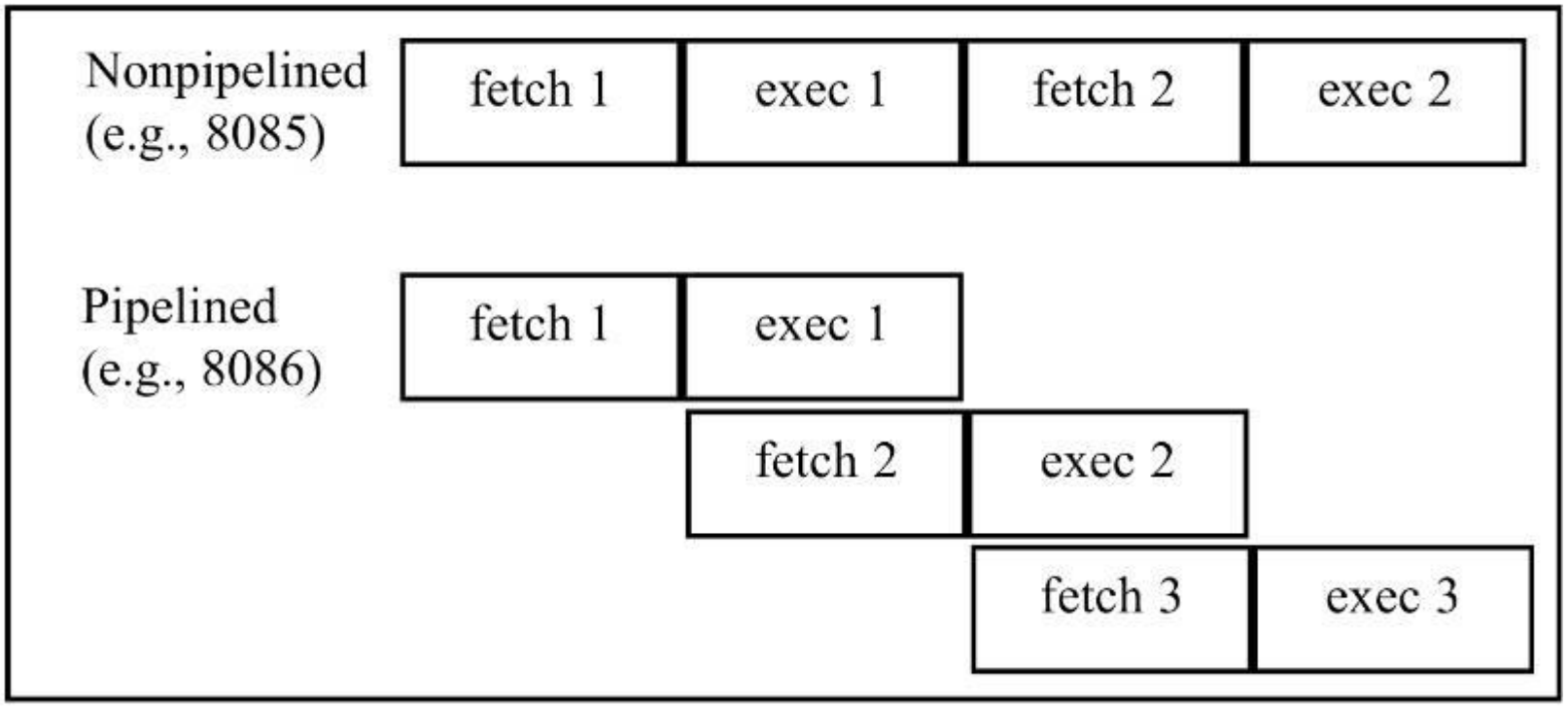


Figure 1-2 Pipelined vs Nonpipelined Execution

1.2 INSIDE THE 8088/86

pipelining

- 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.
- 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.
- 8088/86 pipelining has two stages, *fetch & execute.*
  - In more powerful computers, it can have many stages.

1.2 INSIDE THE 8088/86

pipelining

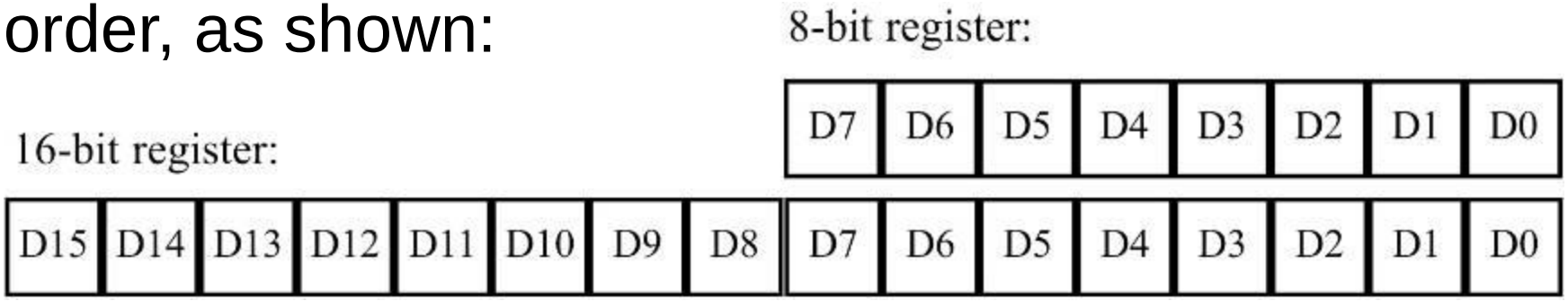
- If an instruction takes too long to execute, the queue is filled to capacity and the buses will sit idle
- In some circumstances, the microprocessor must flush out the queue.
  - When a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information 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.

1.2 INSIDE THE 8088/86 registers

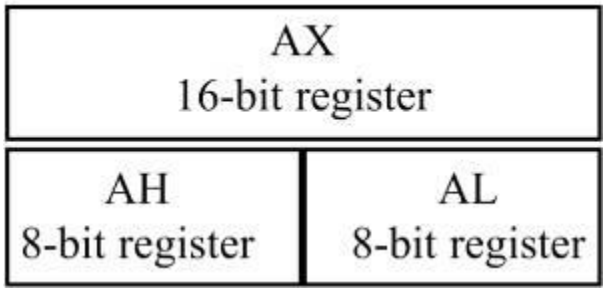
- In the CPU, registers store information temporarily.
  - One or two bytes of data to be processed.
  - The address of data.
- General-purpose registers in 8088/86 processors can be accessed as either 16-bit or 8-bit registers
  - All other registers can be accessed only as the full 16 bits.
- In 8088/86, 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.

1.2 INSIDE THE 8088/86 registers

- The bits of a register are numbered in descending order, as shown:



- The first letter of each register indicates its use.
  - AX is used for the accumulator.
  - BX is a base addressing register.
  - CX is a counter in loop operations.
  - DX points to data in I/O operations.



1.2 INSIDE THE 8088/86 registers

Table 1-4: Registers of the 8088/86/286 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).

