Lecture 03: 80X86 Microprocessor

Key Content

- Internal organization of 8086
  - Registers, pipelining
- Chip interface of 8086
- Memory management in 8086
- Addressing mode in 8086

Key Content

- Internal organization of 8086
  - Registers, pipelining
- Chip interface of 8086
- Memory management in 8086
- Addressing mode in 8086

The 80x86 IBM PC and Compatible Computers

Chapter 1

80X86 Microprocessor

Chapter 9.1

8088 Microprocessor

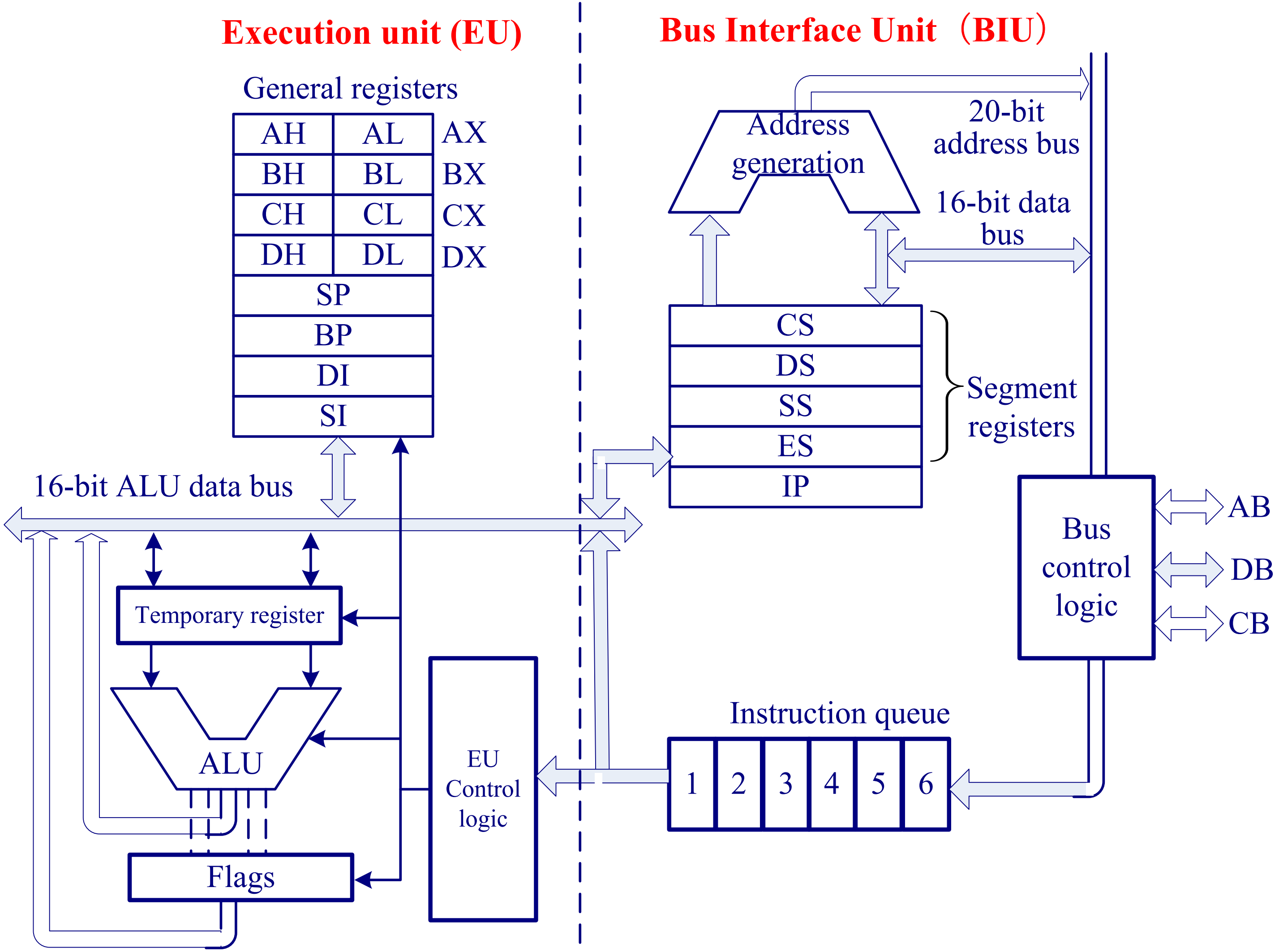
Evolution of 80X86 Family

- 8086, born in 1978
  - First **16**-bit microprocessor
  - **20**-bit address data bus, i.e. 2^{20} = 1MB memroy
  - First pipelined microprocessor
- 8088
  - Data bus: 16-bit internal, 8-bit external
  - Fit in the 8-bit world, e.g., motherboard, peripherals
  - Adopted in the IBM PC + MS-DOS open system
- 80286, 80386, 80486
  - Real/protected modes
  - Virtual memory

Internal Structure of 8086

- Two sections
  - ***Bus interface unit*** (BIU): accesses memory and peripherals
  - ***Execution unit*** (EU): executes instructions previously fetched
  - Work simultaneously

