Unit 03 – Computer Programming

COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

Machine Language vs. Assembly Language

Machine vs. Assembly Language

- **Machine language** or **Object Code** is the only code a computer an execute
- It is nearly impossible for a human to work with

Example the Object Code:

- **E4 27 88 C3 E4 27 00 D8 E6 30 F4**

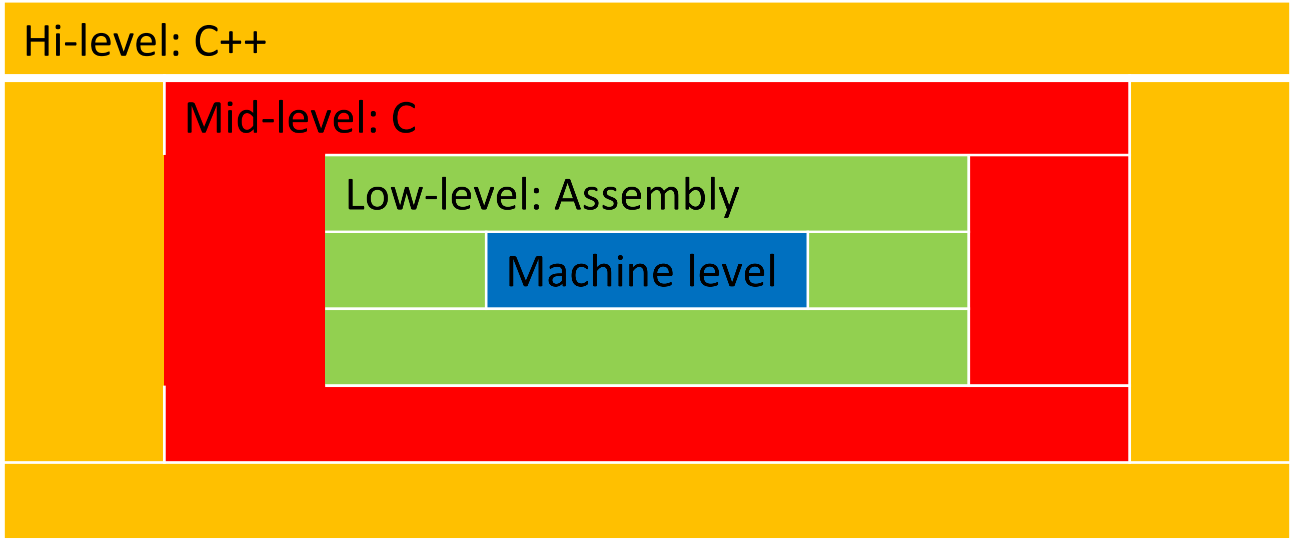
- Object code for adding two numbers inputted from the keyboard
- Are usually hexadecimal values to represent the instruction and data

- Programmers often use **Assembly Language** when programming a microprocessor
- Involves 3 to 5 letter abbreviations to represent the object codes or instruction codes
- These are called **mnemonics**
- **Mnemonics** are divided into operational codes (**op-codes**) and Operands

Machine Code					Assembly Code		
					Mnemonics		
Address	Hex Object Code				Opcode	Operands	Description
0100	E4	27			IN	AL, 27H	Get value from port 27H and store to AL register
0102	88	C3			MOV	BL, AL	Copy AL contents to BL register
0104	E4	27			IN	AL, 27H	Get next value from port 27H and store to AL register
0106	00	D8			ADD	AL, BL	Add contents of BL to AL and store result to AL
0107	E6	30			OUT	30H, AL	Output AL result to port 30H
0109	F4				HLT		Halt the operation

Types of Software

- System Software
- Operating System Software
- Application Software
- Programming Software
  - Programming Languages
    - Hi-level – Pascal, C++, Java
    - Mid-level - C
    - Low-level – Assembly



- Conversion of HLL to ML done by **compiler** or **interpreter**
- Conversion of LL to ML done by **assembler**

Assembly language

- Applies a one-to-one relationship between assembly and machine language instructions
- Using a high-level language to compile machine code results in an in-efficient code
- Results to more machine language instructions than assembled version from purely written assembly language code
- Key benefits of assembly language
    - Takes up less memory
    - Executes much faster
    - Used in real-time applications
- **Real-time** means the task required by the application must be completed before any other input to the program will alter its operation

Edit, Assemble, Test, and Debug Cycle

Edit, Assemble, Test, and Debug Cycle

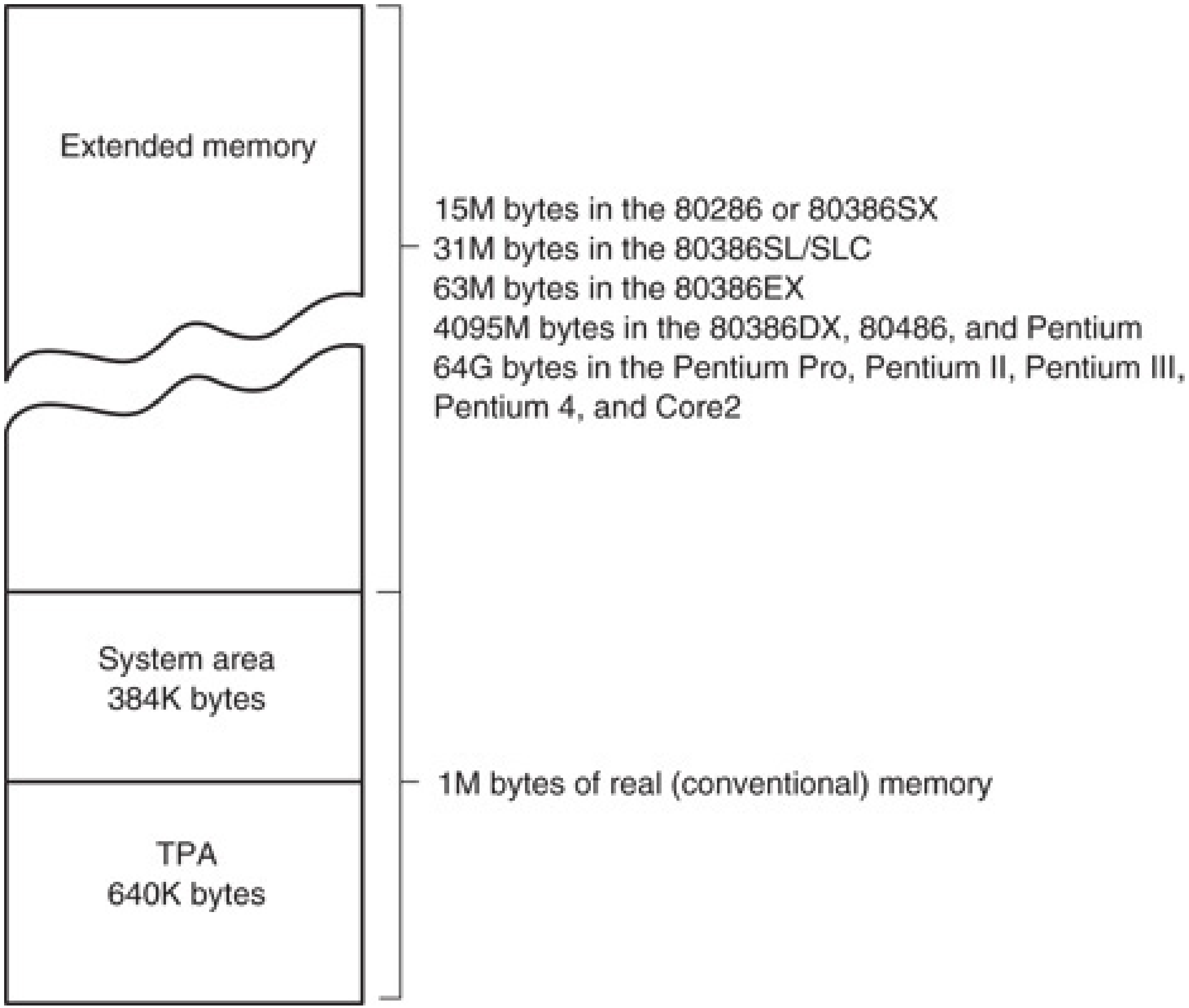
- Using an **editor**, the source code of the program is created.
- This means selecting the appropriate instruction mnemonics to accomplish the task.
- A compiler program which examines the source code file generated by the editor and determines the object code for each instruction in the program, is then run.
- In assembly language programming, this is called an **assembler** (MASM, TASM, A86, EMU86).
- The object code produced by the computer is loaded into the target computer's memory and is then run.
- **Debugging** refers to locating and fixing the source of the error
- Mid to High level programming Languages uses a **compiler** or an **interpreter**
  - Compiler: Pascal, C, C++
  - Interpreter: Java, VB, Basic

Memory Address Space and Organization

Memory Map of a PC

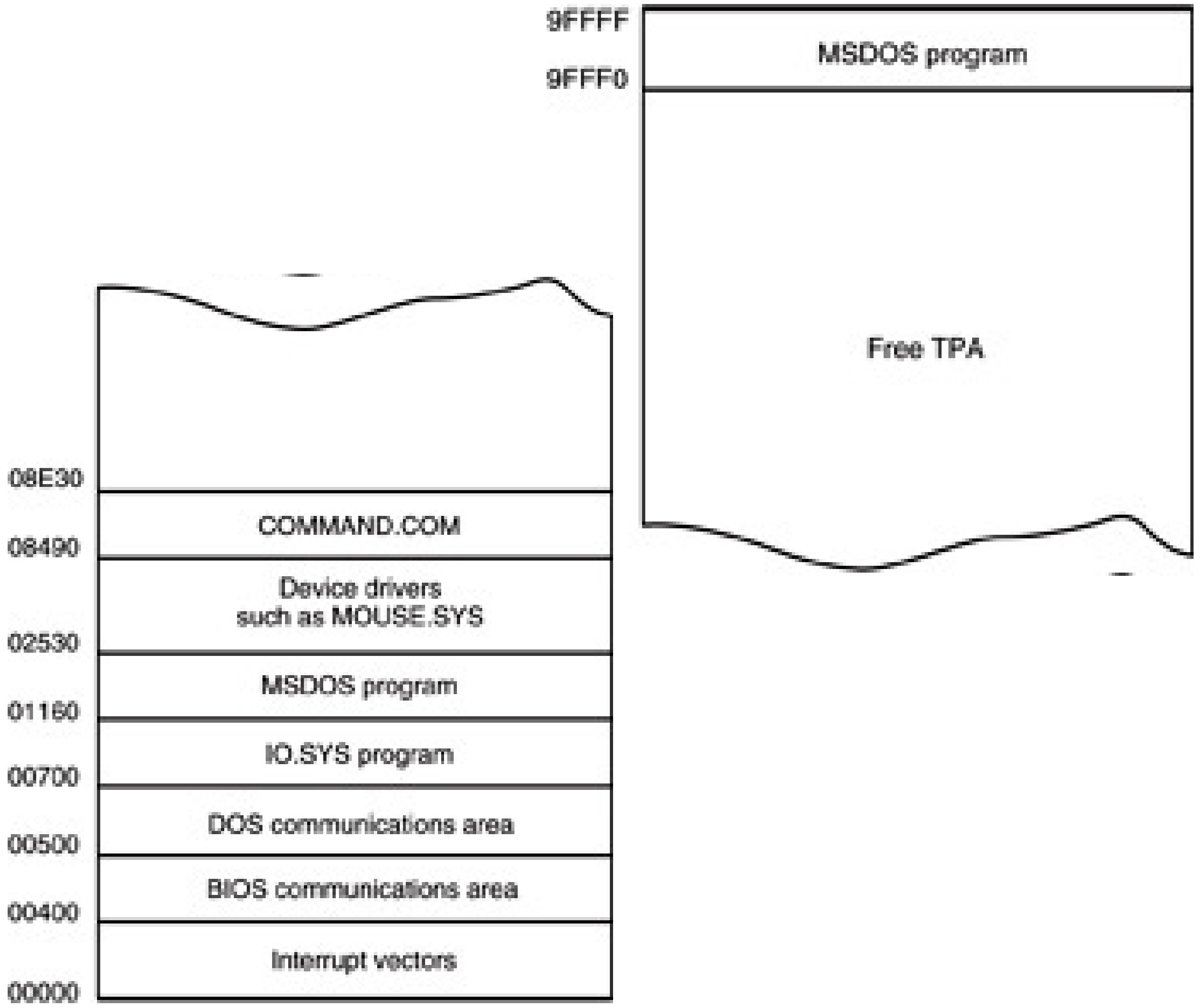
Transient Program Area (TPA)

- holds the DOS (**disk operating system**) operating system;
- other programs that control the computer system.
- TPA is a DOS concept and not really applicable in Windows
- also stores any currently active or inactive DOS application programs
- length of the TPA is 640K bytes


Extended memory
15M bytes in the 80286 or 80386SX
31M bytes in the 80386SL/SLC
63M bytes in the 80386EX
4095M bytes in the 80386DX, 80486, and Pentium
64G bytes in the Pentium Pro, Pentium II, Pentium III, Pentium 4, and Core2
System area
384K bytes
TPA
640K bytes
1M bytes of real (conventional) memory

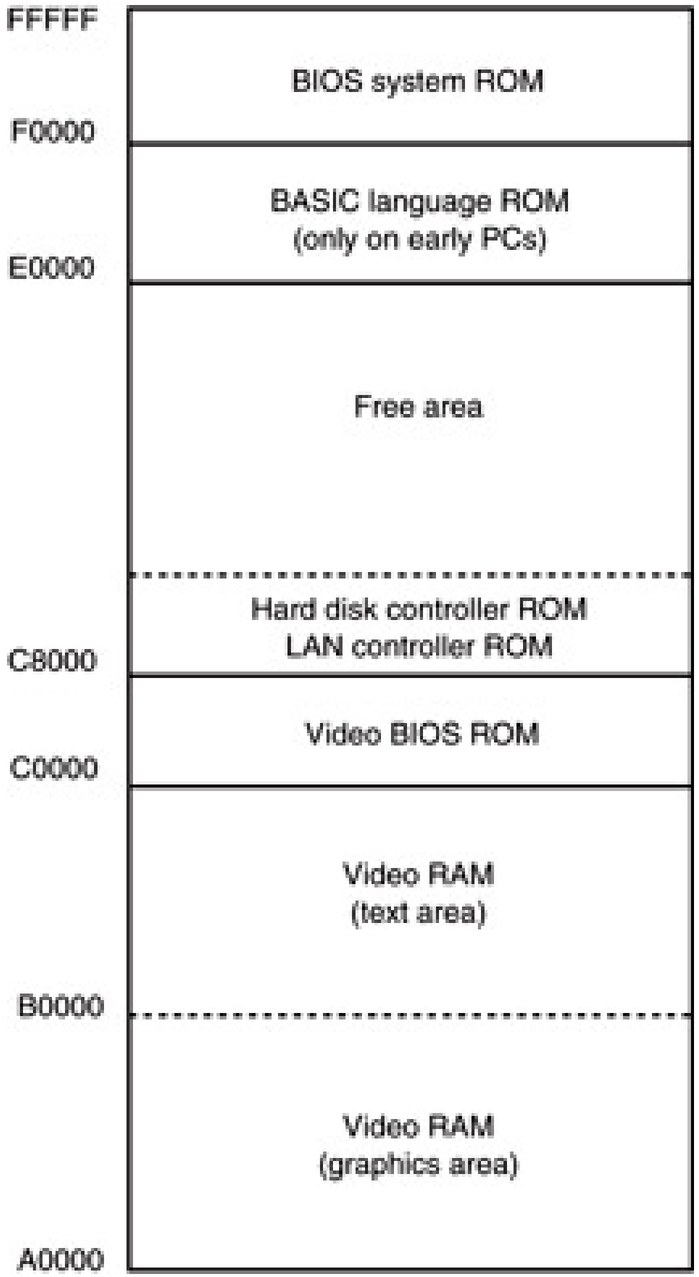
TPA area

- DOS memory map shows how areas of TPA are used for system programs, data and drivers.
- Shows a large area of memory available for application programs (free TPA area)
- Hexadecimal number to left of each area represents the memory addresses that begin and end each data area