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING

- The CPU can work only in binary, very high speeds.
  - It is tedious & slow for humans to deal with 0s & 1s in order to program the computer.
- A program of 0s & 1s is called machine language.
  - Early computer programmers actually coded programs in machine language.
- Eventually, Assembly languages were developed, which provided *mnemonics* for machine code.
  - Mnemonic is typically used in computer science and engineering literature to refer to codes & abbreviations that are relatively easy to remember.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING

- Assembly language is referred to as a *low-level* language because it deals directly with the internal structure of the CPU.
  - Assembly language programs must be translated into machine code by a program called an *assembler.*
  - To program in Assembly language, programmers must know the number of registers and their size.
    - As well as other details of the CPU.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING

- Today there are many different programming languages, such as C/C++, BASIC, C#, etc.
  - Called *high-level* languages because the programmer does not have to be concerned with internal CPU details.
- High-level languages are translated into machine code by a program called a *compiler.*
  - To write a program in C, one must use a *C compiler* to translate the program into machine language.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING

- There are numerous assemblers available for translating x86 Assembly language programs into machine code.
  - One of the most commonly used assemblers is MASM by Microsoft.
- The present chapter is designed to correspond to Appendix A: DEBUG Programming.
  - Provided with the Microsoft Windows operating system.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING 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.*

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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)
```

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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
```

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
MOV instruction

- In the 8086 CPU, data can be moved among all the registers, as long as the source and destination registers match in size (*Except* the flag register.)
  - There is no such instruction as "**MOV FR,AX**".
- Code such as "**MOV AL,DX**" will cause an error.
  - One cannot move the contents of a 16-bit register into an 8-bit register.

Table 1-4: Registers of the 8088/86/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
MOV instruction

- Using the MOV instruction, data can be moved directly into nonsegment registers only.
  - The following demonstrates legal & illegal instructions.

```
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)
```

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
MOV instruction

- Values *cannot* be loaded directly into any segment register (CS, DS, ES, or SS).
  - To load a value into a segment register, load it to a non-segment register, then move it to the segment register.

```
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
```

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
MOV instruction

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

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
ADD instruction

- The program above can be written in many ways, depending on the registers used, such as:

```
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 program above results in DH = 59H and CL = 34H.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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.

1.3 INTRODUCTION TO ASSEMBLY PROGRAMMING
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.*

1.4 INTRODUCTION TO PROGRAM SEGMENTS

- A typical Assembly language program consists of at least three segments:
  - **A code segment** - which contains the Assembly language instructions that perform the tasks that the program was designed to accomplish.
  - **A data segment** - used to store information (data) to be processed by the instructions in the code segment.
  - **A stack segment** - used by the CPU to store information temporarily.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
origin and definition of the segment

- A *segment* is an area of memory that includes up to 64K bytes, and begins on an address evenly divisible by 16 (such an address ends in 0H)
  - 8085 addressed a maximum of 64K of physical memory, since it had only 16 pins for address lines. (2^{16} = 64K)
    - Limitation was carried into 8088/86 design for compatibility.
- In 8085 there was 64K bytes of memory for all code, data, and stack information.
  - In 8088/86 there can be up to 64K bytes in each category.
    - The code segment, data segment, and stack segment.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

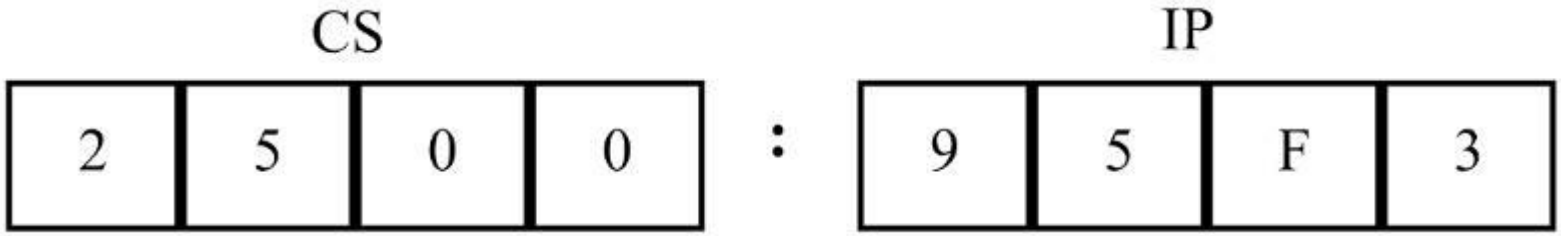
1.4 INTRODUCTION TO PROGRAM SEGMENTS

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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
code segment

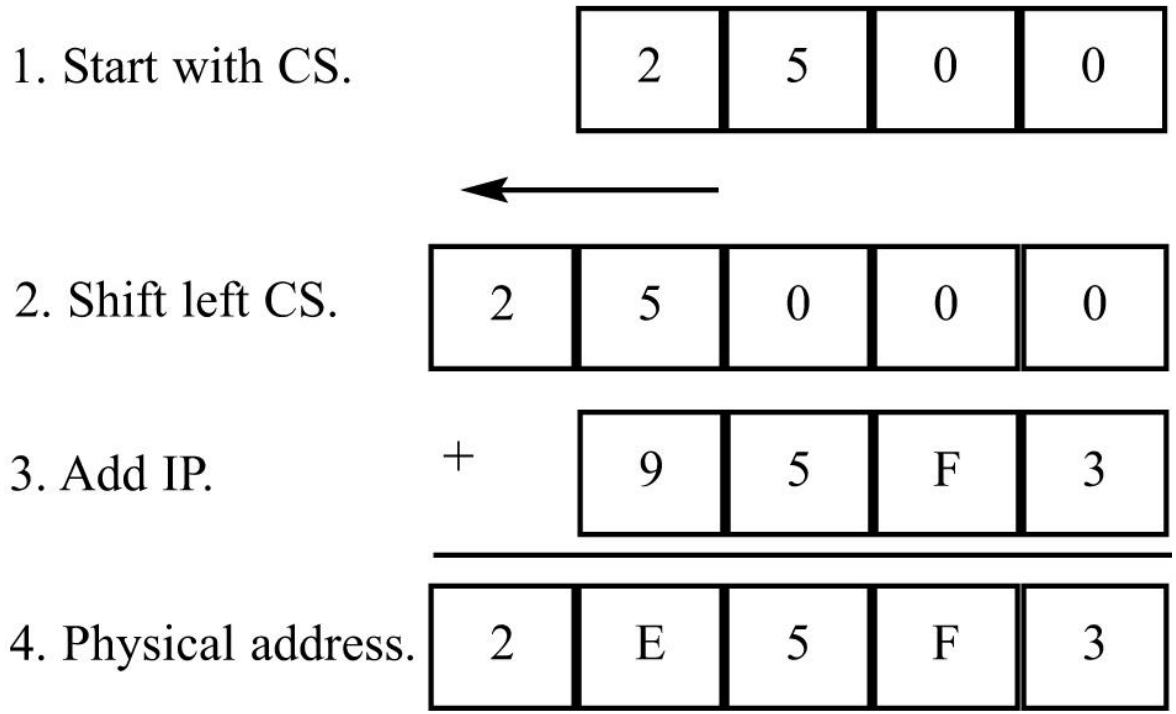
- Assume values in CS & IP as shown in the diagram:



  - 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

1.4 INTRODUCTION TO PROGRAM SEGMENTS
code segment

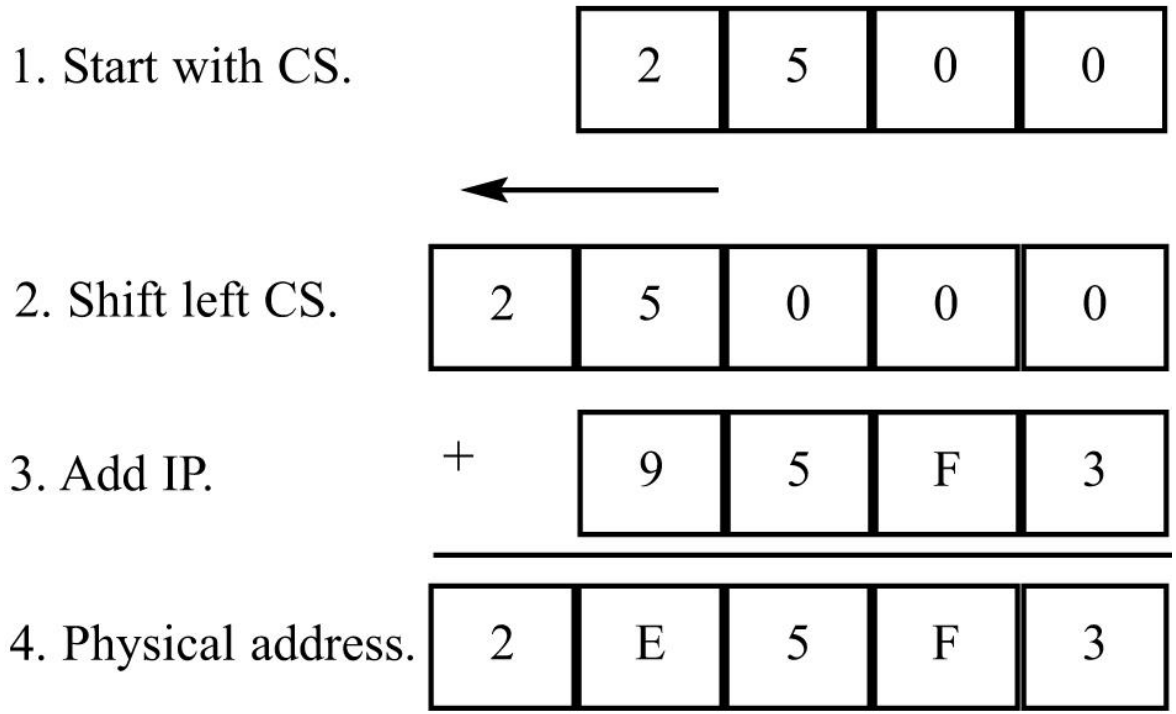
- Calculate the physical address of an instruction:



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

1.4 INTRODUCTION TO PROGRAM SEGMENTS
code segment

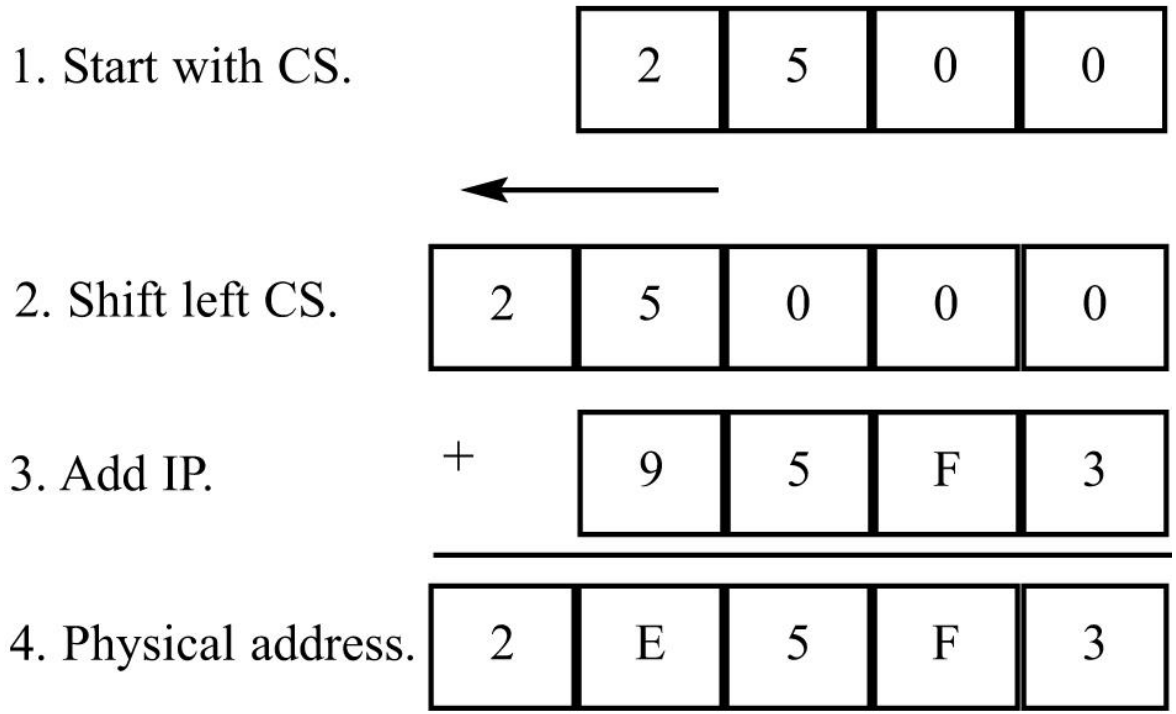
- Calculate the physical address of an instruction:



  - 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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
code segment

- Calculate the physical address of an instruction:



  - 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).

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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)

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
code segment logical/physical address

LOGICAL ADDRESS CS:IP	MACHINE LANGUAGE OPCODE AND OPERAND	ASSEMBLY LANGUAGE 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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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
```