Internal Structure of 8086



Execution unit (EU)

General registers

AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX
SP		
BP		
DI		
SI		

16-bit ALU data bus

Temporary register

ALU

Flags

EU Control logic

Bus Interface Unit （BIU）

Address generation

20-bit address bus

16-bit data bus

CS

DS

SS

ES

IP

Segment registers

Bus control logic

AB

DB

CB

Instruction queue

1 2 3 4 5 6

Bus Interface Unit

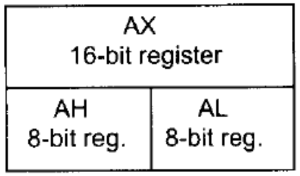
- Take in charge of data transfer between CPU and memory and I/O devices as well
  - Instruction fetch, instruction queuing, operand fetch and storage, address relocation and Bus control
- Consists of :
  - four 16-bit segment registers: CS, DS, ES, SS
  - One 16-bit instruction pointer: IP
  - One 20-bit address adder: e.g., CS left-shifted by 4 bits + IP (CS*16+IP)
  - A 6-byte instruction queue
- While the EU is executing an instruction, the BIU will fetch the next one or several instructions from the memory and put in the queue

Execution Unit

- Take in charge of instruction execution
- Consists of:
  - Four 16-bit general registers: Accumulator (AX), Base (BX), Count (CX) and Data (DX)
  - Two 16-bit pointer registers: Stack Pointer (SP), Base Pointer (BP)
  - Two 16-bit index registers: Source Index (SI) and Destination Index (DI)
  - One 16-bit flag register: 9 of the 16 bits are used
  - ALU

Registers

- On-chip storage: super fast & expensive
- Store information temporarily



8-bit register:									D7	D6	D5	D4	D3	D2	D1	D0
16-bit register:	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

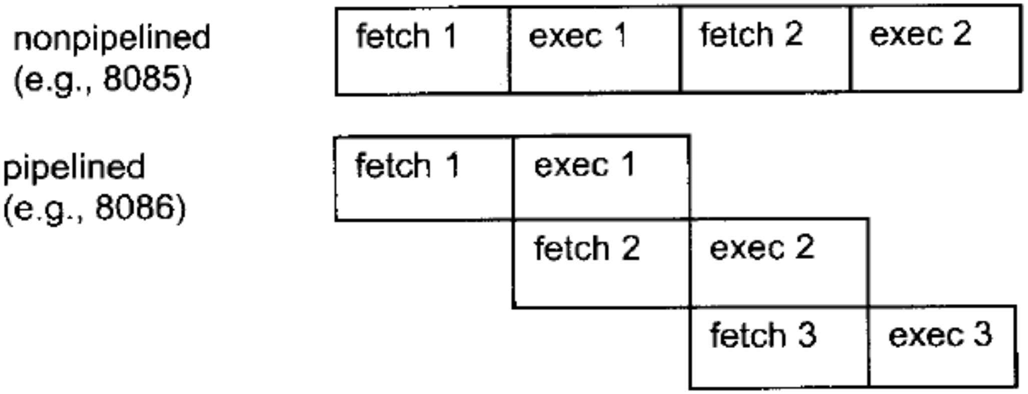
- Six groups

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).

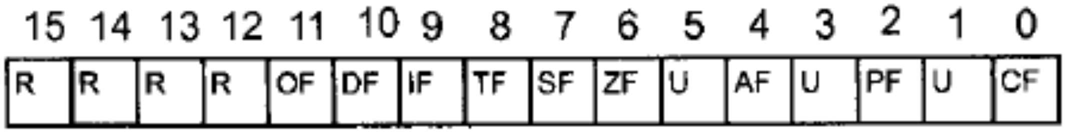
Pipelining in 8086

- BIU fetches and stores instructions once the queue has more than 2 empty bytes
- EU consumes instructions pre-fetched and stored in the queue at the same time
- Increases the efficiency of CPU
- *When it works?*
  - Sequential instruction execution
  - *Branch penalty*: when jump instruction executed, all pre-fetched instructions are discarded



Flag Register

- 16-bit, *status register*, processor status word (PSW)
- 6 conditional flags
  - CF, PF, AF, ZF, SF, and OF
- 3 control flags
  - DF, IF, TF



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

R =	reserved	SF =	sign flag
U =	undefined	ZF =	zero flag
OF =	overflow flag	AF =	auxiliary carry flag
DF =	direction flag	PF =	parity flag
IF =	interrupt flag	CF =	carry flag
TF =	trap flag		

Conditional Flags

- **CF (Carry Flag):** set whenever there is a carry out, from d7 after a 8-bit op, from d15 after a 16-bit op
- **PF (Parity Flag):** the parity of the op result's low-order byte, set when the byte has an even number of 1s
- **AF (Auxiliary Carry Flag):** set if there is a carry from d3 to d4, used by BCD-related arithmetic
- **ZF (Zero Flag):** set when the result is zero
- **SF (Sign Flag):** copied from the sign bit (the most significant bit) after op
- **OF (Overflow Flag):** set when the result of a signed number operation is too large, causing the sign bit error