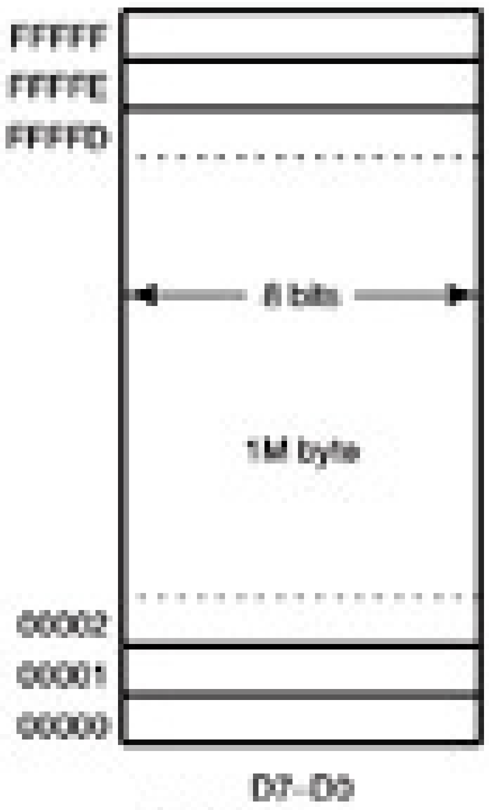


8086 SYSTEM area

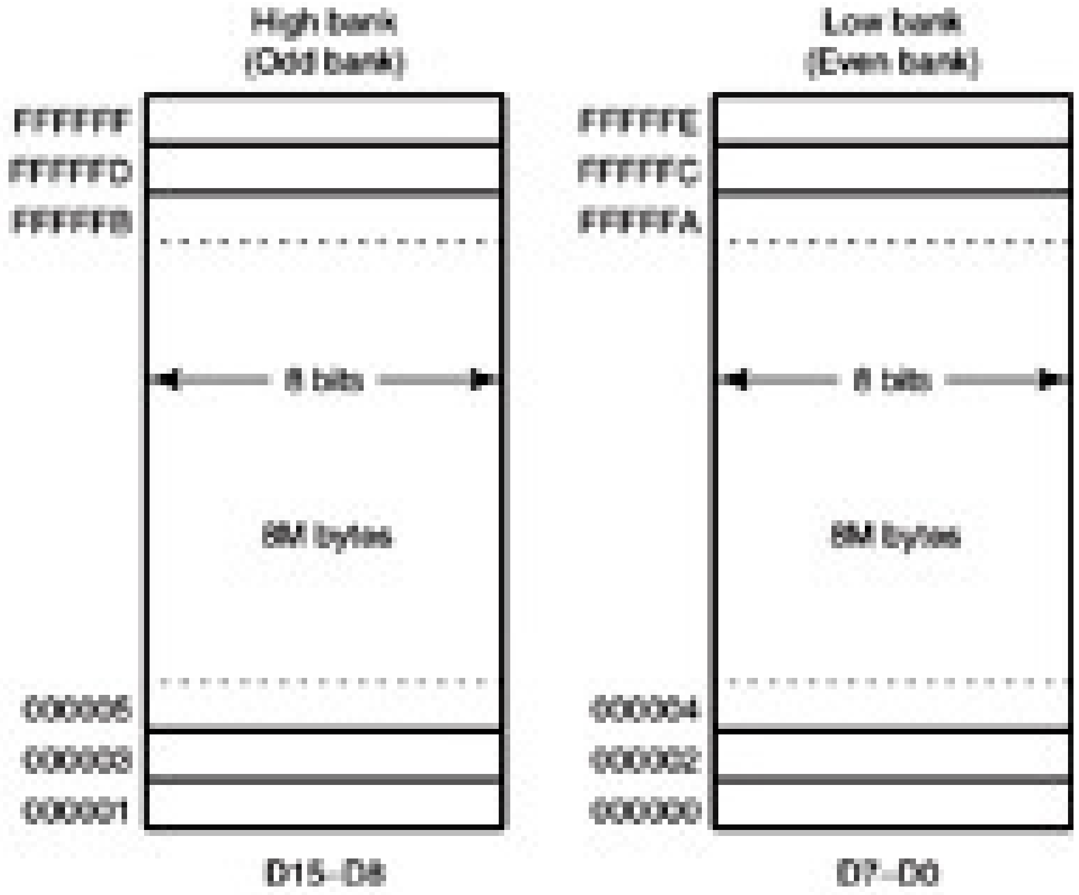
- First area of system space contains video display RAM and video control programs on ROM or flash memory.
- Area starts at location A0000H and extends to C7FFFH
- Size/amount of memory depends on type of video display adapter attached



Physical memory systems of 8086 through Core2 microprocessors.



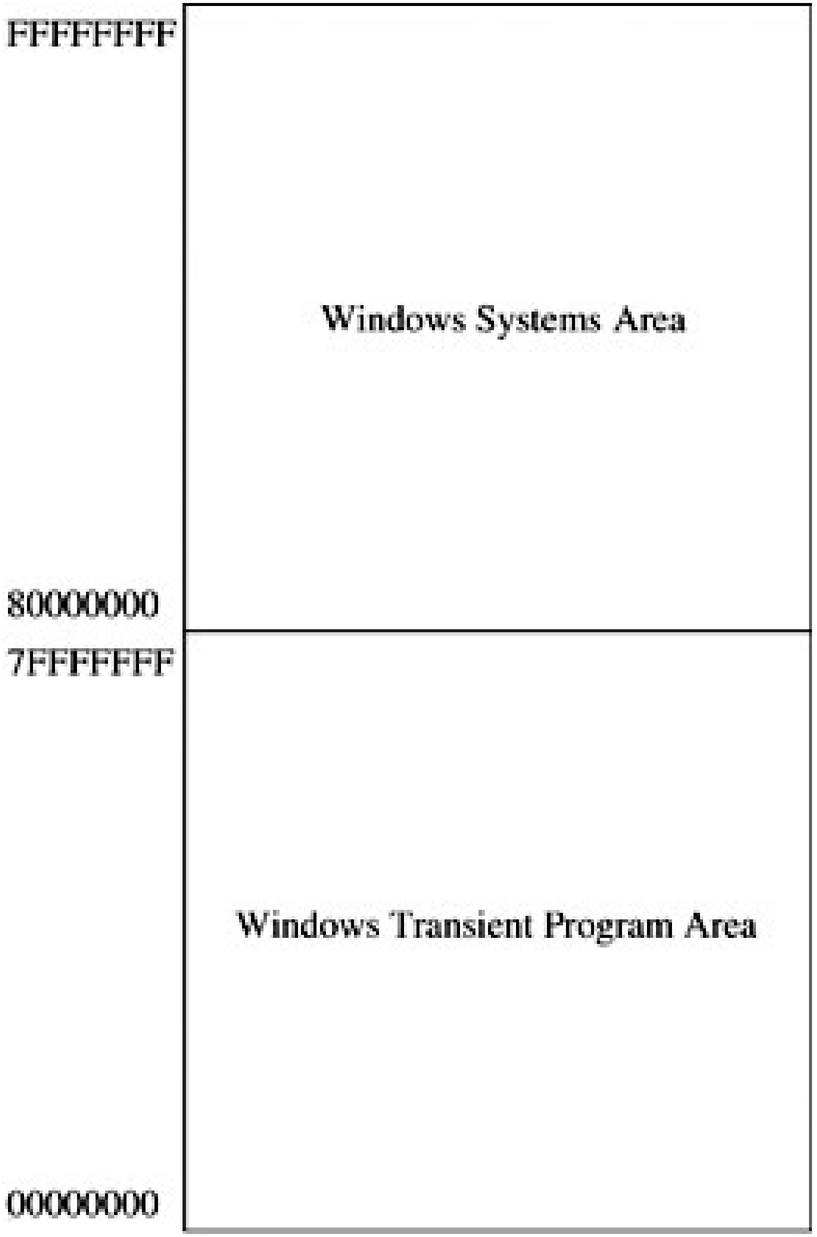
8088 microprocessor



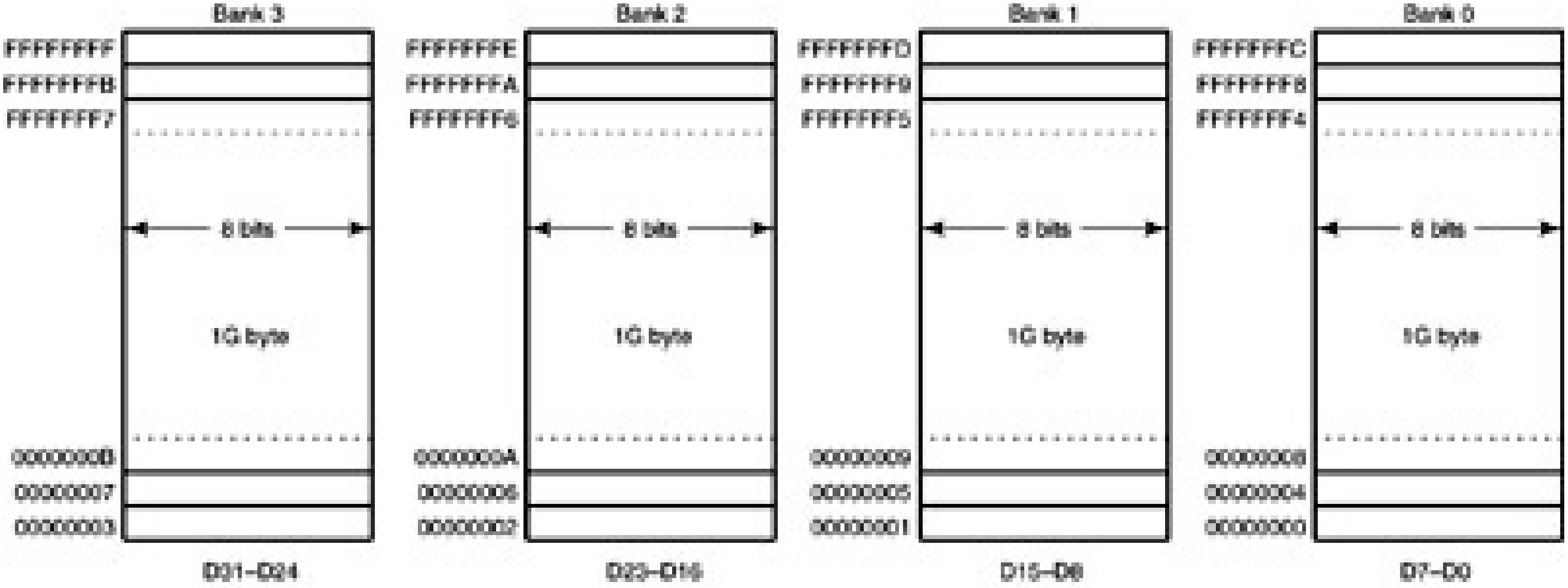
8086 microprocessor (memory is only 1M bytes)
80286 microprocessor
80386SX microprocessor
80386SL microprocessor (memory is 32M bytes)
80386SLC microprocessor (memory is 32M bytes)

Windows XP memory map

- TPA is first 2G bytes from locations 00000000H to 7FFFFFFFH.
- Every Windows program can use up to 2G bytes of memory located at linear addresses 00000000H through 7FFFFFFFH.
- System area is last 2G bytes from 80000000H to FFFFFFFFH.

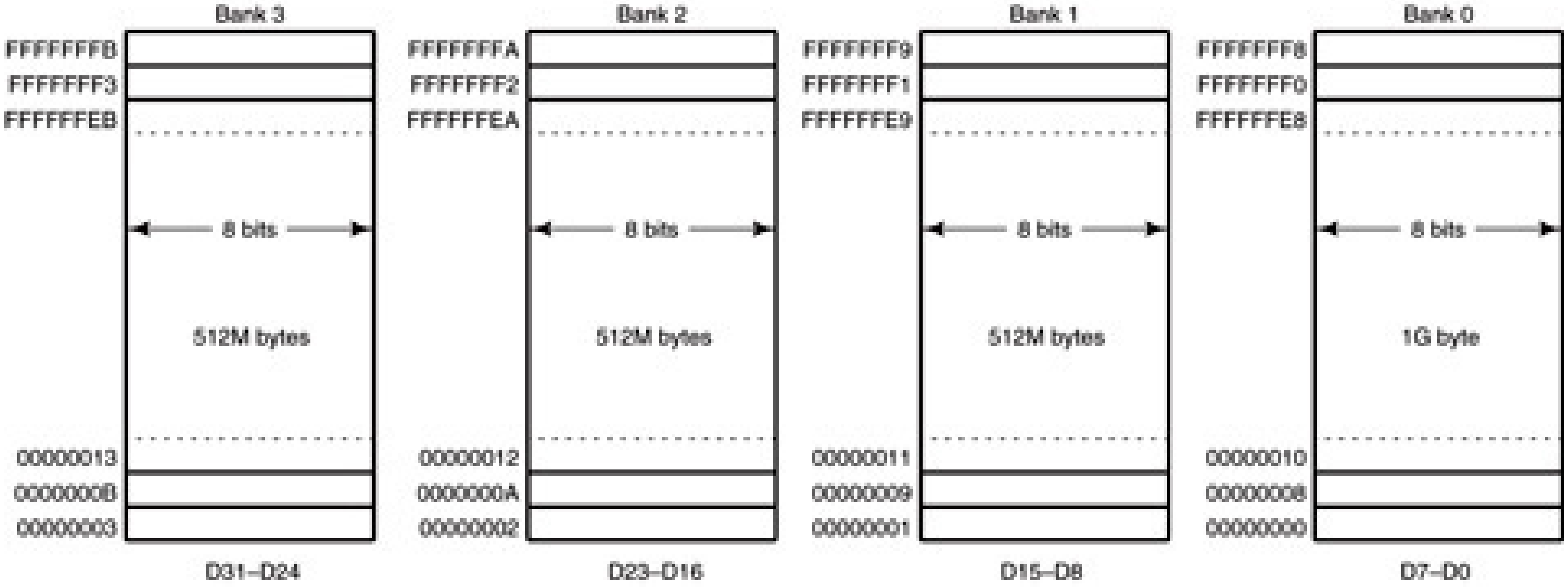


Physical Memory systems of 8086 through Core2 microprocessors.