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
data segment logical/physical address

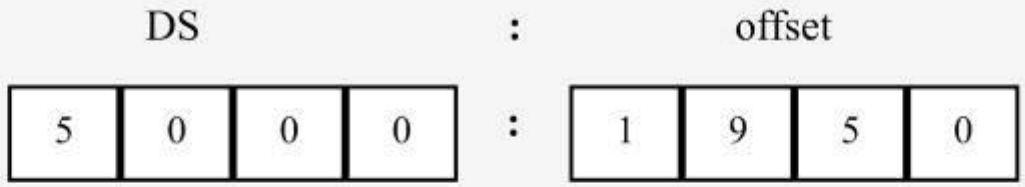
- The physical address for data is calculated using the same rules as for the code segment.
  - The physical address of data is calculated by shifting DS left one hex digit and adding the offset value, as shown in Examples 1-2, 1-3, and 1-4.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
data segment logical/physical address

Example 1-2

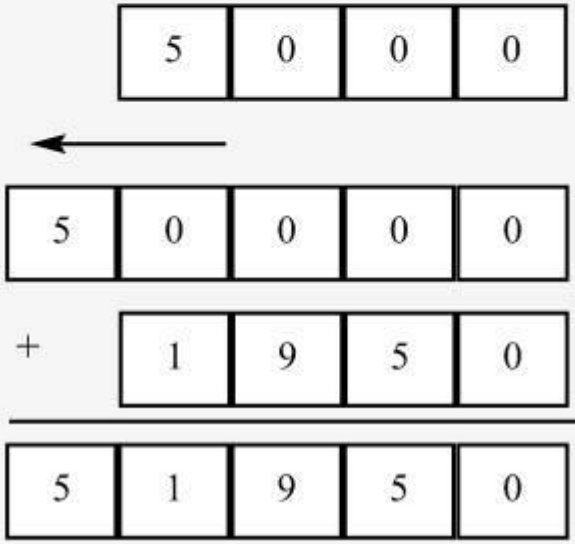
Assume that DS is 5000 and the offset is 1950. Calculate the physical address.

Solution:



The physical address will be 50000 + 1950 = 51950.