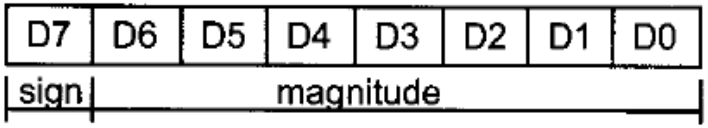
Signed Number

- Original value 原码
  - Can't be added directly
  - Two zeros (+0/-0)
- One's complement 反码
  - Can be added directly
  - Two zeros (+0/-0)
- Two's complement 补码
  - Can be added directly
  - Only one zero

Value	Original value	One's complement	Two's complement
+0	0000 0000	0000 0000	0000 0000
-0	1000 0000	1111 1111	0000 0000
+1	0000 0001	0000 0001	0000 0001
-1	1000 0001	1111 1110	1111 1111

More about Signed Number, CF&OF

- The most significant bit (MSB) as sign bit, the rest of bits as magnitude

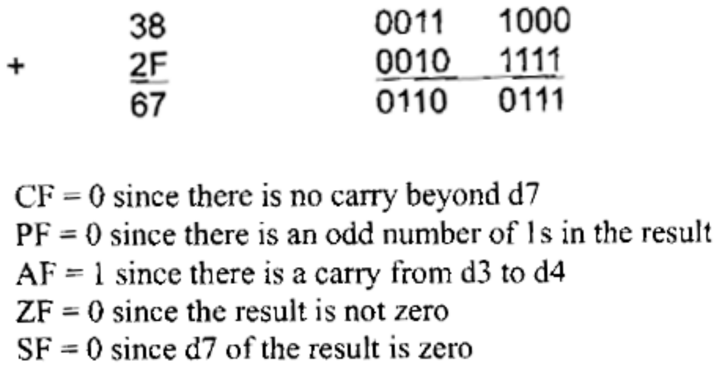


D7 D6 D5 D4 D3 D2 D1 D0

sign magnitude

  - For negative numbers, D7 is 1, but the magnitude is represented in 2's complement
- CF is used to detect errors in unsigned arithmetic operations
- OF is used to detect errors in signed arithmetic operations
  - Two ways to understand the OF
  - 1. OF = 1, when two values are positive, but the result is negative; when two values are negative, but the result is positive
  - 2. E.g., for 8-bit ops, OF is set to 1 when there is a carry from d6 to d7 and no carry from d7;or when there is a carry from d7 and no carry from d6 to d7

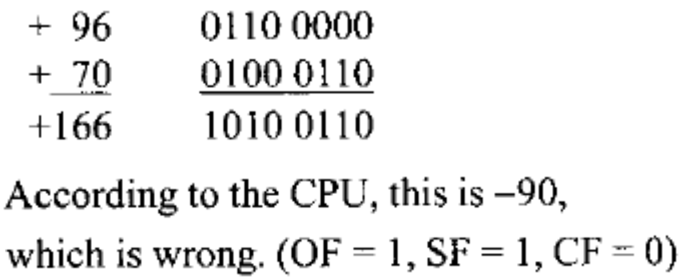
Examples of Conditional Flags



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

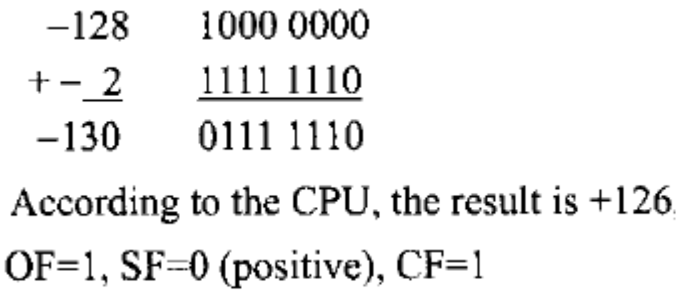
CF = 0 since there is no carry beyond d7
PF = 0 since there is an odd number of 1s in the result
AF = 1 since there is a carry from d3 to d4
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero
OF = 0 since there is no carry from d6 to d7 and no carry beyond d7

How can CPU know whether an operation is unsigned or signed?



```
+  96      0110 0000
+  70      0100 0110
+166       1010 0110
```

According to the CPU, this is –90,
which is wrong. (OF = 1, SF = 1, CF = 0)



```
 –128      1000 0000
+ – 2      1111 1110
 –130      0111 1110
```