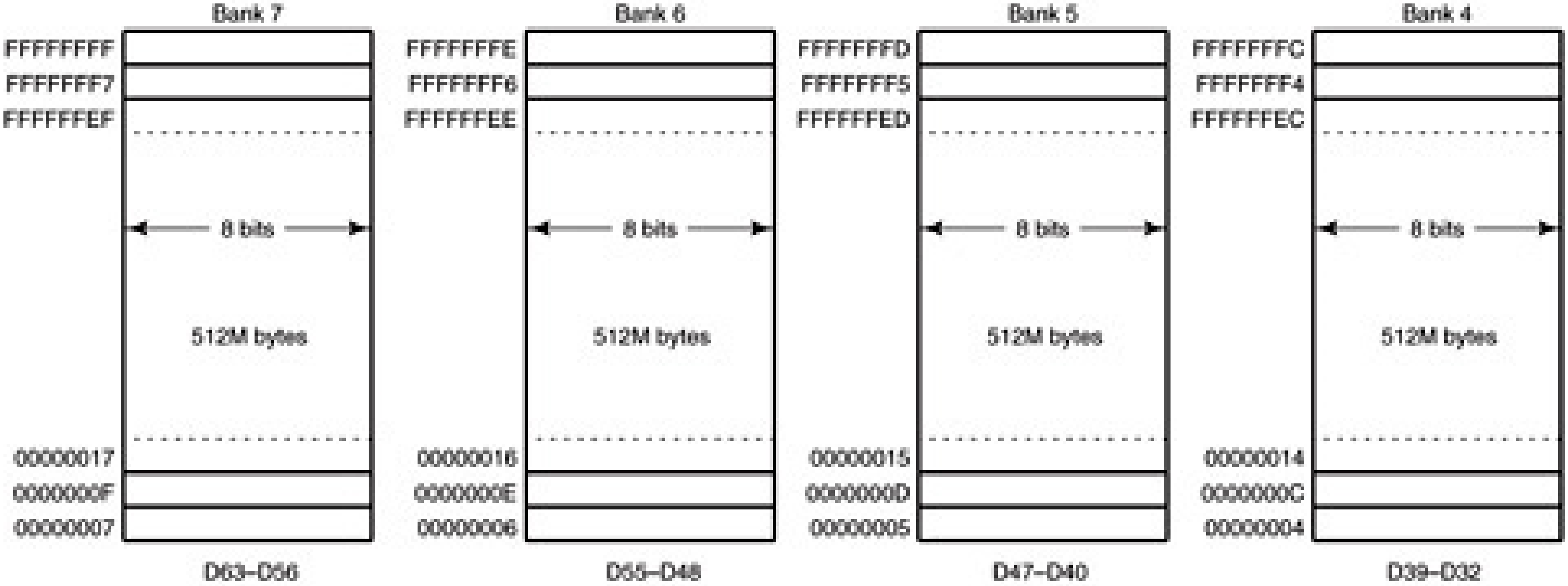
80386DX microprocessor
80486SX microprocessor
80486DX microprocessor

Physical Memory systems of 8086 through Core2 microprocessors.

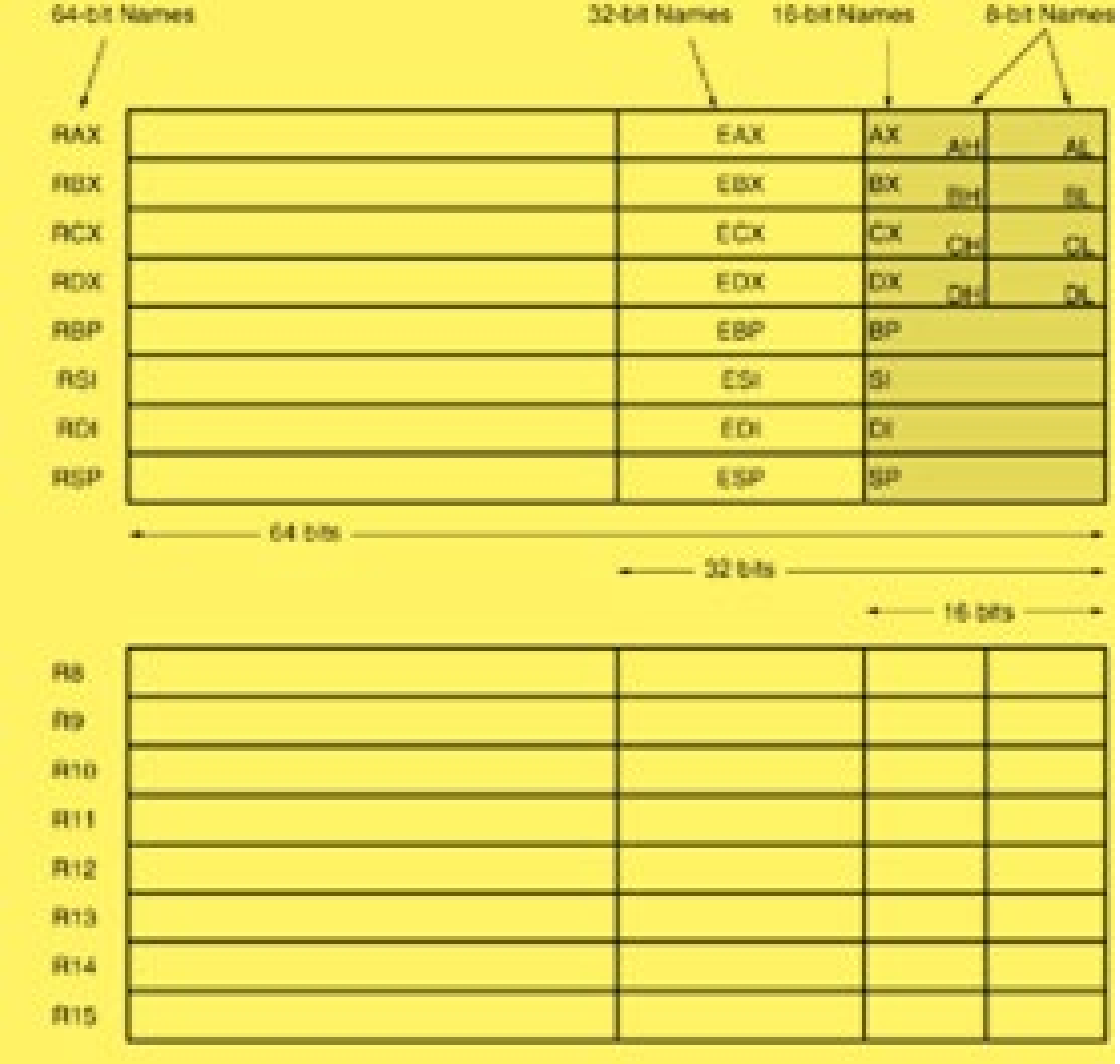


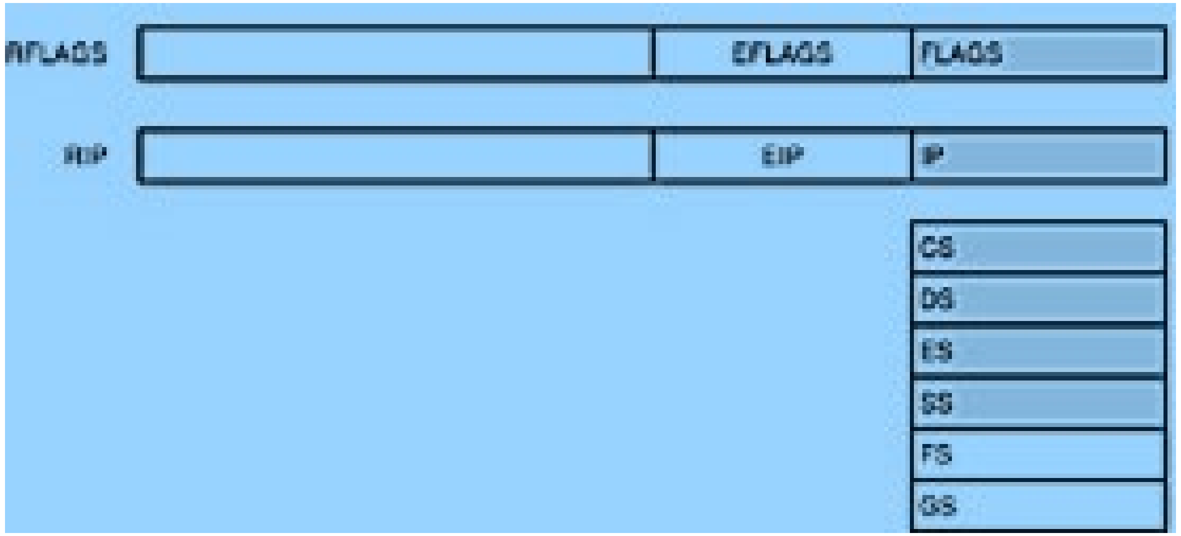
Pentium–Core2 microprocessors

Physical Memory systems of 8086 through Core2 microprocessors.



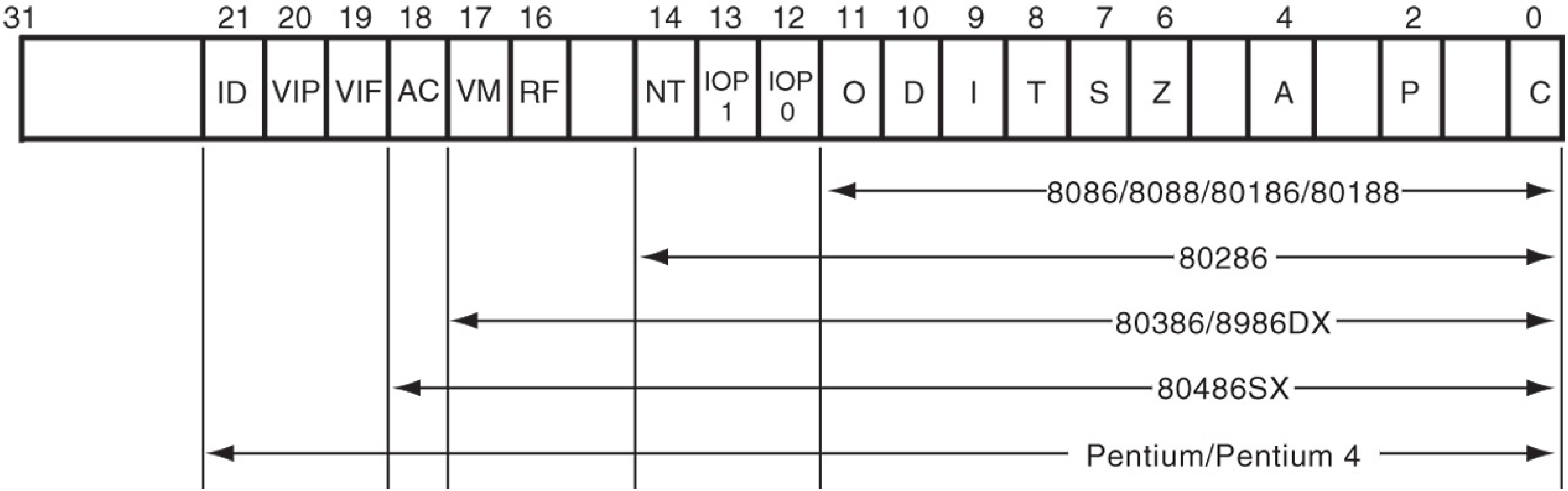
Register model of 8086 through Core2 microprocessor including 64-bit extensions





EFLAG and FLAG register counts for the entire 8086 and Pentium microprocessor family

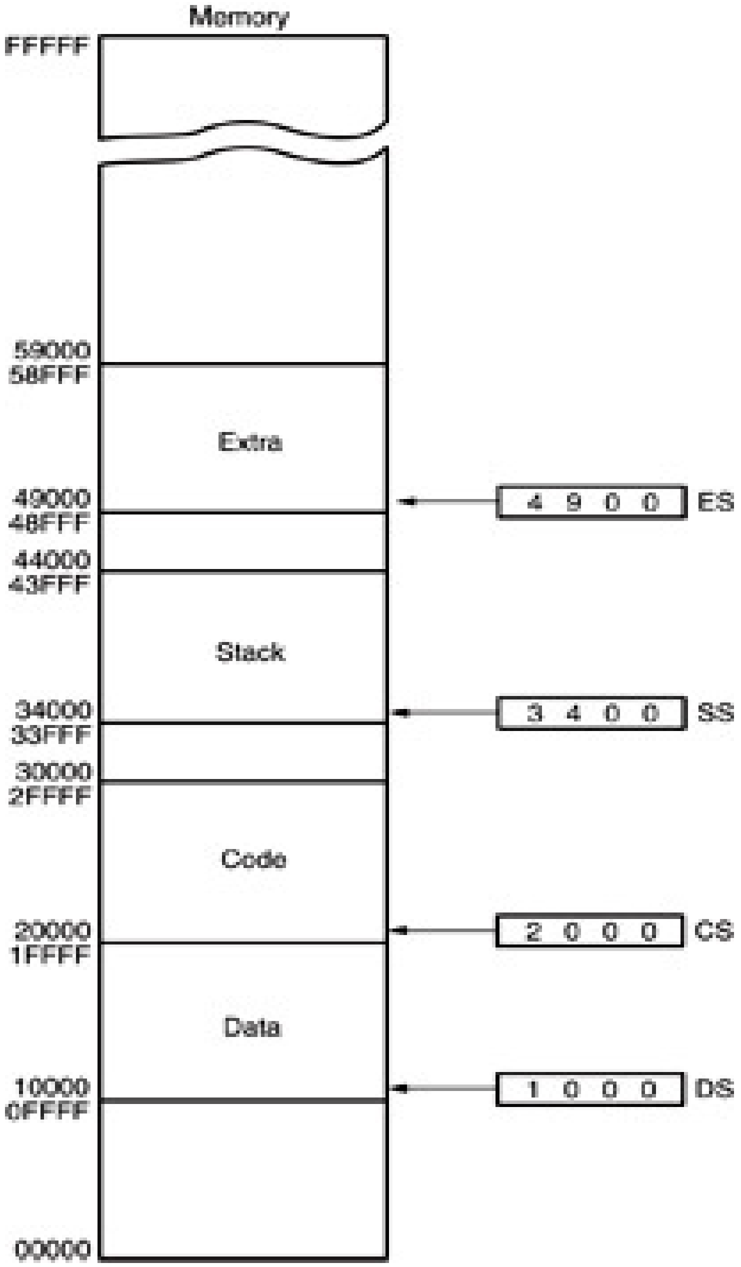
- Flags never change for any data transfer or program control operation.
- Some of the flags are also used to control features found in the microprocessor.