1. Start with DS.



2. Shift DS left.

3. Add the offset.

4. Physical address.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

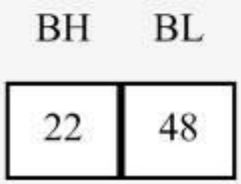
Example 1-5

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



1.4 INTRODUCTION TO PROGRAM SEGMENTS
little endian convention

- All Intel microprocessors and many microcontrollers use the little endian convention.
  - Freescale (formerly Motorola) microprocessors, along with some other microcontrollers, use big endian.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
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.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
memory map of the IBM PC

- The 20-bit address of 8088/86 allows 1mb (1024K bytes) of memory space with the address range 00000–FFFFF.
  - During the design phase of the first IBM PC, engineers had to decide on the allocation of the 1-megabyte memory space to various sections of the PC.
    - This memory allocation is called a *memory map*.

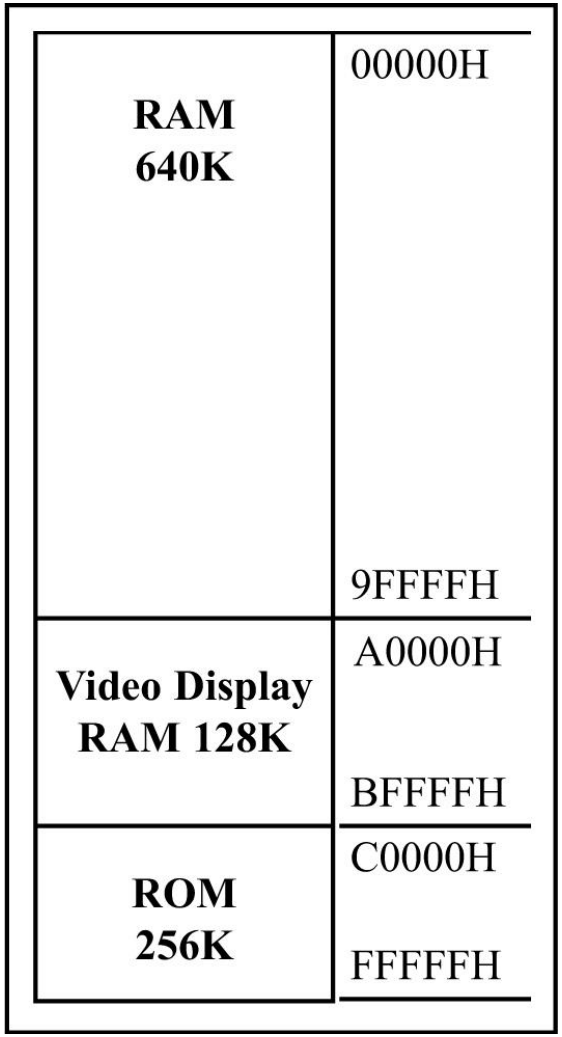


Figure 1-3 Memory Allocation in the PC

1.4 INTRODUCTION TO PROGRAM SEGMENTS
memory map of the IBM PC

- Of this 1 megabyte, 640K bytes from addresses **00000–9FFFFH** were set aside for RAM
- 128K bytes **A0000H– BFFFFH** were allocated for video memory
- The remaining 256K bytes from **C0000H–FFFFFH** were set aside for ROM

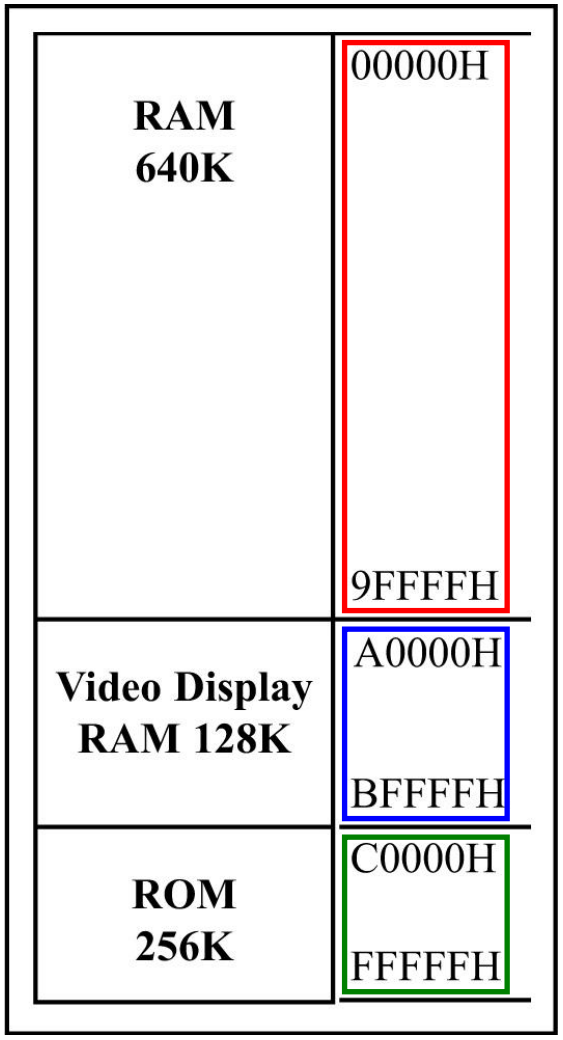


Figure 1-3 Memory Allocation in the PC

1.4 INTRODUCTION TO PROGRAM SEGMENTS
more about RAM

- In the early 80s, most PCs came with 64K to 256K bytes of RAM, more than adequate at the time
  - Users had to buy memory to expand up to 640K.
- Managing RAM is left to Windows because...
  - The amount of memory used by Windows varies.
  - Different computers have different amounts of RAM.
  - Memory needs of application packages vary.
- For this reason, we do not assign any values for the CS, DS, and SS registers.
  - Such an assignment means specifying an exact physical address in the range 00000–9FFFFH, and this is beyond the knowledge of the user.

1.4 INTRODUCTION TO PROGRAM SEGMENTS
video RAM

- From A0000H to BFFFFH is set aside for video
  - The amount used and the location vary depending on the video board installed on the PC

1.4 INTRODUCTION TO PROGRAM SEGMENTS
more about ROM

- C0000H to FFFFFH is set aside for ROM.
  - Not all the memory in this range is used by the PC's ROM.
- 64K bytes from location F0000H–FFFFFH are used by BIOS (basic input/output system) ROM.
  - Some of the remaining space is used by various adapter cards (such as the network card), and the rest is free.
- The 640K bytes from 00000 to 9FFFFH is referred to as *conventional memory*.
  - The 384K bytes from A0000H to FFFFFH are called the UMB (*upper memory block*).

1.4 INTRODUCTION TO PROGRAM SEGMENTS function of BIOS ROM

- There must be some permanent (nonvolatile) memory to hold the programs telling the CPU what to do when the power is turned on
  - This collection of programs is referred to as BIOS.
- BIOS stands for *basic input-output system*.
  - It contains programs to test RAM and other components connected to the CPU.
  - It also contains programs that allow Windows to communicate with peripheral devices.
  - The BIOS tests devices connected to the PC when the computer is turned on and to report any errors.