According to the CPU, the result is +126.
OF=1, SF=0 (positive), CF=1

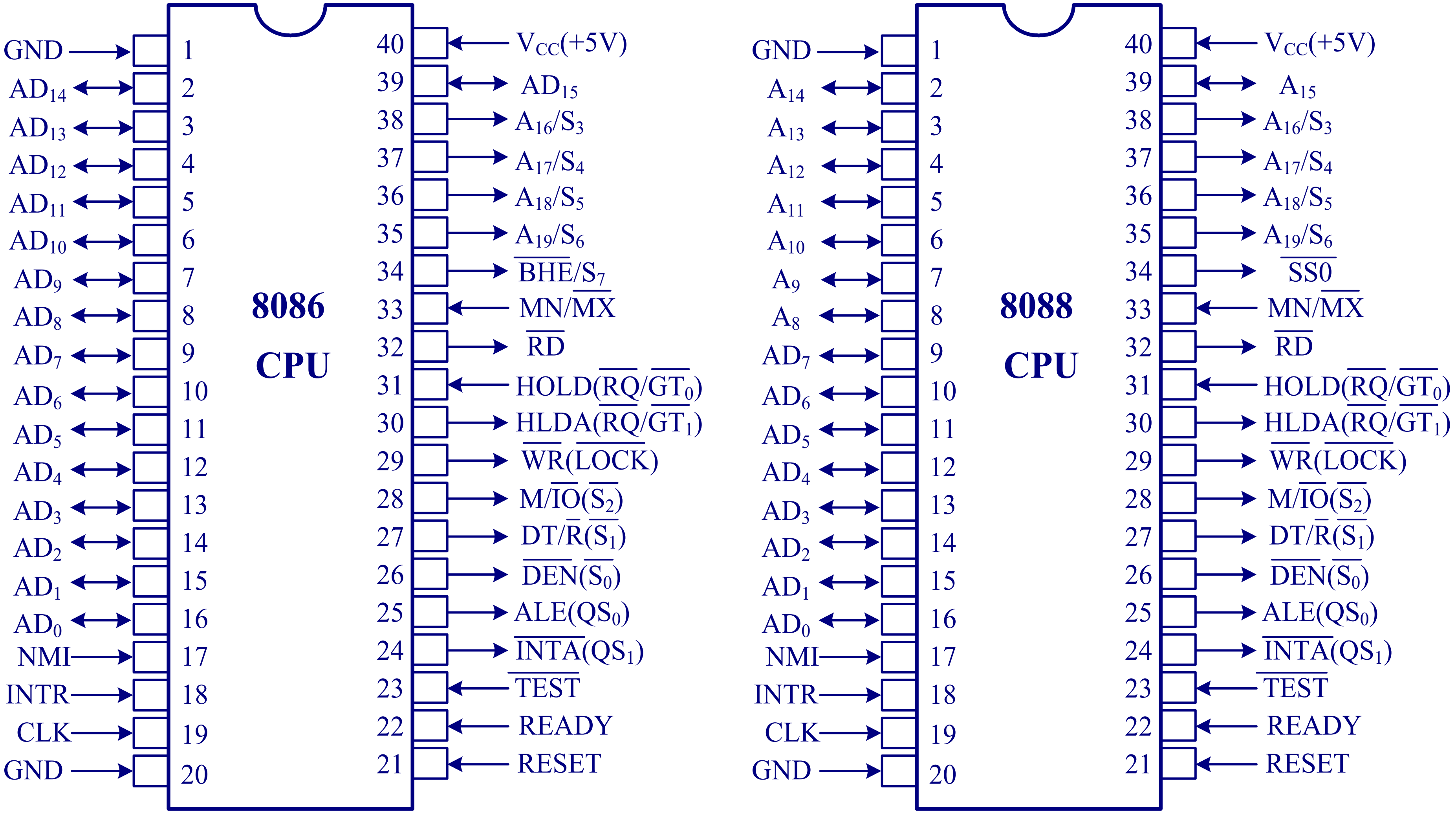
Control Flags

- **IF (Interrupt Flag):** set or cleared to enable or disable only the external maskable interrupt requests
  - After reset, all flags are cleared which means you (as a programmer) have to set IF in your program if allow INTR.
- **DF (Direction Flag):** indicates the direction of string operations
- **TF (Trap Flag):** when set it allows the program to single-step, meaning to execute one instruction at a time for debugging purposes