Memory Segmentation

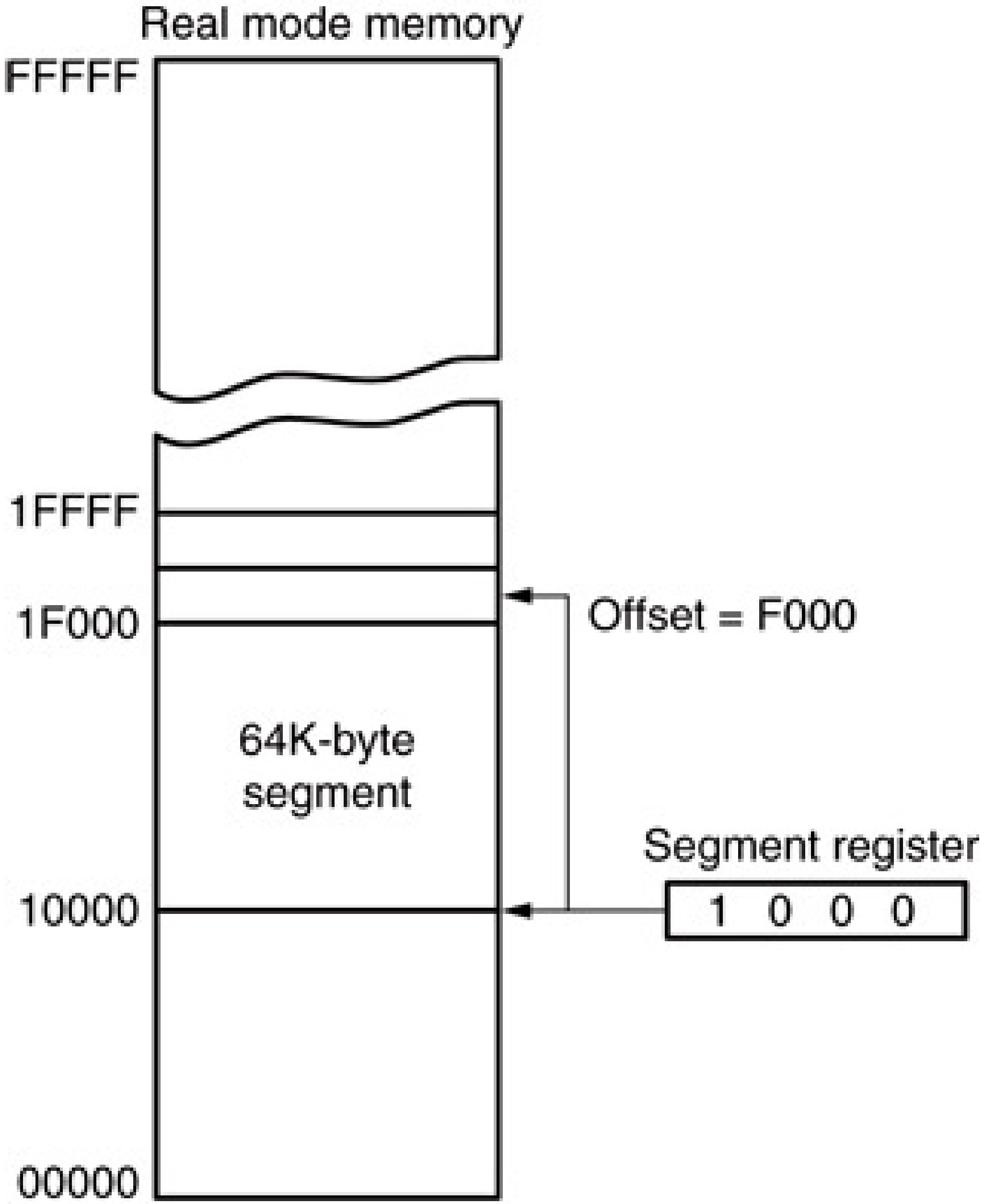
Memory Segmentation

- Think of segments as Windows that can be moved over any area of memory to access data or code
- A program can have more than four or six segments,
- But only access four or six segments at a time



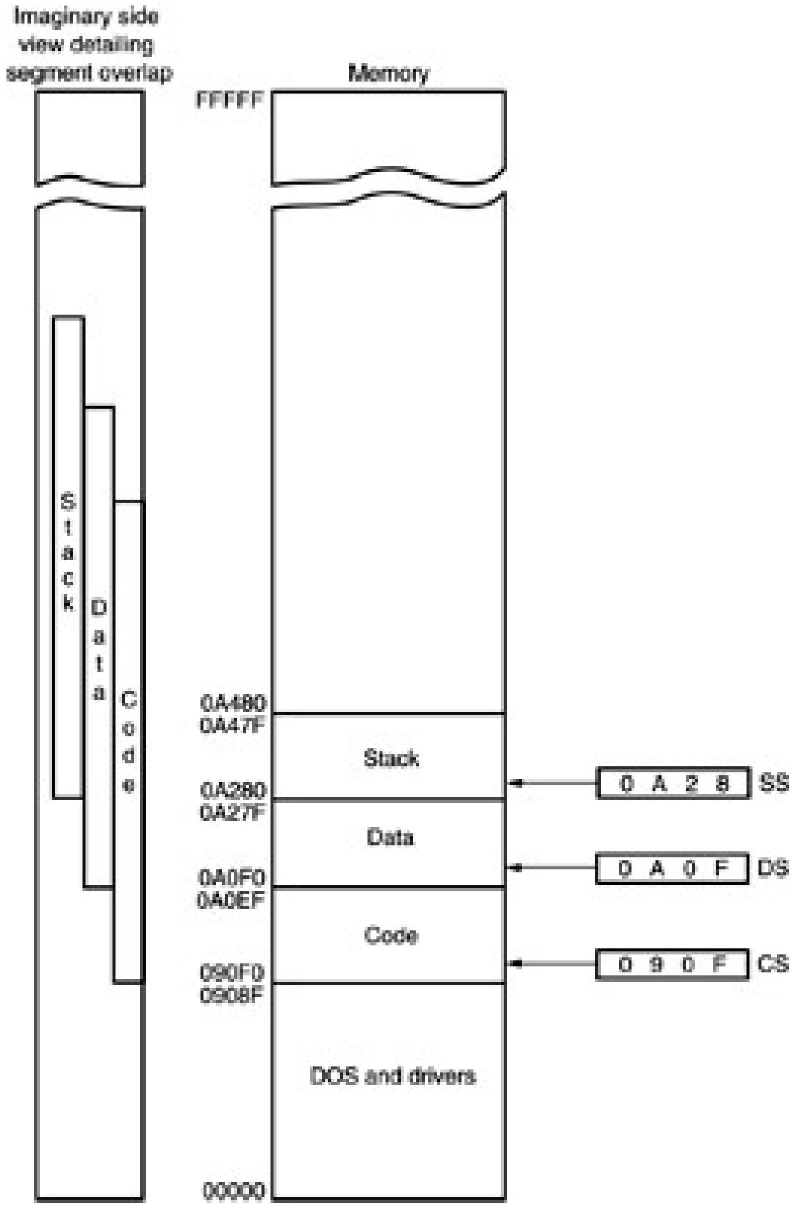
real mode memory-addressing scheme, segment + offset.

- this shows a memory segment beginning at **10000H**, ending at location **IFFFFH**
- 64K bytes in length
- also shows how an offset address, called a **displacement**, of **F000H** selects location **1F000H** in the memory

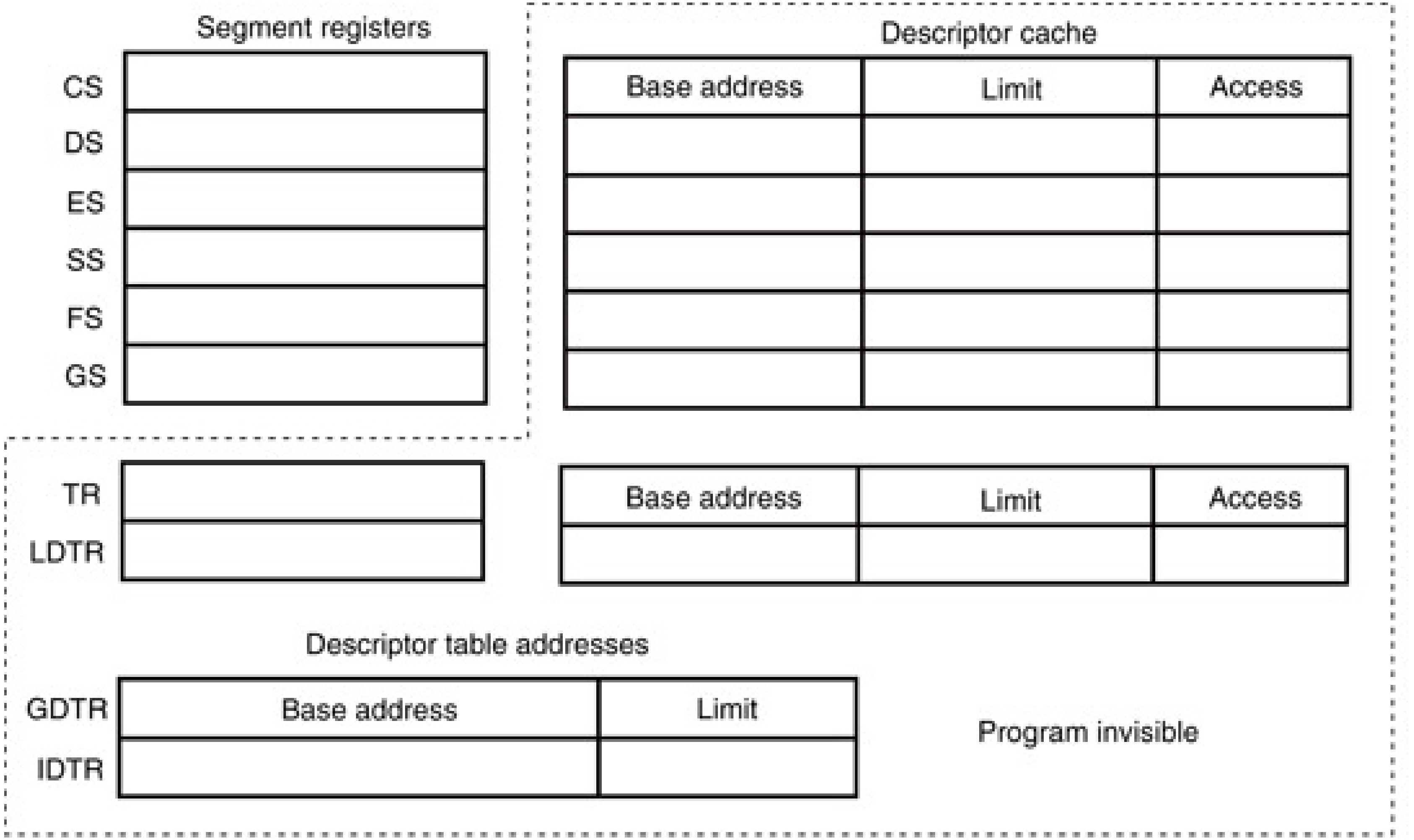


Application program containing a code, data, and stack segment loaded into a DOS system memory

- A program placed in memory by DOS is loaded in the TPA at the first available area of memory above drivers and other TPA programs
- Area is indicated by a **free-pointer** maintained by DOS
- Program loading is handled automatically by the **program loader** within DOS



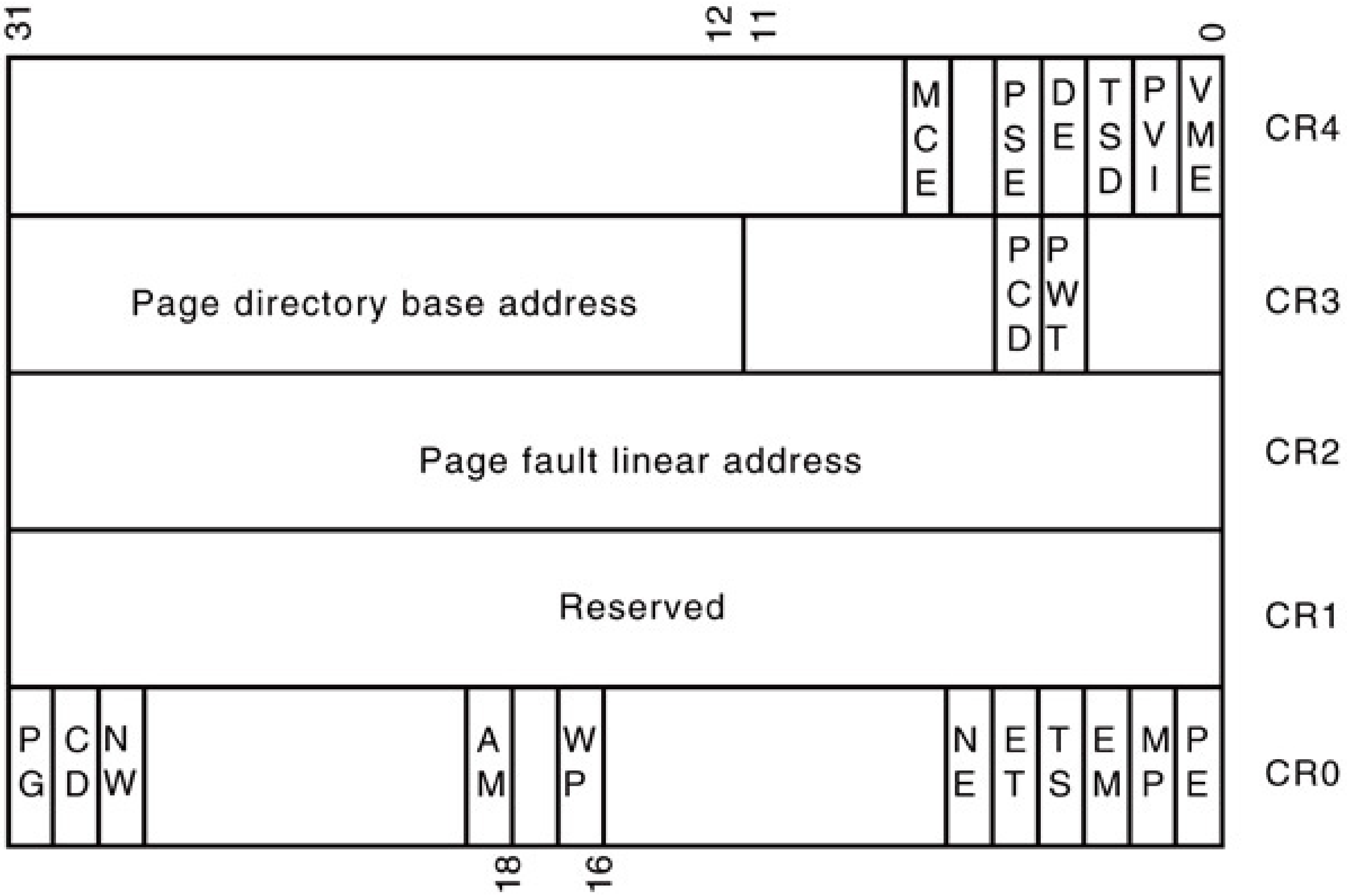
Program-invisible Register within 80286 to Core2 microprocessors



Notes:

1. The 80286 does not contain FS and GS nor the program-invisible portions of these registers.
2. The 80286 contains a base address that is 24-bits and a limit that is 16-bits.
3. The 80386/80486/Pentium/Pentium Pro contain a base address that is 32-bits and a limit that is 20-bits.
4. The access rights are 8-bits in the 80286 and 12-bits in the 80386/80486/Pentium–Core2.