1.5 THE STACK
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.

1.5 THE STACK
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.*

1.5 THE STACK
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`**".

1.5 THE STACK
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.

1.5 THE STACK
pushing onto the stack

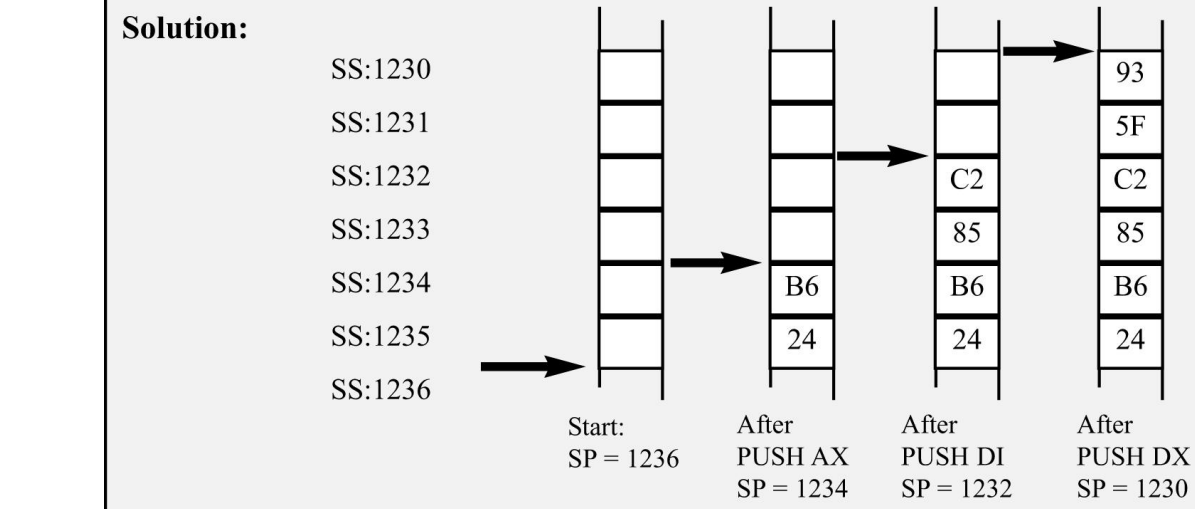
- As each PUSH is executed, the register contents are saved on the stack and SP is decremented by 2.

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:



1.5 THE STACK
pushing onto the stack

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

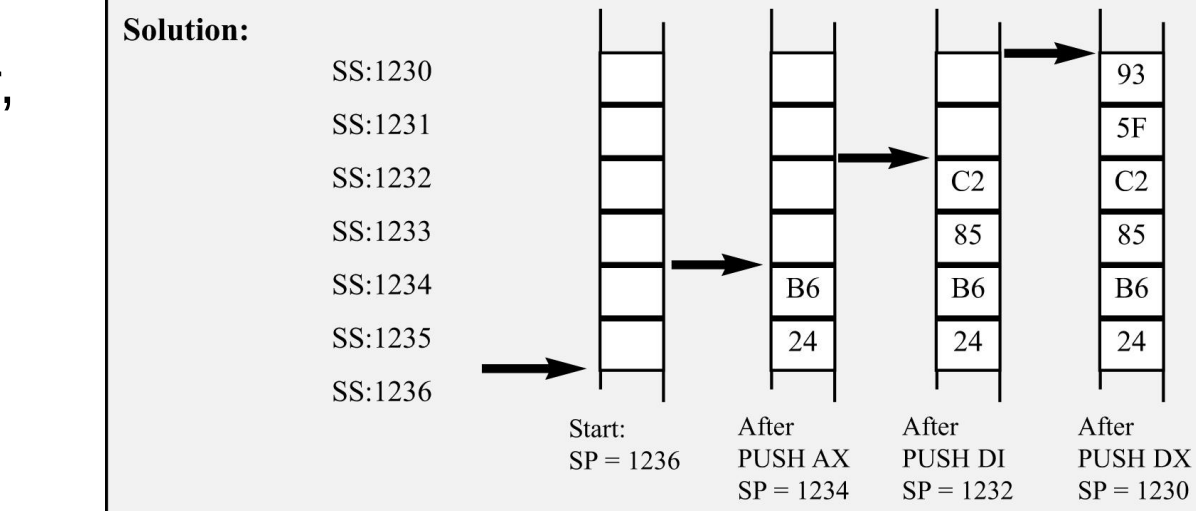
Since the push is saving the contents of a 16-bit register, it decrements *twice.*

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:



1.5 THE STACK
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.

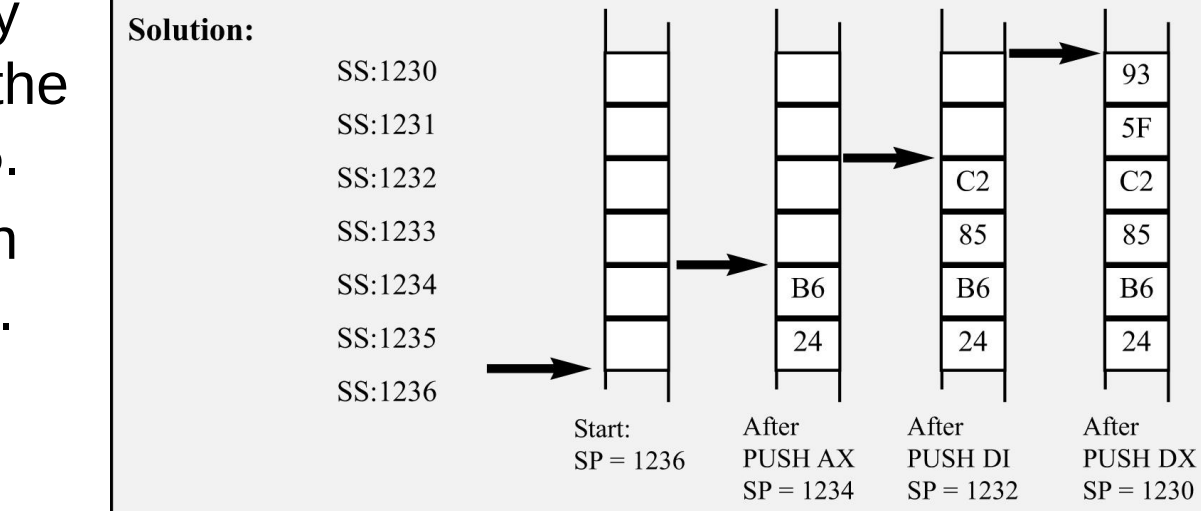
AL is stored in location 1234.

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:



1.5 THE STACK
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.

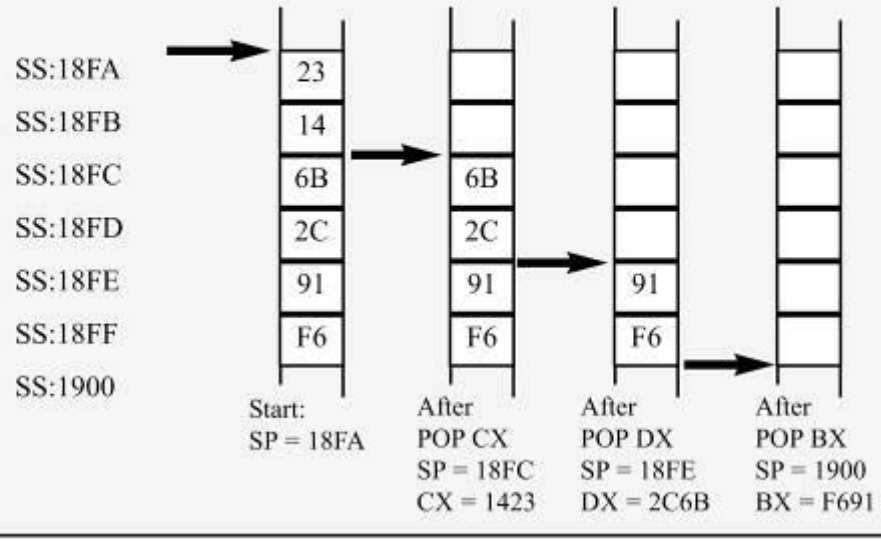
While the data actually remains in memory, it is not accessible, since the stack pointer, SP is *beyond* that point.

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:



1.5 THE STACK
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

  - Windows assigns values for the SP and SS.