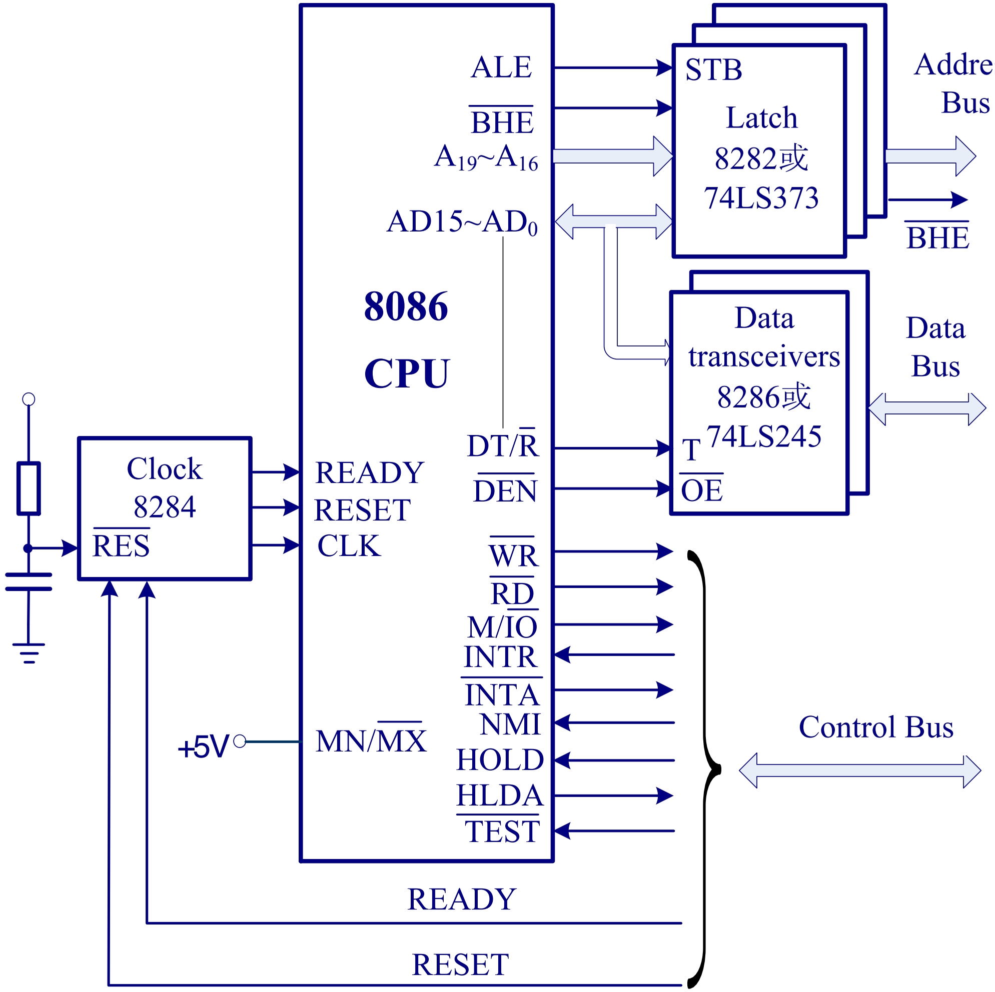
Key Content

- Internal organization of 8086
  - Registers, pipelining
- Chip interface of 8086
- Memory management in 8086
- Addressing mode in 8086

8086/8088 Pins (Compare them and tell the difference)



Minimum Mode Configuration



8086/88's two work modes:

- Minimum mode :$MN/\overline{MX}$=1
  - Single CPU;
  - Control signals from the CPU
- Maximum mode :$MN/\overline{MX}$=0
  - Multiple CPUs(8086+8087)
  - 8288 control chip supports

Control Signals

- **MN/~MX:** Minimum mode (high level), Maximum mode (low level)

- **~RD:** output, CPU is reading from memory or IO
- **~WR:** output, CPU is writing to memory or IO
- **M/~IO:** output, CPU is accessing memory (high level) or IO (low level)
- **READY:** input, memory/IO is ready for data transfer

- **~DEN:** output, used to enable the data transceivers
- **DT/~R:** output, used to inform the data transceivers the direction of data transfer, i.e., sending data (high level) or receiving data (low level)

- **~BHE:** output, ~BHE=0, AD8-AD15 are used, ~BHE=1, AD8-AD15 are not in use
- **ALE:** output, used as the latch enable signal of the address latch

Control Signals

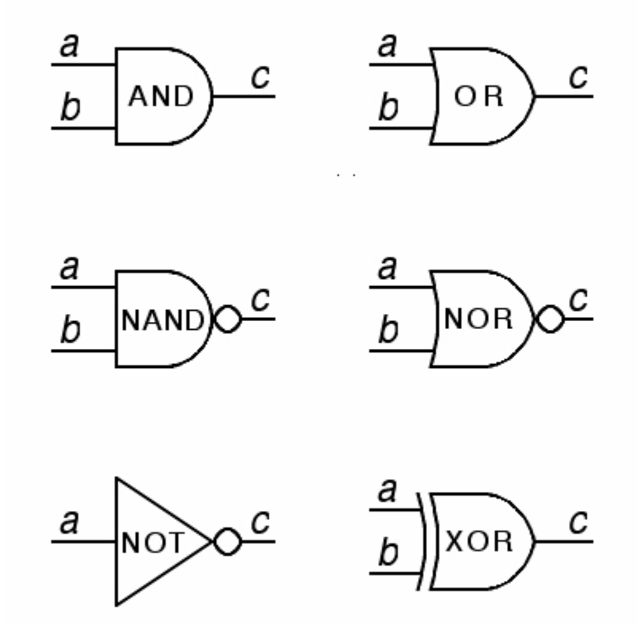
- **HOLD:** input signal, hold the bus request
- **HLDA:** output signal, hold request ack