Control Register structure of the microprocessor



31 12 11 0

MCE PSE DE TSD PVI VME CR4

Page directory base address PCD PWT CR3

Page fault linear address CR2

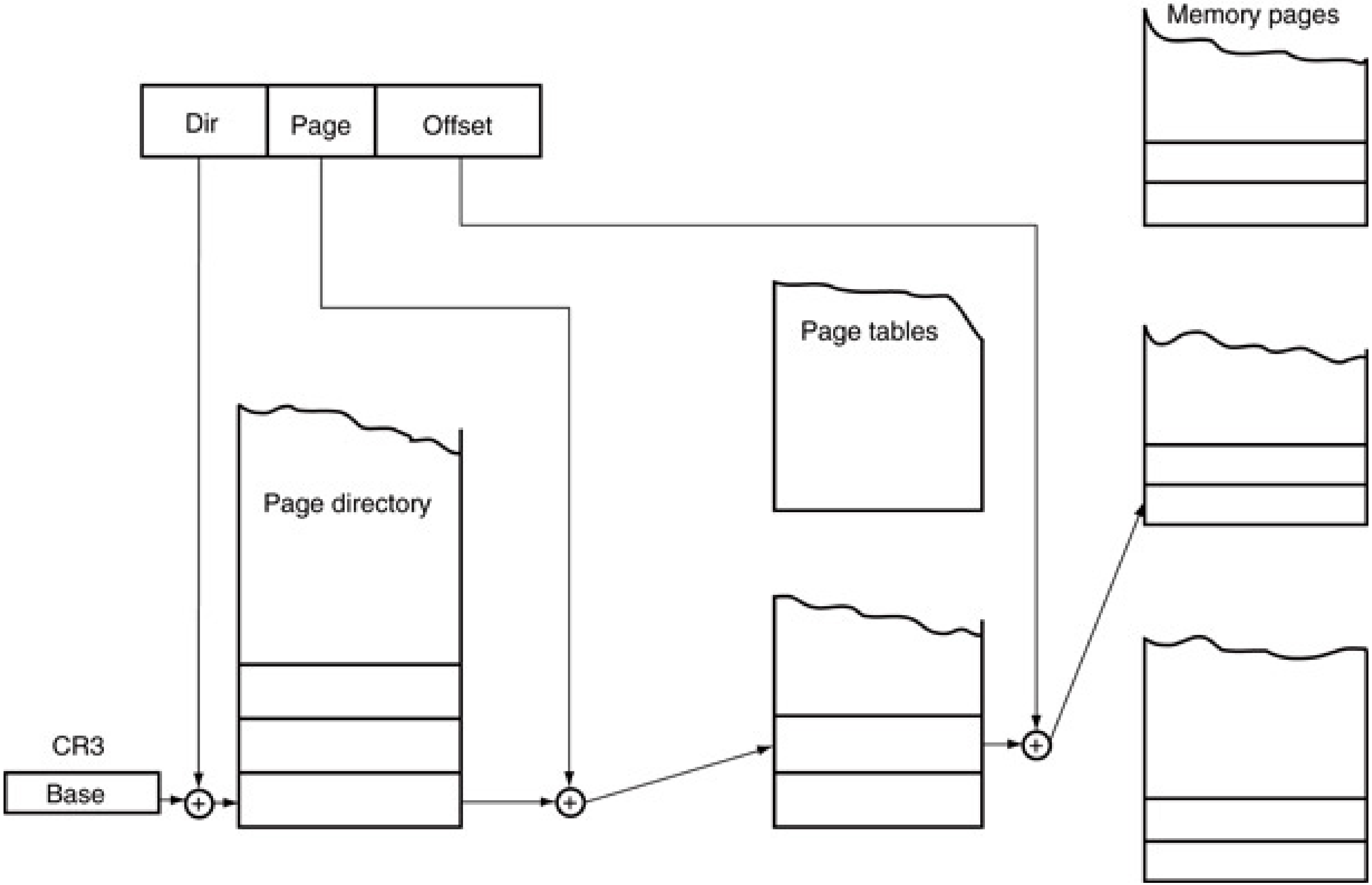
Reserved CR1

PG CD NW AM WP NE ET TS EM MP PE CR0

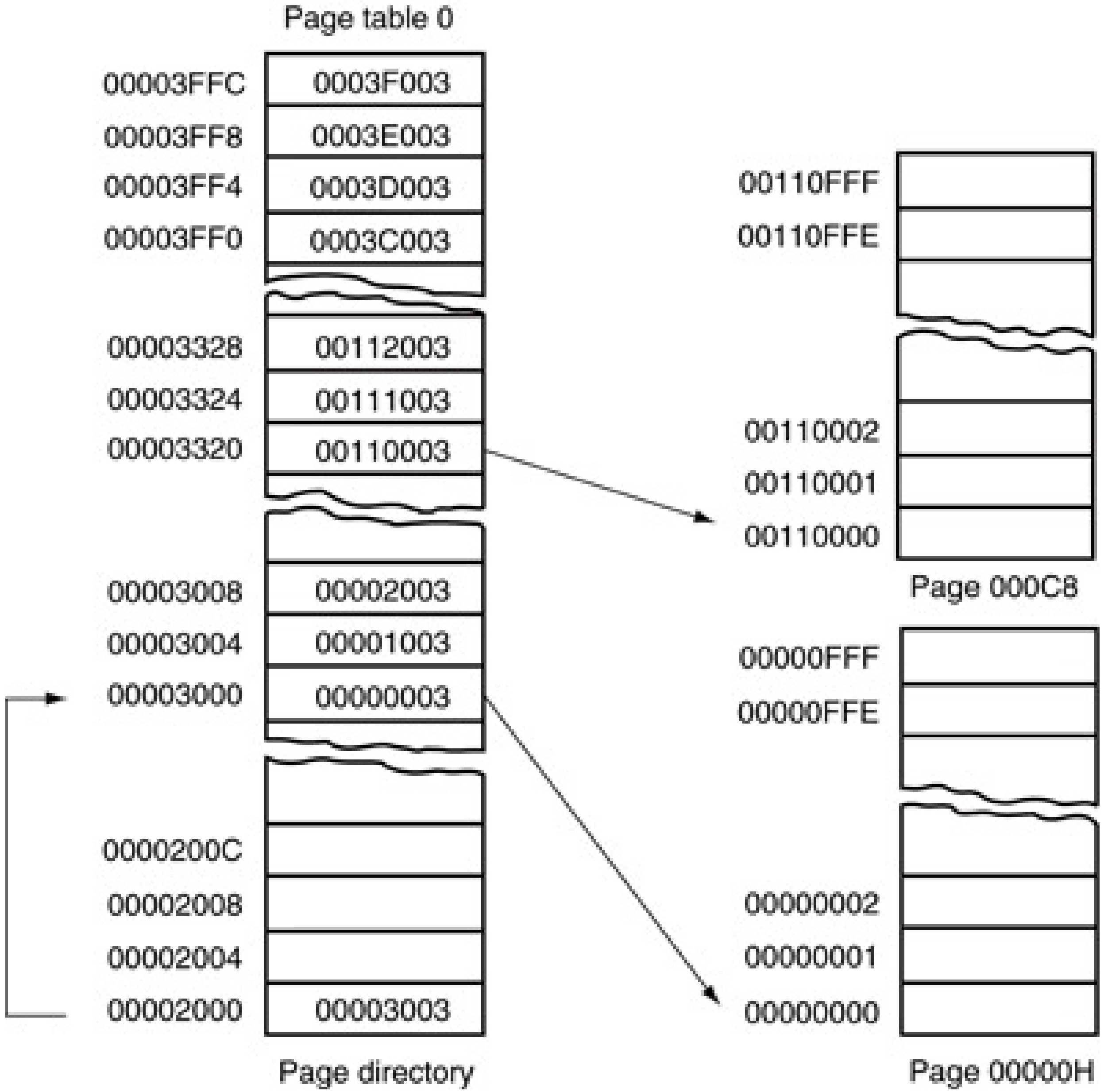
18 16

Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4 and Core2.

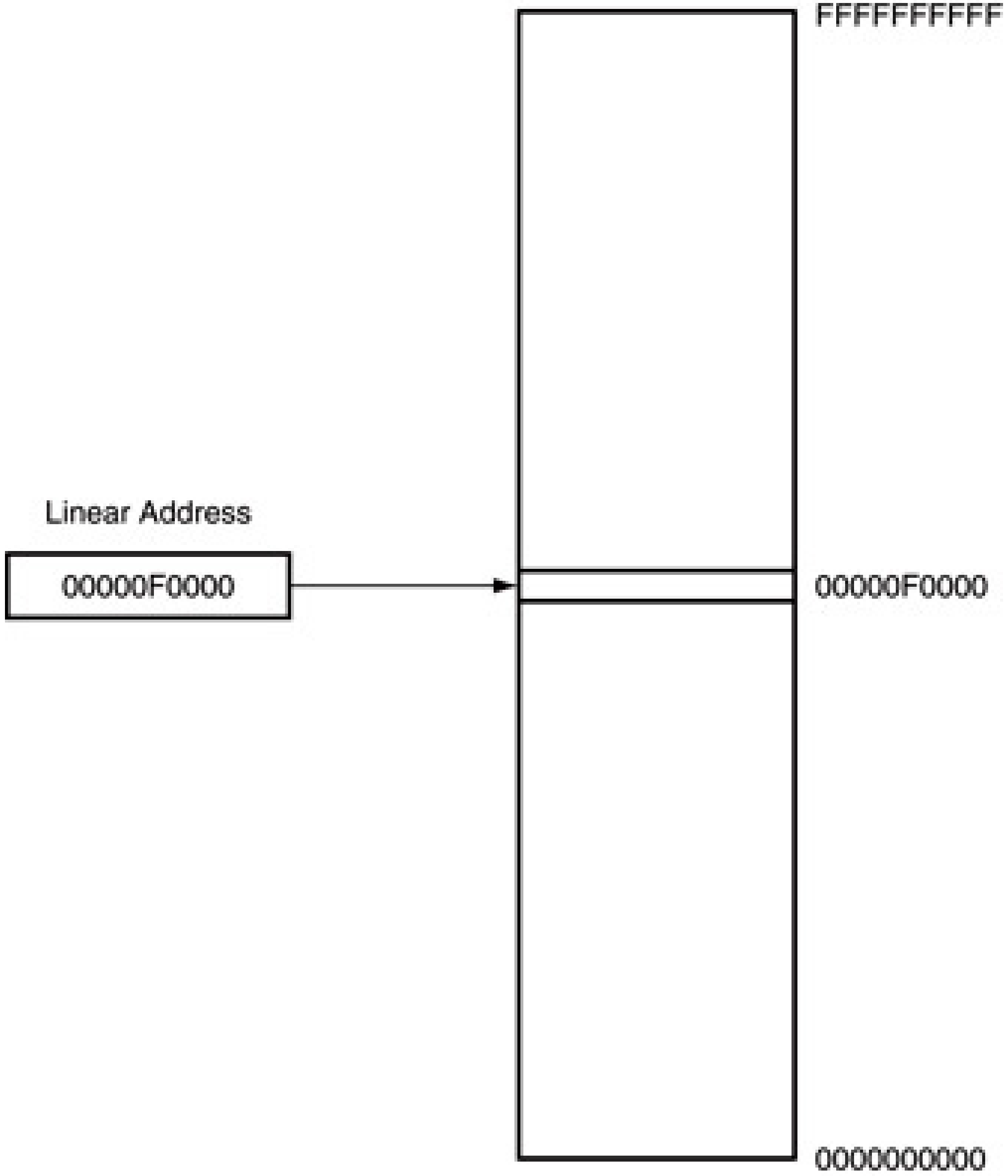
Paging mechanism in 80386 through Core2 microprocessors



Page directory, page table 0, and two memory pages



64-bit flat mode memory model

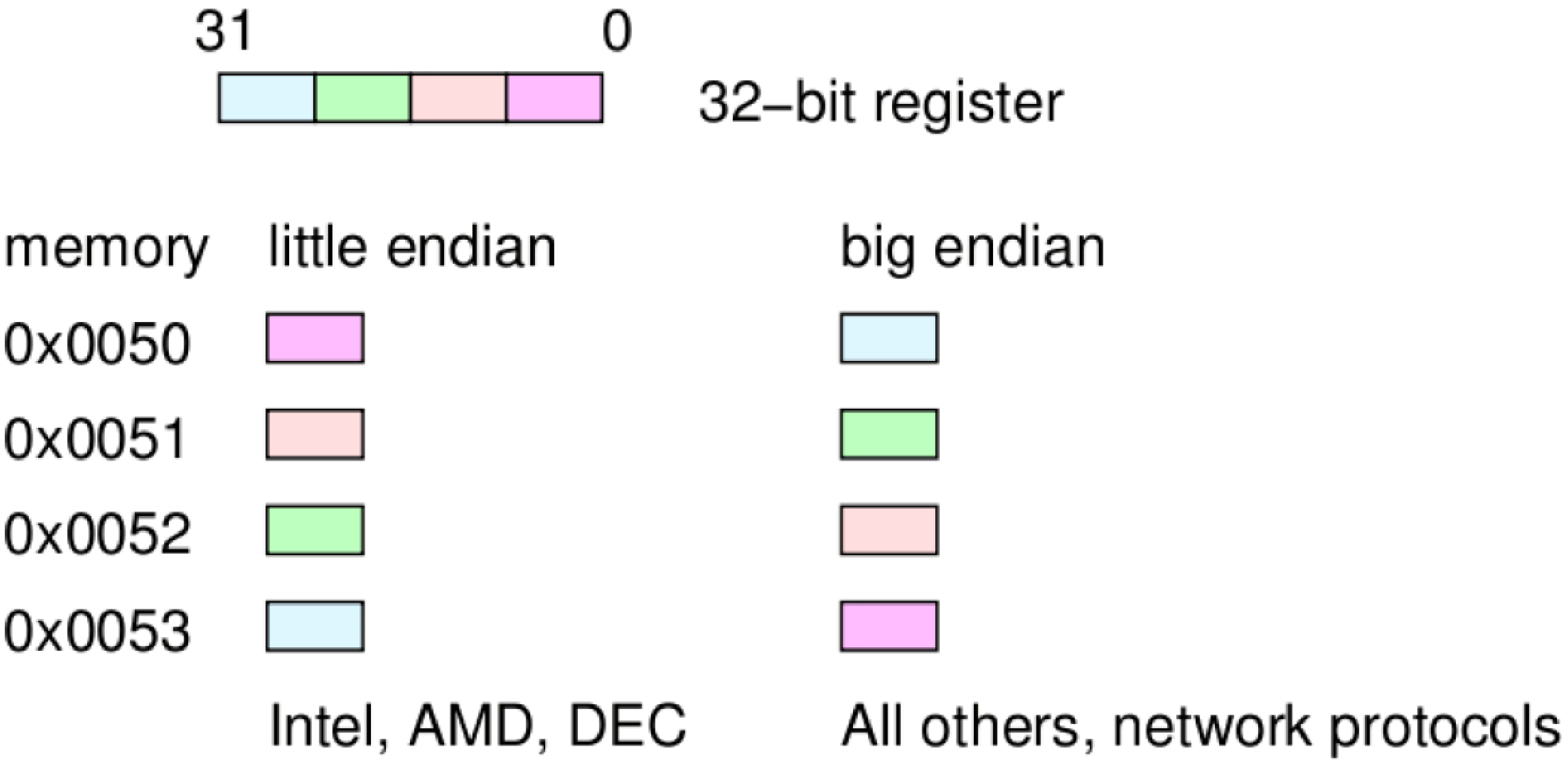


Little and Big Endian Convention

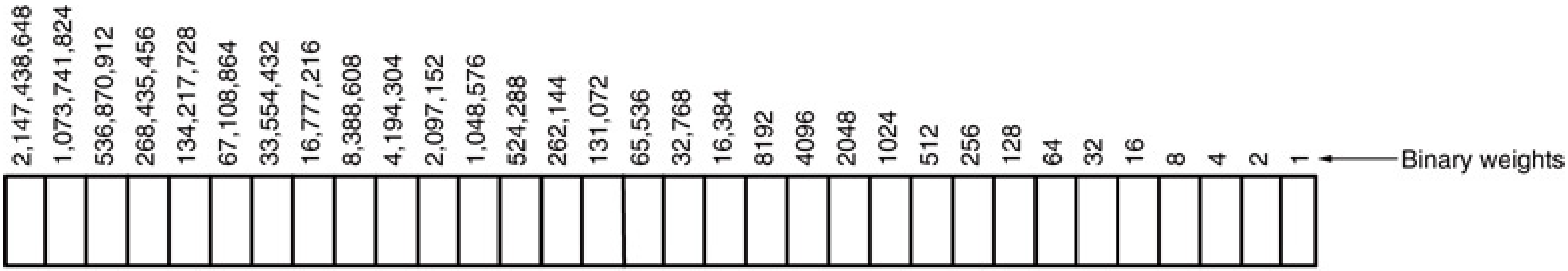
Little and Big Endian

Endianess

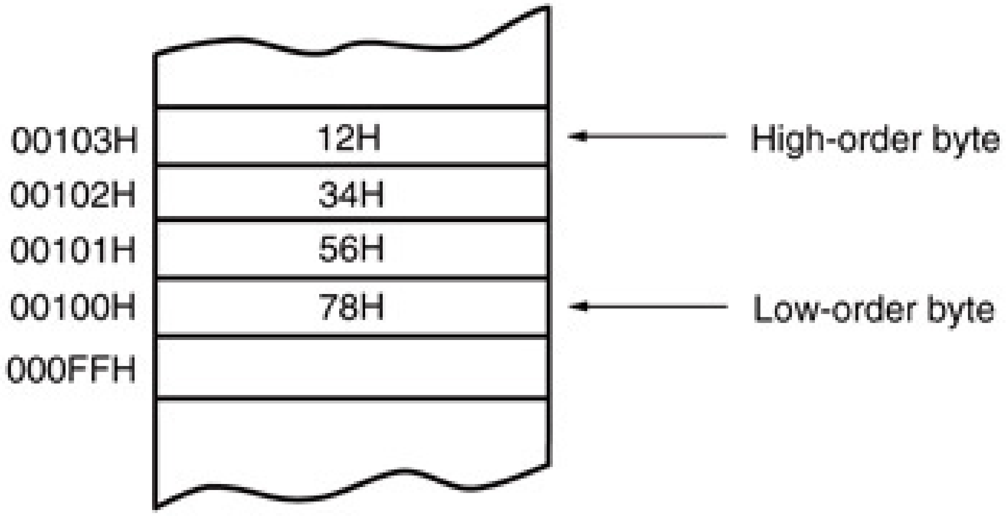
How to store integers with a size larger than 8 bits?