1.5 THE STACK
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.

1.5 THE STACK
a few more words about x86 segments

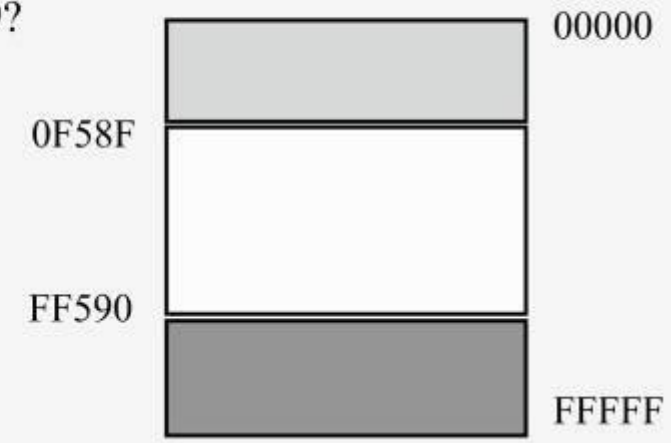
- When adding the offset to the shifted segment register results in an address beyond the maximum allowed range of FFFFFH, *wrap-around* will occur.

Example 1-9

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

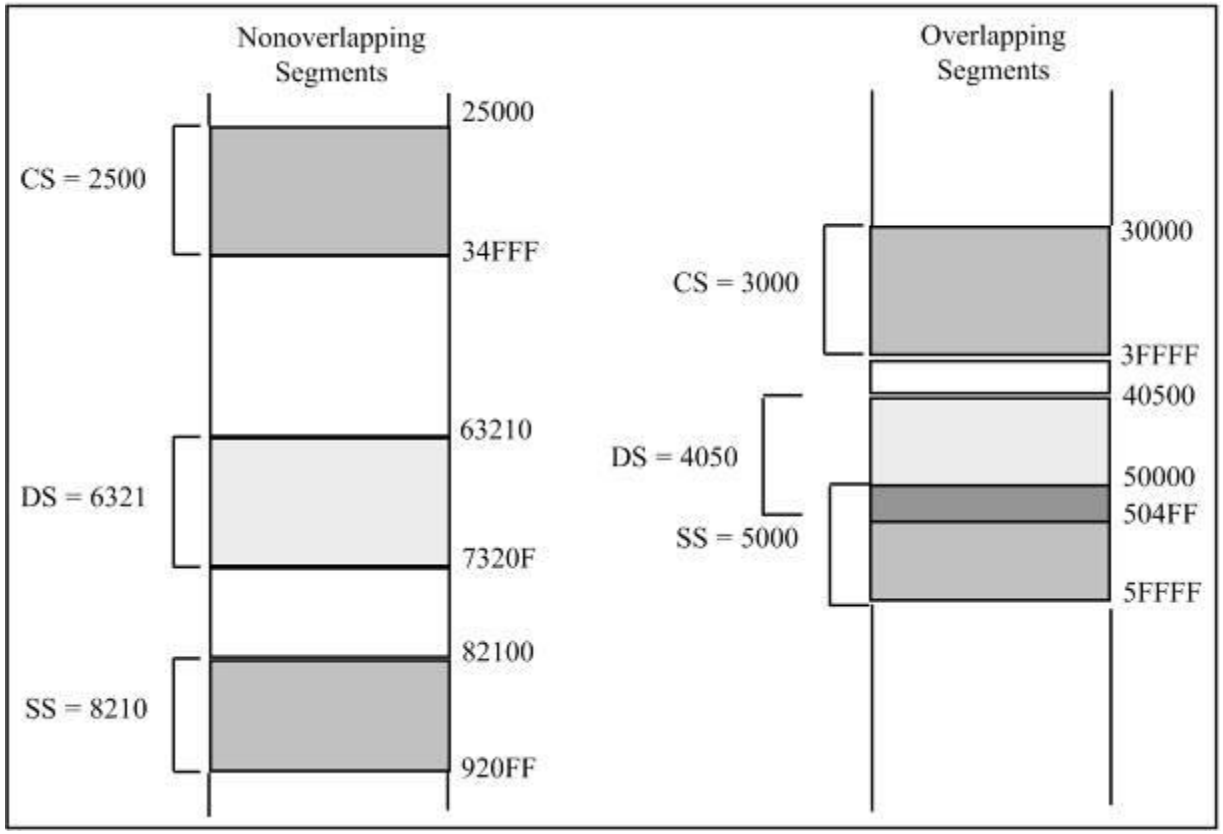
Solution:

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.



1.5 THE STACK
overlapping

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

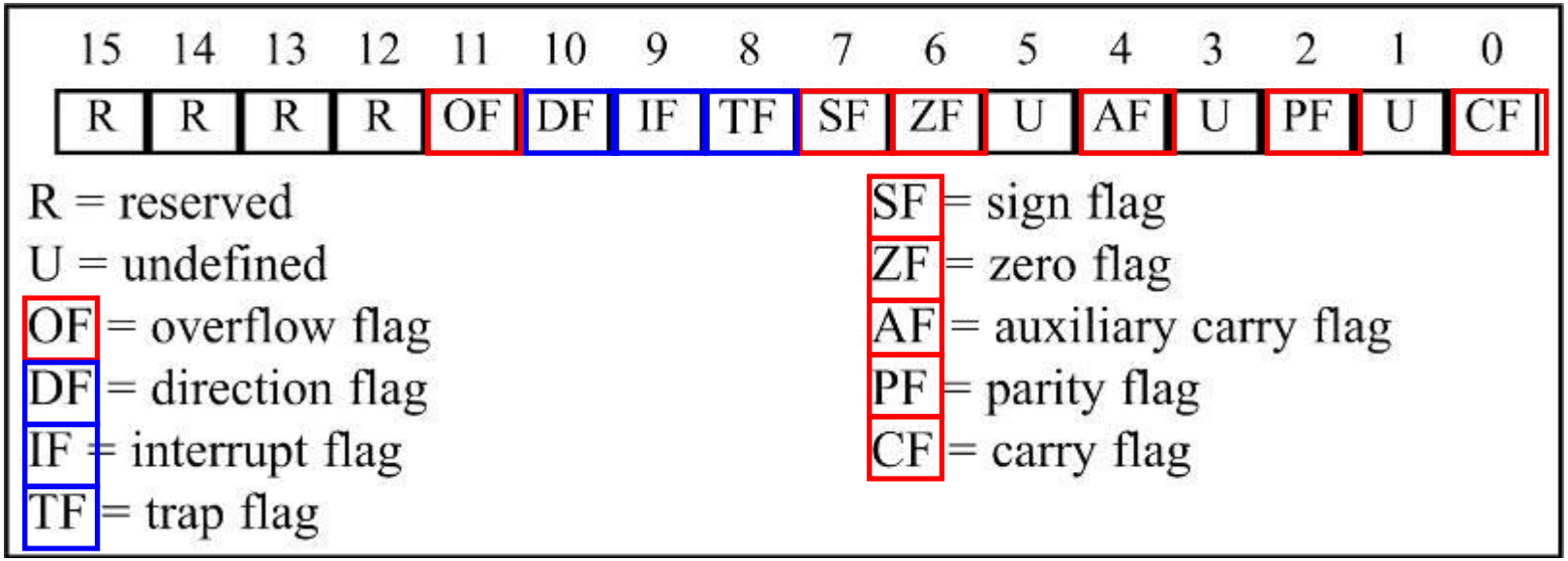


1.6 FLAG REGISTER

- Many Assembly language instructions alter flag register bits & some instructions function differently based on the information in the flag register.
- The flag register is a 16-bit register sometimes referred to as the *status register.*
  - Although 16 bits wide, only some of the bits are used.
    - The rest are either undefined or reserved by Intel.