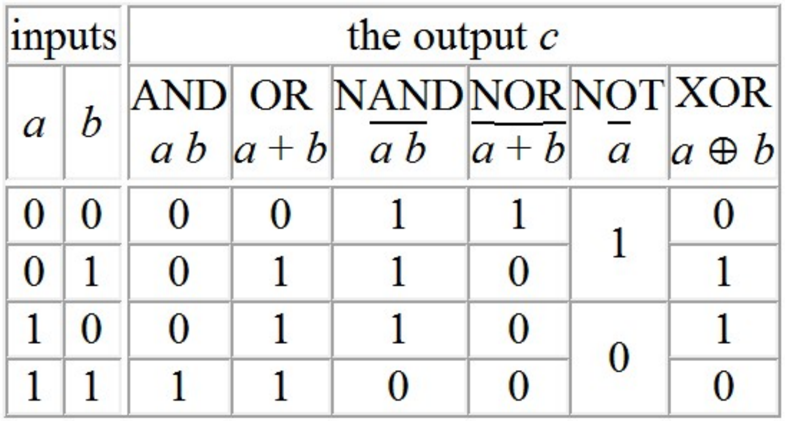
- **INTR:** input, interrupt request from 8259 interrupt controller, maskable by clearing the IF in the flag register
- **INTA:** output, interrupt ack
- **NMI:** input, non-maskable interrupt, CPU is interrupted after finishing the current instruction; cannot be masked by software

- **RESET:** input signal, reset the CPU
  - IP, DS, SS, ES and the instruction queue are cleared
  - CS = FFFFH
  - *What is the address of the first instruction that the CPU will execute after reset?*

Remember CMOS Gates?

- These following gates are called combinational logic (组合逻辑电路)
  - The output is only determined by the input (输出仅仅由输入决定)



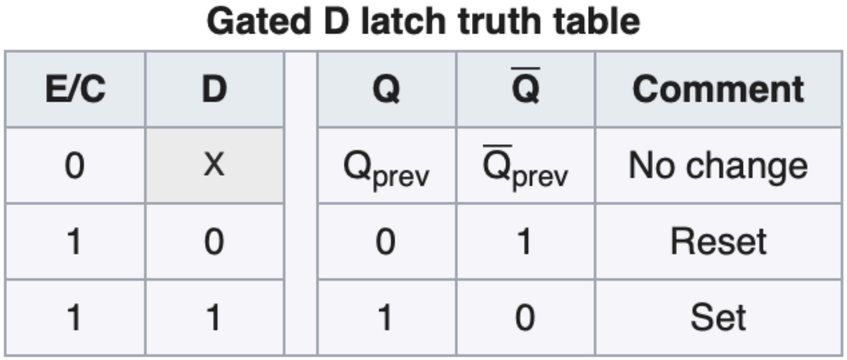


inputs		the output c					
a	b	AND $a\,b$	OR $a+b$	NAND $\overline{a\,b}$	NOR $\overline{a+b}$	NOT $\overline{a}$	XOR $a \oplus b$
0	0	0	0	1	1	1	0
0	1	0	1	1	0		1
1	0	0	1	1	0	0	1
1	1	1	1	0	0		0

Boolean logic operations

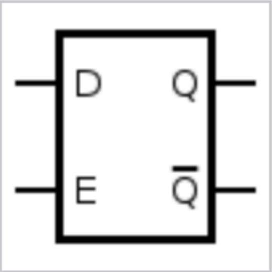
Sequential Logic 时序逻辑电路

- D Latch: D锁存器
  - Logically, D-latch is the same as a SRAM cell

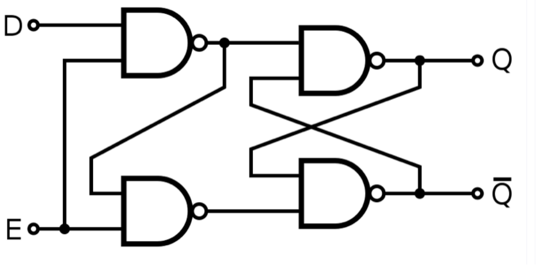


Gated D latch truth table

E/C	D	Q	$\overline{Q}$	Comment
0	X	Q_{prev}	$\overline{Q}_{prev}$	No change
1	0	0	1	Reset
1	1	1	0	Set