Little and Big Endian



(a) Unsigned doubleword

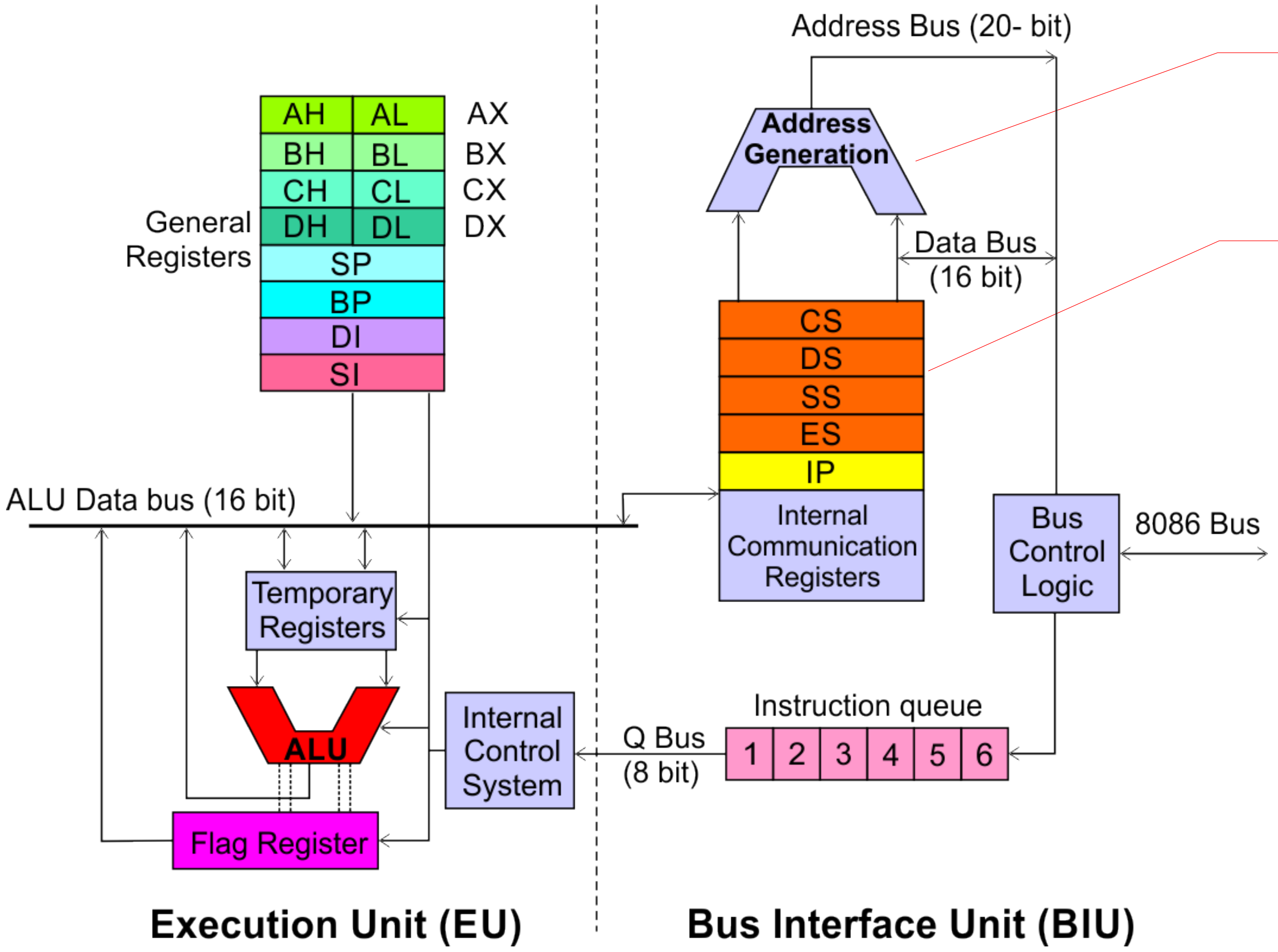


(b) The contents of memory location 00100H–00103H are the doubleword 12345678H.

Segment Registers

8086 Architecture

Bus Interface Unit (BIU)

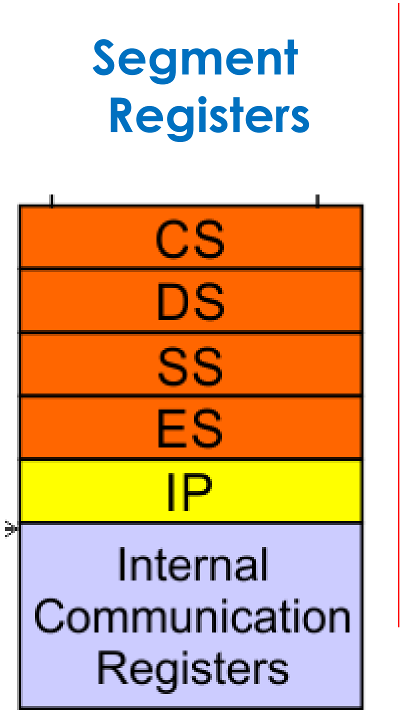


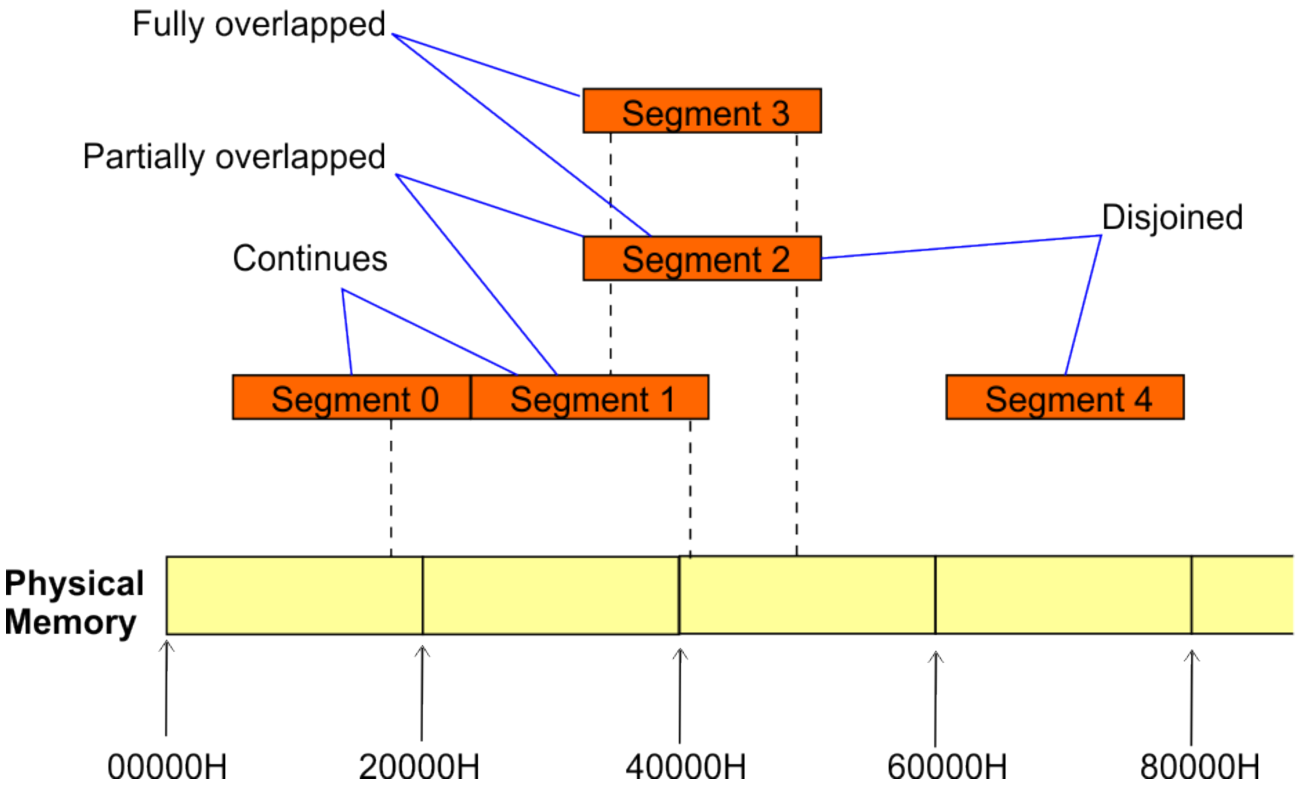
Dedicated Adder to generate 20 bit address

Four 16-bit segment registers

Code Segment (**CS**)
Data Segment (**DS**)
Stack Segment (**SS**)
Extra Segment (**ES**)

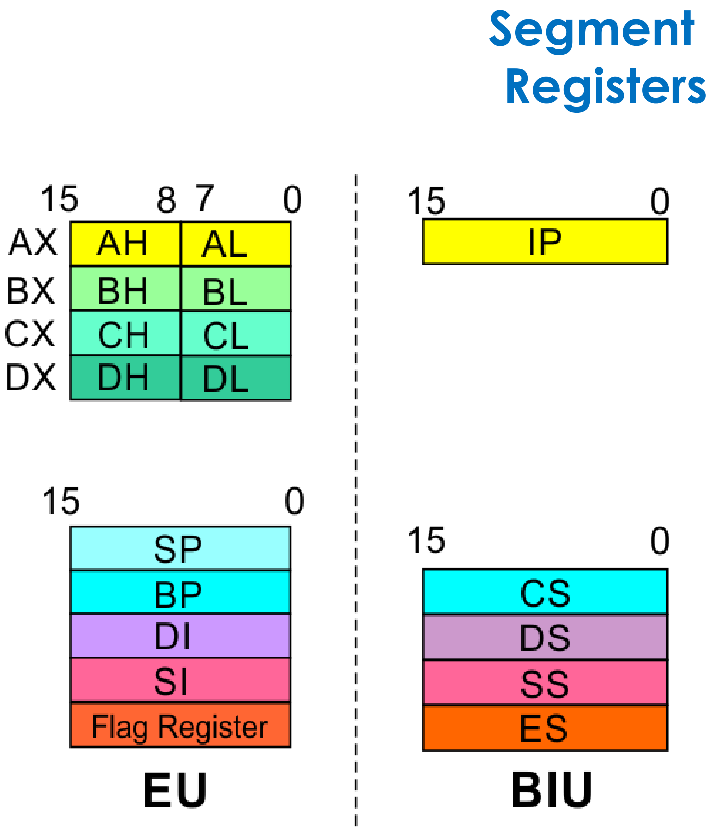
Bus Interface Unit (BIU)





- 8086's 1-megabyte memory is divided into segments of up to 64K bytes each.
- The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.
- Programs obtain access to code and data in the segments by changing the segment register content to point to the desired segments.

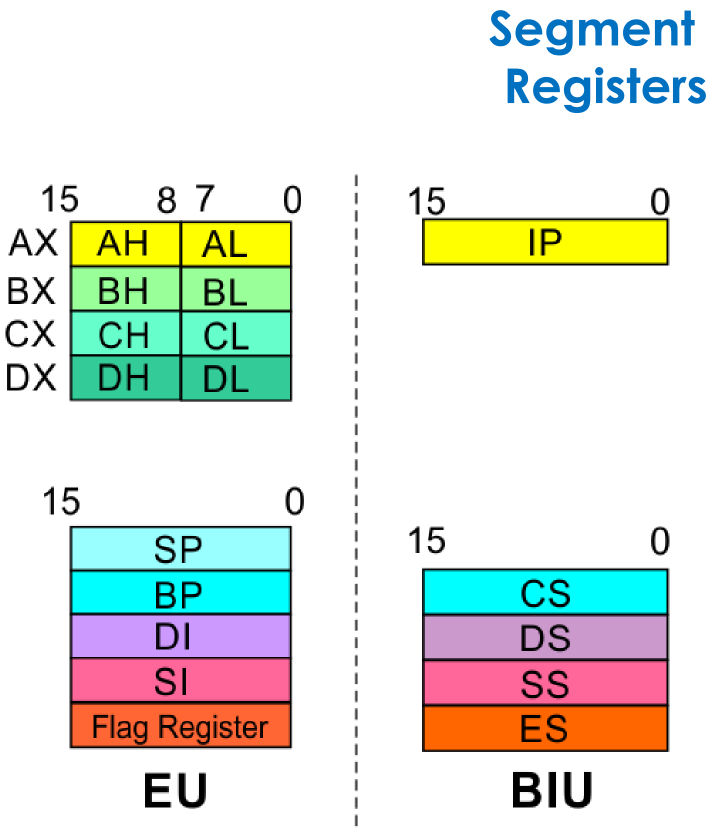
8086 Architecture



Code Segment (CS) Register

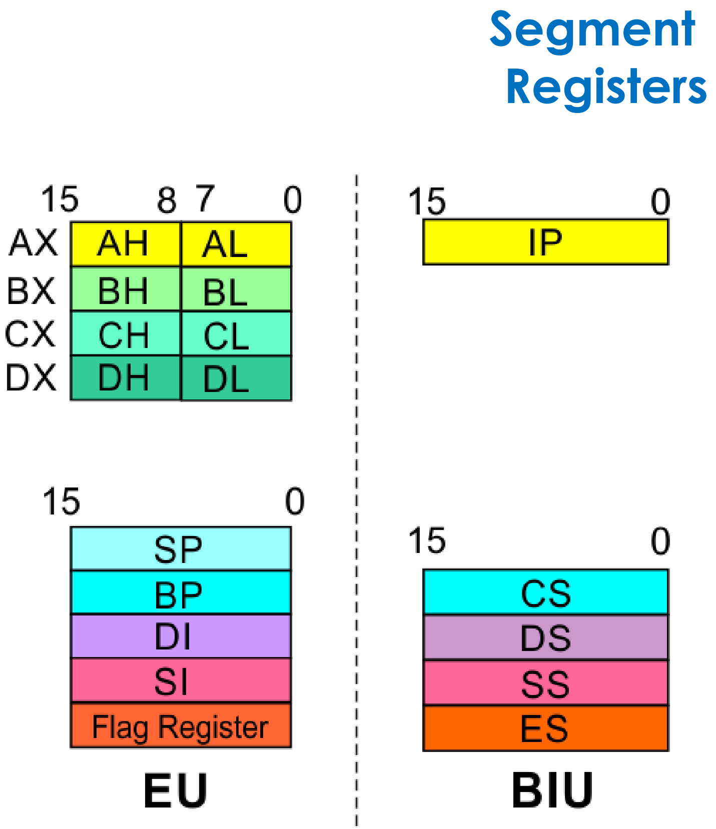
- 16-bit
- **CS** contains the base or start of the current code segment
- **Instruction Pointer** (**IP**) contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.
- That is, all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the IP.