1.6 FLAG REGISTER

- Six flags, called *conditional flags*, indicate some condition resulting after an instruction executes.



  - These six are **CF**, **PF**, **AF**, **ZF**, **SF**, and **OF**.
  - The remaining three, often called *control flags*, control the operation of instructions *before* they are executed.

1.6 FLAG REGISTER
bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **CF (Carry Flag) -** Set when there is a carry out, from d7 after an 8-bit operation, or d15 after a 16-bit operation.
    - Used to detect errors in unsigned arithmetic operations.
  - **PF (Parity Flag) -** After certain operations, the parity of the result's low-order byte is checked.
    - If the byte has an even number of 1s, the parity flag is set to 1; otherwise, it is cleared.
  - **AF (Auxiliary Carry Flag) -** If there is a carry from d3 to d4 of an operation, this bit is set; otherwise, it is cleared.
    - Used by instructions that perform BCD (binary coded decimal) arithmetic.

1.6 FLAG REGISTER
bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **ZF (Zero Flag) -** Set to 1 if the result of an arithmetic or logical operation is zero; otherwise, it is cleared.
  - **SF (Sign Flag) -** Binary representation of signed numbers uses the most significant bit as the sign bit.
    - After arithmetic or logic operations, the status of this sign bit is copied into the SF, indicating the sign of the result.
  - **TF (Trap Flag) -** When this flag is set it allows the program to single-step, meaning to execute one instruction at a time.
    - Single-stepping is used for debugging purposes.

1.6 FLAG REGISTER
bits of the flag register

- Flag register bits used in x86 Assembly language programming, with a brief explanation each:
  - **IF (Interrupt Enable Flag) -** This bit is set or cleared to enable/disable only external maskable interrupt requests.
  - **DF (Direction Flag) -** Used to control the direction of string operations.
  - **OF (Overflow Flag) -** Set when the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.
    - Used only to detect errors in signed arithmetic operations.

1.6 FLAG REGISTER
flag register and ADD instruction

- Flag bits affected by the ADD instruction:
  - CF (carry flag); PF (parity flag); AF (auxiliary carry flag).
  - ZF (zero flag); SF (sign flag); OF (overflow flag).

Example 1-10

Show how the flag register is affected by the addition of 38H and 2FH.

Solution:

```
        MOV   BH,38H          ;BH= 38H
        ADD   BH,2FH          ;add 2F to BH, now BH=67H

        38        0011  1000
+       2F        0010  1111
        67        0110  0111
```

CF = 0 since there is no carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 0 since there is an odd number of 1s in the result
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero

1.6 FLAG REGISTER
flag register and ADD instruction

- Flag bits affected by the ADD instruction:
  - CF (carry flag); PF (parity flag); AF (auxiliary carry flag).
  - ZF (zero flag); SF (sign flag); OF (overflow flag).

Example 1-11

Show how the flag register is affected by

```
MOV   AL,9CH      ;AL=9CH
MOV   DH,64H      ;DH=64H
ADD   AL,DH       ;now AL=0
```

Solution:

	9C	1001	1100
+	64	0110	0100
	00	0000	0000

CF = 1 since there is a carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the result
ZF = 1 since the result is zero
SF = 0 since d7 of the result is zero

1.6 FLAG REGISTER
flag register and ADD instruction

- It is important to note differences between 8- and 16-bit operations in terms of impact on the flag bits.
  - The parity bit only counts the lower 8 bits of the result and is set accordingly.

Example 1-12

Show how the flag register is affected by

```
MOV   AX,34F5H    ;AX= 34F5H
ADD   AX,95EBH    ;now AX= CAE0H
```

Solution:

	34F5	0011	0100	1111	0101
+	95EB	1001	0101	1110	1011
	CAE0	1100	1010	1110	0000

CF = 0 since there is no carry beyond d15
ZF = 0 since the result is not zero
AF = 1 since there is a carry from d3 to d4
SF = 1 since d15 of the result is one
PF = 0 since there is an odd number of 1s in the lower byte

1.6 FLAG REGISTER
flag register and ADD instruction

- The carry flag is set if there is a carry beyond bit d15 instead of bit d7.
  - Since the result of the entire 16-bit operation is zero (meaning the contents of BX), ZF is set to high.

Example 1-13

Show how the flag register is affected by

```
        MOV    BX,AAAAH     ;BX= AAAAH
        ADD    BX,5556H     ;now BX= 0000H
```

Solution:

	AAAA	1010	1010	1010	1010
+	5556	0101	0101	0101	0110
	0000	0000	0000	0000	0000

CF = 1 since there is a carry beyond d15
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the lower byte
ZF = 1 since the result is zero
SF = 0 since d15 of the result is zero

1.6 FLAG REGISTER
flag register and ADD instruction

- Instructions such as data transfers (MOV) affect no flags.

Example 1-14

Show how the flag register is affected by

```
MOV   AX,94C2H    ;AX=94C2H
MOV   BX,323EH    ;BX=323EH
ADD   AX,BX       ;now AX=C700H
MOV   DX,AX       ;now DX=C700H
MOV   CX,DX       ;now CX=C700H
```

Solution:

	94C2	1001	0100	1100	0010
+	323E	0011	0010	0011	1110
	C700	1100	0111	0000	0000

After the ADD operation, the following are the flag bits:

CF = 0 since there is no carry beyond d15

ZF = 0 since the result is not zero

AF = 1 since there is a carry from d3 to d4

SF = 1 since d15 of the result is 1

PF = 1 since there is an even number of 1s in the lower byte

1.6 FLAG REGISTER
use of the zero flag for looping

- A widely used application of the flag register is the use of the zero flag to implement program loops.
  - A *loop* is a set of instructions repeated a number of times.

1.6 FLAG REGISTER
use of the zero flag for looping

- As an example, to add 5 bytes of data, a *counter* can be used to keep track of how many times the loop needs to be repeated.
  - Each time the addition is performed the counter is decremented and the zero flag is checked.
    - When the counter becomes zero, the zero flag is set (ZF = 1) and the loop is stopped.

```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

1.6 FLAG REGISTER
use of the zero flag for looping

- Register CX is used to hold the counter.
  - BX is the offset pointer.
    - (SI or DI could have been used instead)

```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

1.6 FLAG REGISTER
use of the zero flag for looping

- AL is initialized before the start of the loop
  - In each iteration, ZF is checked by the JNZ instruction
    - JNZ stands for "Jump Not Zero", meaning that if ZF = 0, jump to a new address.
    - If ZF = 1, the jump is *not* performed, and the instruction below the jump will be executed.

```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

1.6 FLAG REGISTER
use of the zero flag for looping

- JNZ instruction must come *immediately after* the instruction that decrements CX.
  - JNZ needs to check the effect of "DEC CX" on ZF.
    - If any instruction were placed between them, that instruction might affect the zero flag.