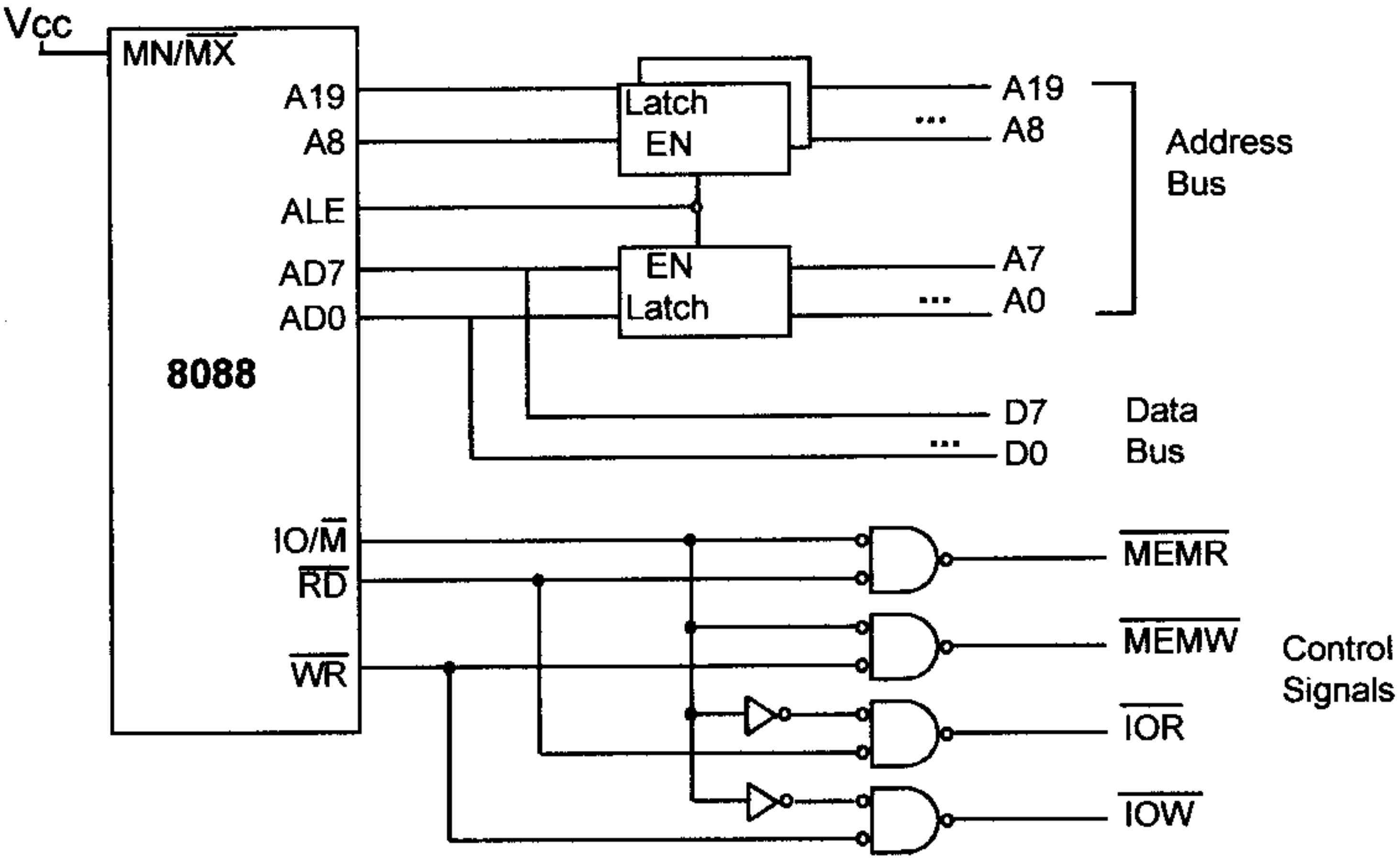


Symbol for a gated D latch

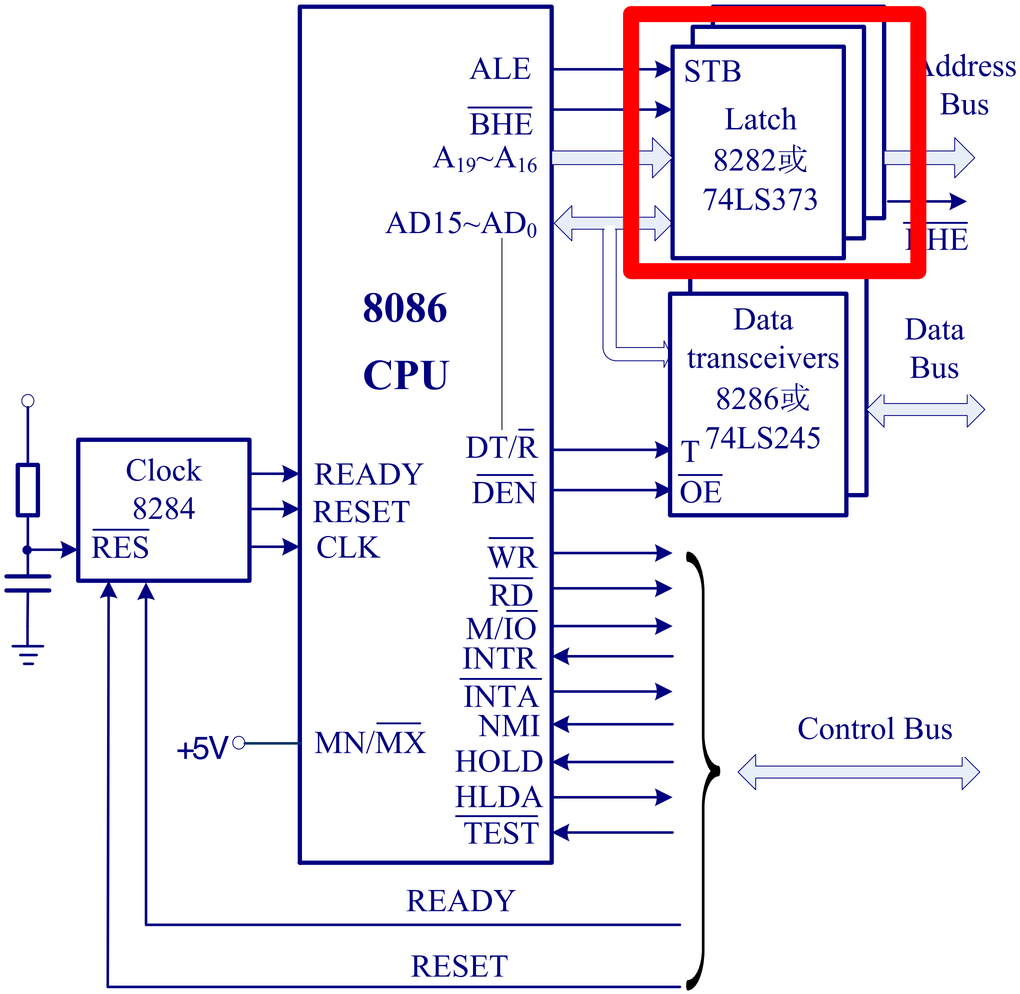


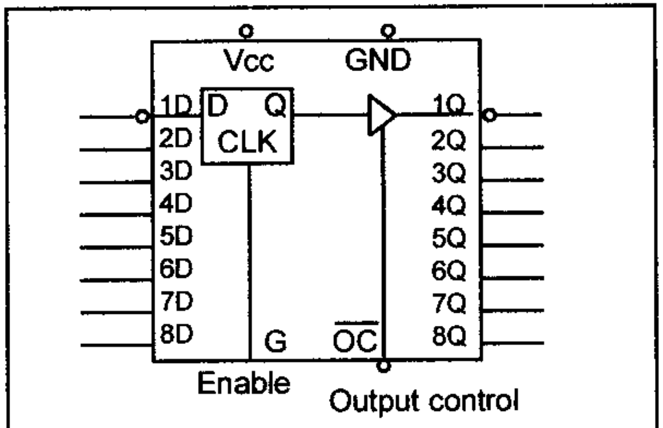