8086 Architecture



Data Segment (DS) Register

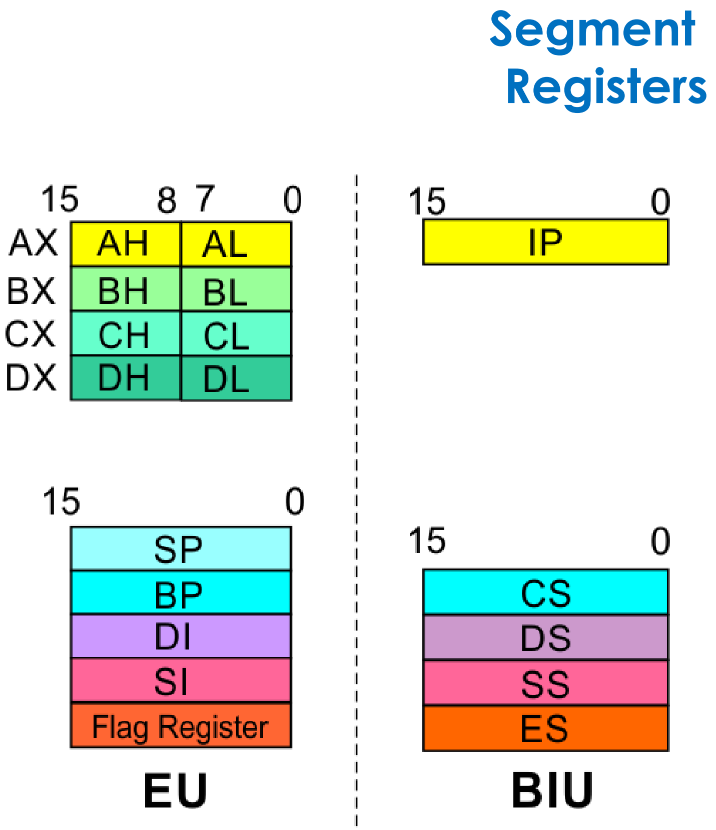
- 16-bit
- Points to the current data segment; operands for most instructions are fetched from this segment.
- The 16-bit contents of the **Source Index** (**SI**) or **Destination Index** (**DI**) or a 16-bit displacement are used as offset for computing the 20-bit physical address**.**



Stack Segment (SS) Register

- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the **Stack Segment** (**SS**) and the **Stack Pointer** (**SP**) for stack instructions such as **PUSH** and **POP**.
- In based addressing mode, the 20-bit physical stack address is calculated from the **Stack segment** (**SS**) and the **Base Pointer** (**BP**).

8086 Architecture

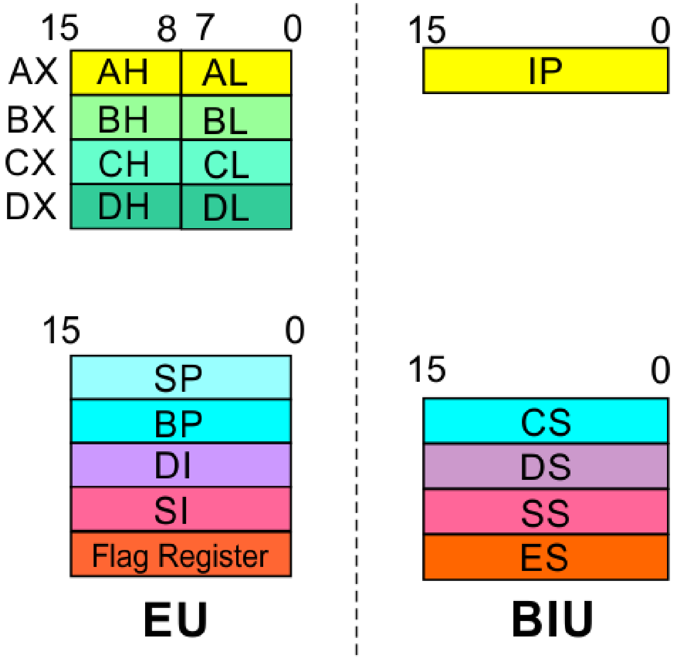


Extra Segment (ES) Register

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

8086 Architecture

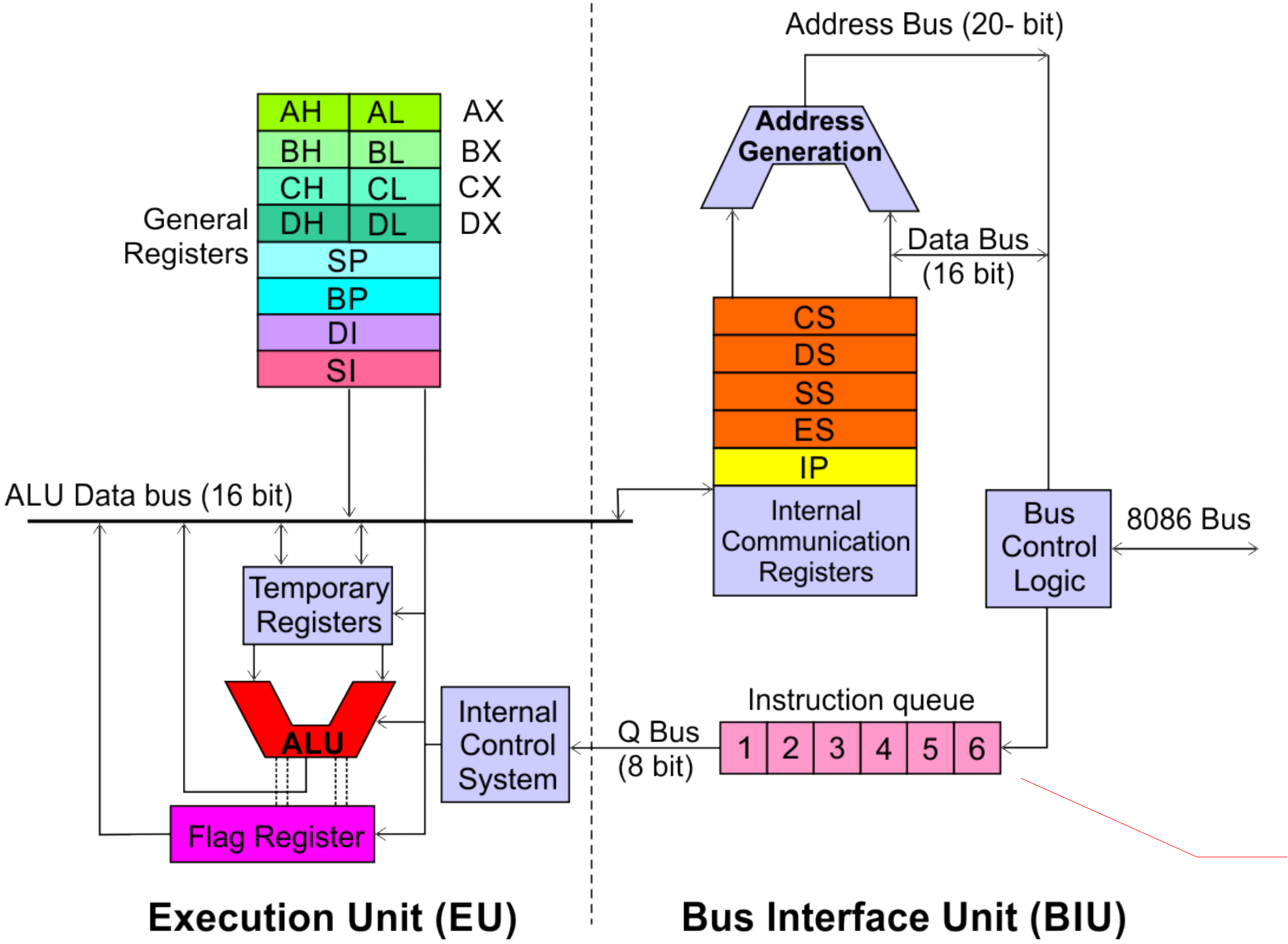
Segment Registers



Instruction Pointer

- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.

Bus Interface Unit (BIU)



- A group of **First-In-First-Out** (**FIFO**) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.
- This is done in order to speed up the execution by overlapping instruction fetch with execution.
- This mechanism is known as **pipelining**.

Instruction queue

Data Registers

Execution Unit (EU)

EU decodes and executes instructions

A decoder in the EU control system translates instructions.

Four **General purpose registers**(AX, BX, CX, DX)

Pointer registers (Stack Pointer, Base Pointer)

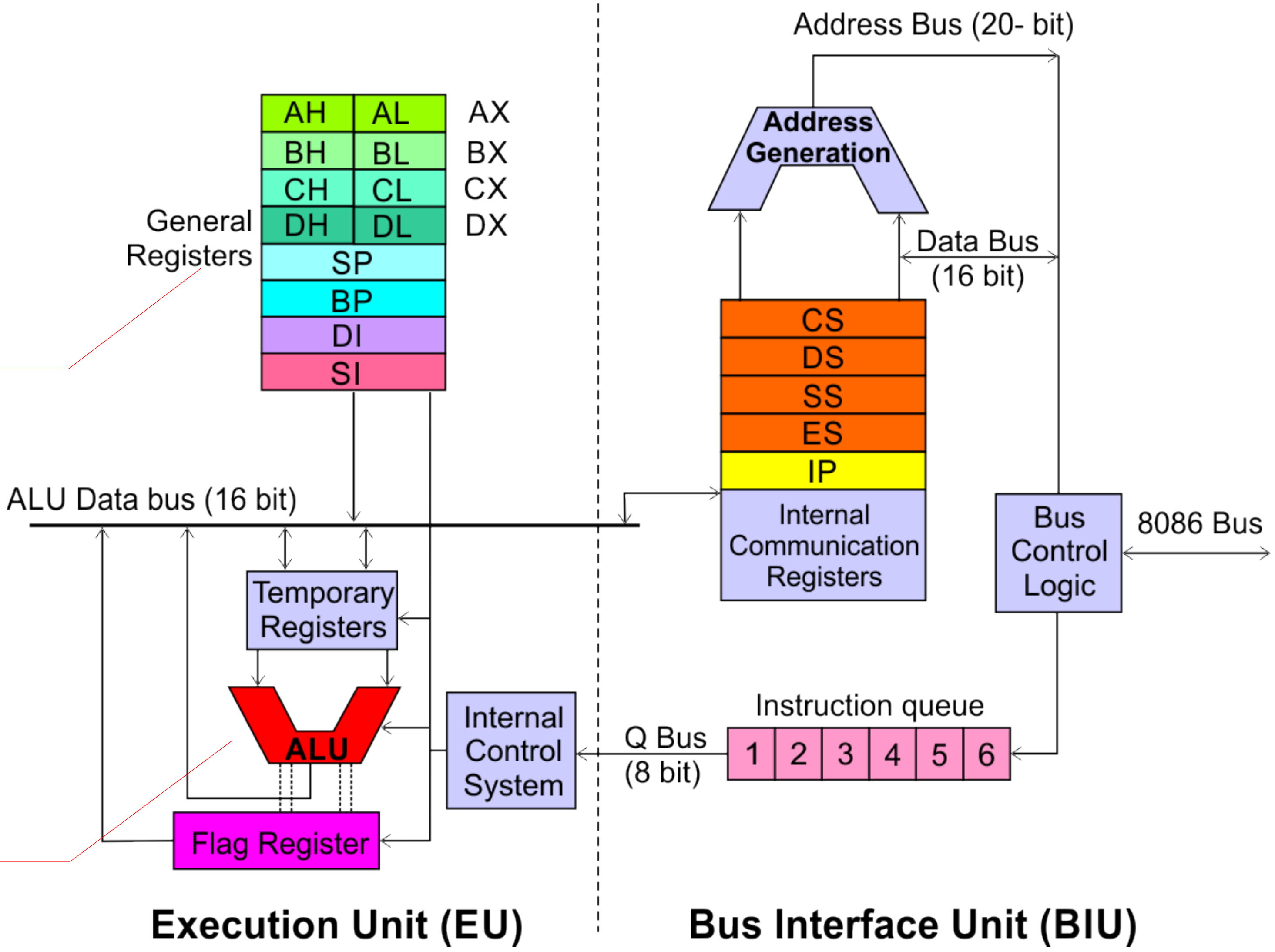
Index registers (Source Index, Destination Index)

All are registers are 16-bits wide

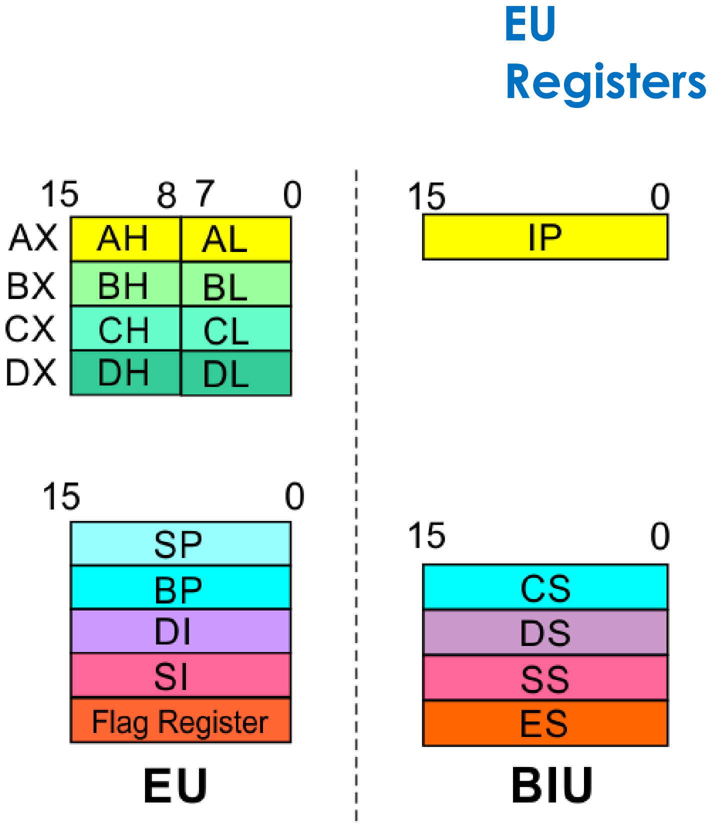
Some of the 16 bit registers can be used as two 8 bit registers as :

AX can be used as **AH** and **AL**
BX can be used as **BH** and **BL**
CX can be used as **CH** and **CL**
DX can be used as **DH** and **DL**

16-bit ALU for performing arithmetic and logic operation



8086 Architecture

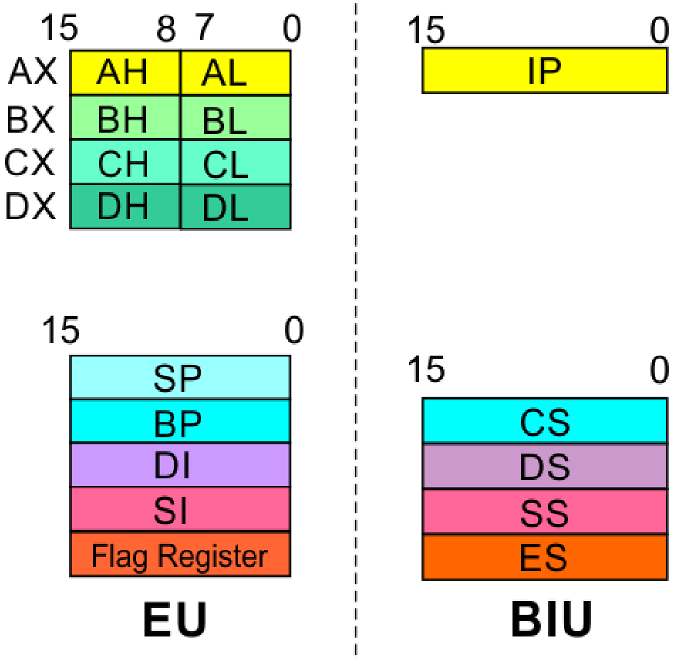


Accumulator Register (AX)

- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.
- The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to or from an I/O port.
- Multiplication and Division instructions also use the AX or AL.