```
        MOV   CX,05      ;CX holds the loop count
        MOV   BX,0200H   ;BX holds the offset data address
        MOV   AL,00      ;initialize AL
ADD_LP: ADD   AL,[ BX]   ;add the next byte to AL
        INC   BX         ;increment the data pointer
        DEC   CX         ;decrement the loop counter
        JNZ   ADD_LP     ;jump to next iteration if counter
                          not zero
```

1.7 x86 ADDRESSING MODES

- The CPU can access operands (data) in various ways, called *addressing modes.*
  - The number of addressing modes is determined when the microprocessor is designed & cannot be changed
- The x86 provides seven distinct addressing modes:
  - 1 - Register
  - 2 - Immediate
  - 3 - Direct
  - 4 - Register indirect
  - 5 - Based relative
  - 6 - Indexed relative
  - 7 - Based indexed relative

1.7 x86 ADDRESSING MODES
register addressing mode

- *Register addressing mode* involves use of registers to hold the data to be manipulated.
  - Memory is not accessed, so it is relatively fast.
- Examples of register addressing mode:

```
MOV   BX,DX  ;copy the contents of DX into BX
MOV   ES,AX  ;copy the contents of AX into ES
ADD   AL,BH  ;add the contents of BH to contents of AL
```

  - The the source & destination registers must match in size.
    - Coding "**`MOV CL,AX`**" will give an error, since the source is a 16-bit register and the destination is an 8-bit register.

1.7 x86 ADDRESSING MODES
immediate addressing mode

- In *immediate addressing mode*, as the name implies, when the instruction is assembled, the operand comes immediately after the opcode.
  - The source operand is a constant.
- This mode can be used to load information into any of register except the segment and flag registers.

```
MOV  AX,2550H      ;move 2550H into AX
MOV CX,625         ;load the decimal value 625 into CX
MOV BL,40H         ;load 40H into BL
```

1.7 x86 ADDRESSING MODES
immediate addressing mode

- To move information to the segment registers, the data must *first* be moved to a general-purpose register, *then* to the segment register.

```
MOV   AX,2550H
MOV   DS,AX
MOV   DS,0123H ;illegal! cannot move data into segment reg.
```

1.7 x86 ADDRESSING MODES
direct addressing mode

- In *direct addressing mode*, the data is in some memory location(s).
  - In most programs, the data to be processed is often in some memory location *outside* the CPU.
  - The address of the data in memory comes immediately after the instruction.

1.7 x86 ADDRESSING MODES
direct addressing mode

- The address of the operand is provided with the instruction, as an offset address.
  - Calculate the physical address by shifting left the DS register and adding it to the offset:

```
MOV DL,[2400]   ;move contents of DS:2400H into DL
```

- Note the **bracket** around the address.
  - If the bracket is absent, executing the command will give an error, as it is interpreted to move the *value* 2400 (16-bit data) into register DL.
    - An 8-bit register.

1.7 x86 ADDRESSING MODES
register indirect addressing mode

- In *register indirect addressing mode*, the address of the memory location where the operand resides is held by a register.
  - The registers used for this purpose are SI, DI, and BX.
- If these three registers are used as pointers, they must be combined with DS in order to generate the 20-bit physical address.

```
MOV AL,[ BX]  ;moves into AL the contents of the memory
              ;location pointed to by DS:BX.
```

  - Notice that **BX** is in **brackets**.
  - The physical address is calculated by shifting DS left one hex position and adding BX to it.

1.7 x86 ADDRESSING MODES
register indirect addressing mode

- The same rules apply when using register SI or DI.

```
MOV   CL,[ SI]        ;move contents of DS:SI into CL
MOV   [ DI] ,AH       ;move contents of AH into DS:DI
```

- Example 1-16 shows 16-bit data.

Example 1-16

Assume that DS = 1120, SI = 2498, and AX = 17FE. Show the contents of memory locations after the execution of "`MOV [ SI] ,AX`".

Solution:

The contents of AX are moved into memory locations with logical address DS:SI and DS:SI + 1; therefore, the physical address starts at DS (shifted left) + SI = 13698. According to the little endian convention, low address 13698H contains FE, the low byte, and high address 13699H will contain 17, the high byte.

1.7 x86 ADDRESSING MODES
based relative addressing mode

- In *based relative addressing mode*, base registers BX & BP, and a displacement value, are used to calculate the *effective address*.
  - Default segments used for the calculation of the physical address (PA) are DS for BX and SS for BP.

```
MOV CX,[BX]+10      ;move DS:BX+10 and DS:BX+10+1 into CX
                    ;PA = DS (shifted left) + BX + 10
```

  - Alternatives are "**MOV CX,[BX+10]**" or "**MOV CX,10[BX]**"
    - Again the low address contents will go into CL and the high address contents into CH.

1.7 x86 ADDRESSING MODES
based relative addressing mode

- In the case of the BP register:

```
MOV   AL,[ BP] +5     ;PA = SS (shifted left) + BP + 5
```

  - Alternatives are "**MOV AL,[BP+5]**" or "**MOV AL,5[BP]**".
    - **BP+5** is called the *effective address* since the fifth byte from the beginning of the offset BP is moved to register AL.

1.7 x86 ADDRESSING MODES
indexed relative addressing mode

- The *indexed relative addressing* mode works the same as the based relative addressing mode.
  - Except that registers DI & SI hold the offset address.

```
MOV   DX,[ SI] +5     ;PA = DS (shifted left) + SI + 5
MOV   CL,[ DI] +20    ;PA = DS (shifted left) + DI + 20
```

1.7 x86 ADDRESSING MODES
indexed relative addressing mode

- The *indexed relative addressing* mode works the same as the based relative addressing mode.
  - Except that registers DI & SI hold the offset address.

Example 1-17

Assume that DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP = 7814, and AX = 2512. All values are in hex. Show the exact physical memory location where AX is stored in each of the following. All values are in hex.

(a) `MOV[ BX] +20,AX` (b) `MOV[ SI] +10,AX`

(c) `MOV[ DI] +4,AX` (d) `MOV[ BP] +12,AX`

Solution:

In each case PA = segment register (shifted left) + offset register + displacement.

(a) DS:BX+20 location 47120 = (12) and 47121 = (25)

(b) DS:SI+10 location 46496 = (12) and 46497 = (25)

(c) DS:DI+4 location 4D504 = (12) and 4D505 = (25)

(d) SS:BP+12 location 27826 = (12) and 27827 = (25)

1.7 x86 ADDRESSING MODES
based indexed addressing mode

- By combining based & indexed addressing modes, a new addressing mode is derived called the *based indexed addressing mode.*
  - One base register and one index register are used.

```
MOV   CL,[BX][DI]+8   ;PA = DS (shifted left) + BX + DI + 8
MOV   CH,[BX][SI]+20  ;PA = DS (shifted left) + BX + SI + 20
MOV   AH,[BP][DI]+12  ;PA = SS (shifted left) + BP + DI + 12
MOV   AH,[BP][SI]+29  ;PA = SS (shifted left) + BP + SI + 29
```

  - The coding of the instructions can vary.

Table 1-5: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

1.7 x86 ADDRESSING MODES
segment overrides

- The x86 CPU allows the program to override the default segment and use any segment register.
  - In "**`MOV AL,[BX]`**", the physical address of the operand to be moved into AL is DS:BX.
    - To override that default, specify the desired segment in the instruction as "**`MOV AL,ES:[BX]`**"

Table 1-6: Sample Segment Overrides

Instruction	Segment Used	Default Segment
MOV AX, CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

1.7 x86 ADDRESSING MODES
summary

Table 1-7: Summary of the x86 Addressing Modes

Addressing Mode	Operand	Default Segment
Register	reg	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	DS
Based indexed relative	[BX][SI]+disp	DS
	[BX][DI]+disp	DS
	[BP][SI]+disp	SS
	[BP][DI]+disp	SS

The x86 PC

assembly language, design, and interfacing

fifth edition

Prentice Hall

The x86 PC

assembly language, design, and interfacing

fifth edition

MUHAMMAD ALI MAZIDI
JANICE GILLISPIE MAZIDI
DANNY CAUSEY