8086 Architecture

EU Registers

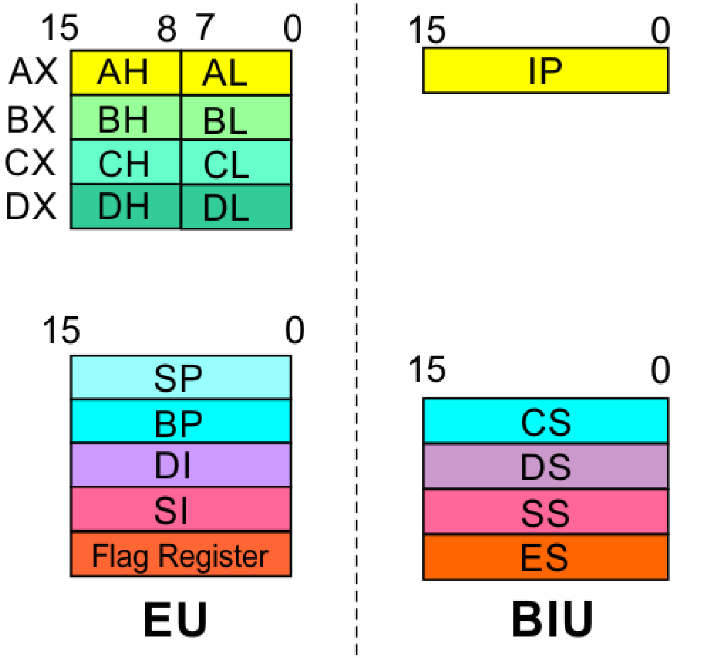


Base Register (BX)

- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- All memory references utilizing this register content for addressing use DS as the default segment register.

8086 Architecture

EU Registers



Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.

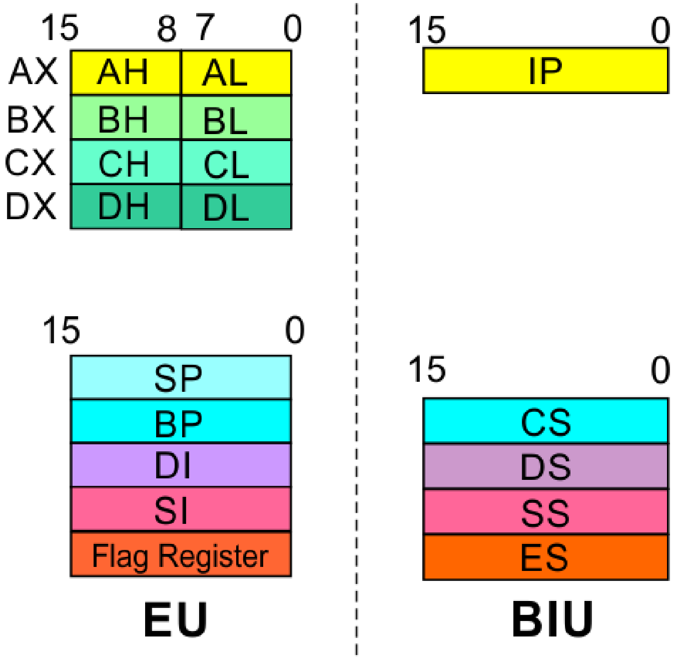
Example:

The instruction **LOOP START** automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label START.

8086 Architecture

EU Registers

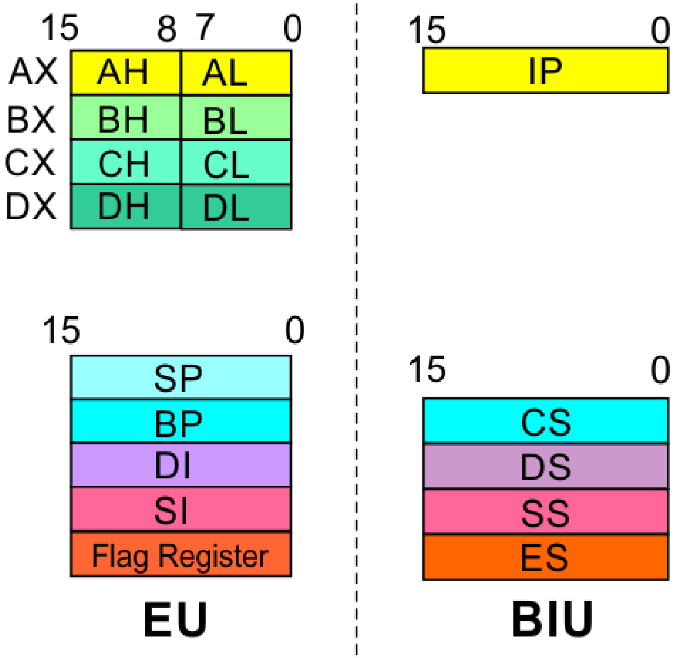


Data Register (DX)

- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.
- Used to hold the high 16-bit result (data) in 16 X 16 multiplication or the high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit reminder after division.

8086 Architecture

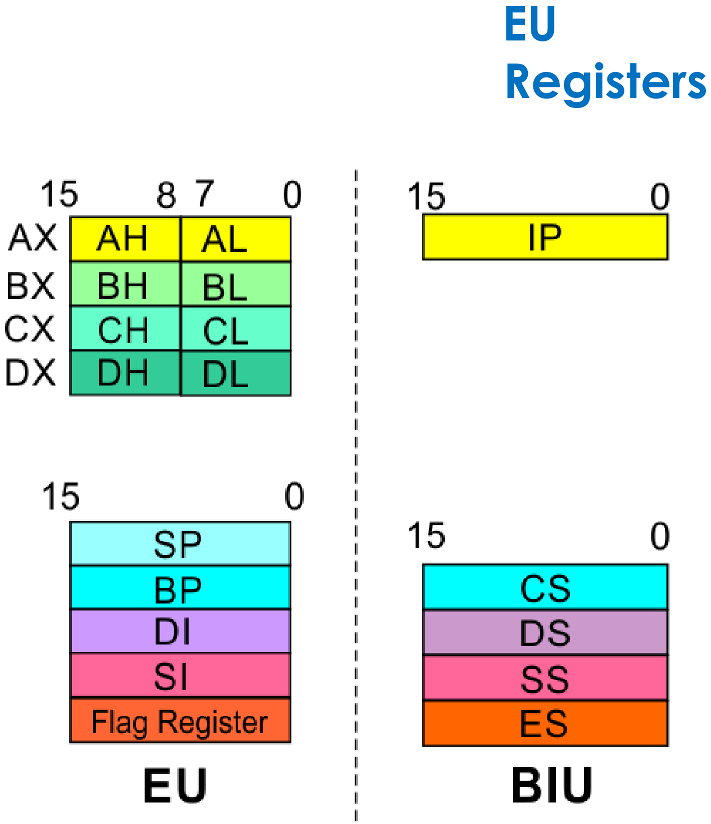
EU Registers



Stack Pointer (SP) and Base Pointer (BP)

- SP and BP are used to access data in the stack segment.
- SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.
- SP contents are automatically updated (incremented/ decremented) due to execution of a POP or PUSH instruction.
- BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.

8086 Architecture



Source Index (SI) and Destination Index (DI)

- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.

Flag Registers

Flag Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

OVERFLOW Flag (OF) – status flag
OF is set if an arithmetic operation results in an overflow.

DIRECTION Flag (DF) – control flag
DF is used for string manipulation instructions.

INTERRUPT Flag (IF) – control flag
IF is set will cause 8086 to recognize external mask interrupts. Clearing IF disables these interrupts.

TRAP Flag (TF) – control flag
TF is set if processor enters the single step execution mode by generating internal interrupts after the execution of each instruction.

SIGN Flag (SF) – status flag
SF is set when result of any computation is negative.

ZERO Flag (ZF) – status flag
ZF is set if the result of the computation or comparison is zero.

AUXILIARY Flag (AF) – status flag
AF is set if there is a carry from the LSB.

PARITY Flag (PF) – status flag
PF is set if the lower byte of the result contains even number of 1's, for odd number of 1's PF is set to zero.

CARRY Flag (CF)
CF is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Basic Assembly Instructions

MOV instruction

8-bit moves

SYNTAX: **MOV** *destination*, *source*

- Destination and source are all 8 bit reg/mem/values

Example:

MOV CL, 55h

MOV DL, CL

MOV BH, CL

16-bit moves

SYNTAX: **MOV** *destination*, *source*

Destination and source are all 16 bit reg/mem/values

Example:

MOV CX, 55FCh

MOV DX, CX

MOV BX, DI

MOV instruction

- Data can be moved among all registers
- Data can not be moved directly into segment registers (CS, DS, ES, SS)
- Data must be loaded into a non-segment register before it is moved to segment register

Example:

```
MOV AX, 3456h
MOV DS, AX
```

- Moving a value too large into a register causes error

Example:

```
MOV BL, 345h       ; illegal
MOV AX, 987654h ; illegal
```

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

Example:

```
MOV BX, 5          ; BX = 0005, where BH = 00h & BL = 05h
```

ADD instruction

SYNTAX: ADD destination, source

- ADD instruction tells CPU to add the source and destination operands and put the results in the destination
- DESTINATION = DESTINATION + SOURCE

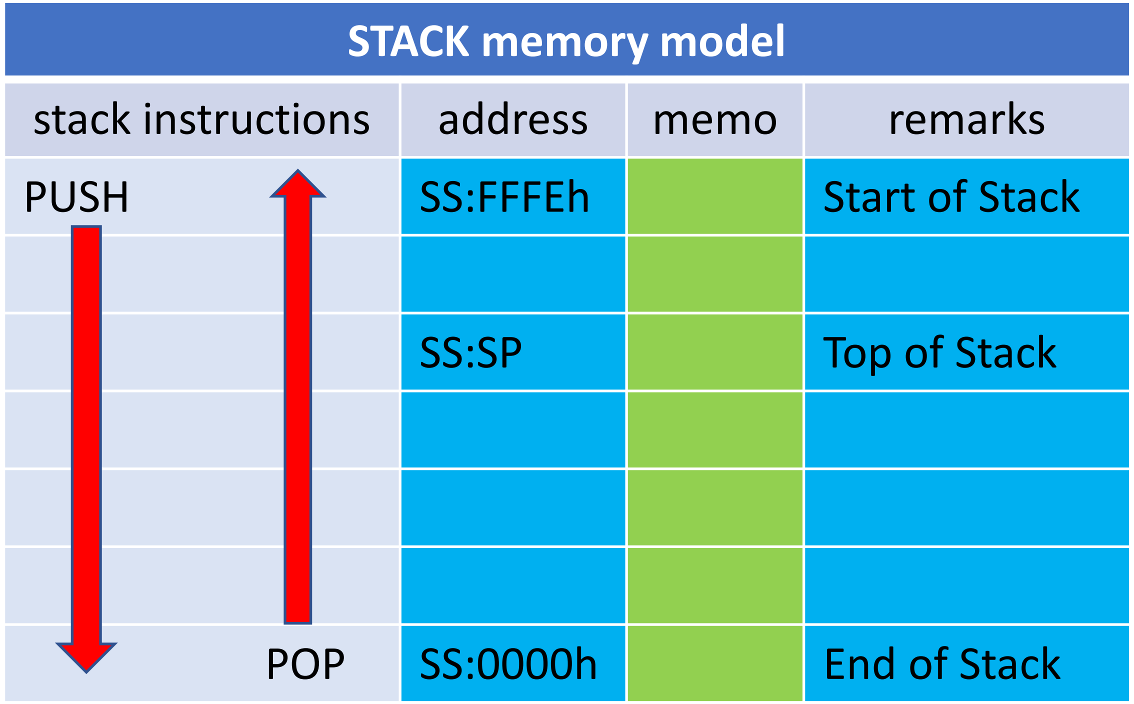
Example:

```
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, 34h is an
                    immediate operand
```

PUSH and POP instructions

PUSH

- Used for temporary storage of information such as data or addresses
- Employs **Last-In-First-Out** (**LIFO**) data movement
- When a CALL is executed, the processor automatically pushes the value of the CS and IP into the STACK
- Other registers can be pushed to the STACK
- Found in the TPA free area and grows downward in memory (inverted stack)
- Accessed using the SS and SP registers
- Values are placed into the stack using the **PUSH** instruction
- The **POP** instruction is used for removing items from the STACK



STACK memory model

stack instructions	address	memo	remarks
PUSH	SS:FFFEh		Start of Stack
	SS:SP		Top of Stack
POP	SS:0000h		End of Stack

PUSH and POP instruction

SYNTAX: **PUSH** reg/mem/val

- Pushes a value stored in reg/mem into the STACK
- Decrements SP by 2

Example:

PUSH AX ; assume the ff:

; AX = 1234h

; SS = 0105h

; SP = 0008h

SYNTAX: **POP** reg/mem/val

- Pops a value from the STACK and stores it to reg/mem i
- Increments SP by 2

BEFORE PUSH OPERATION

address	memo	remarks
SS:FFFEh		Start of Stack
0105:0008h		Top of Stack
0105:0007h		
0105:0006h		
0105:0005h		
SS:0000h		End of Stack

AFTER PUSH OPERATION

address	memo	remarks
SS:FFFEh		Start of Stack
0105:0008h		
0105:0007h	12h	AH
0105:0006h	34h	AL /Top of Stack
0105:0005h		
SS:0000h		End of Stack