A gated D latch based on an $\overline{SR}$ NAND latch

Memory/IO Control Signals



Address/Data Demultiplexing & Address latching



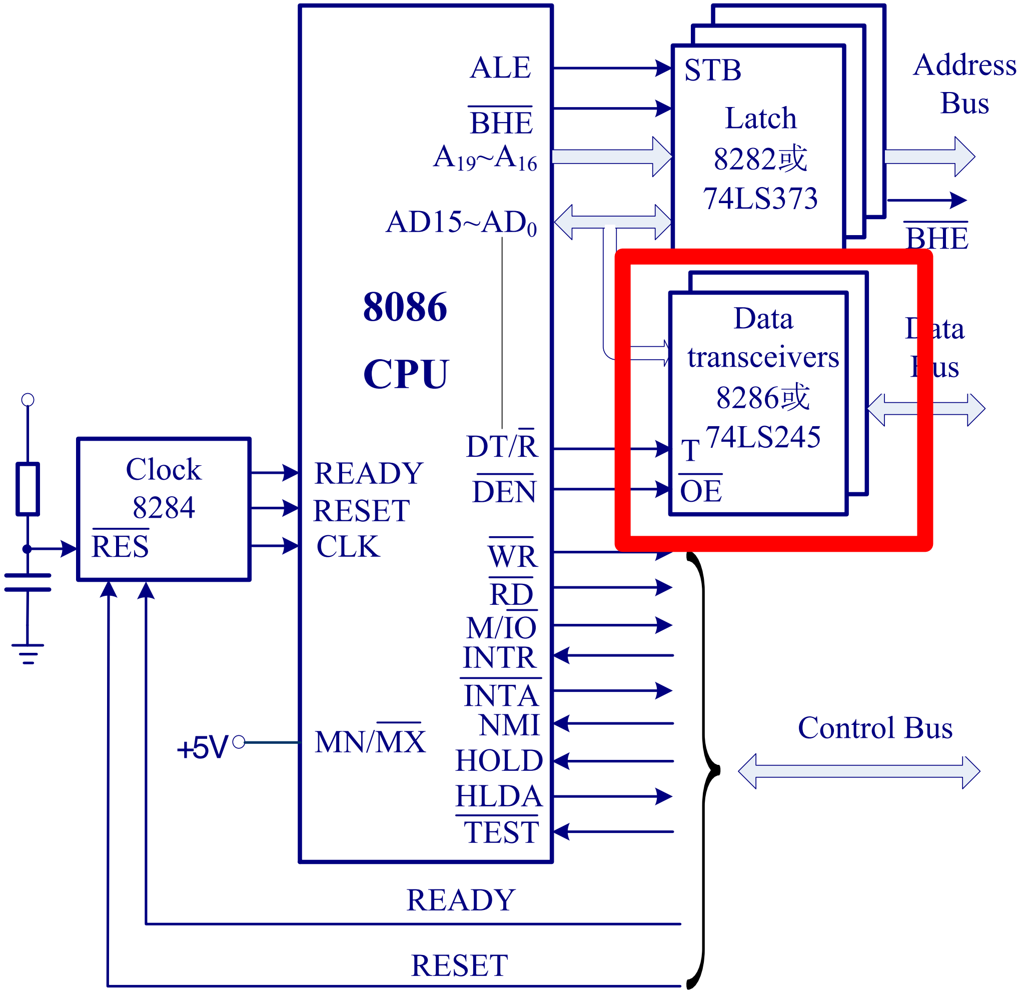


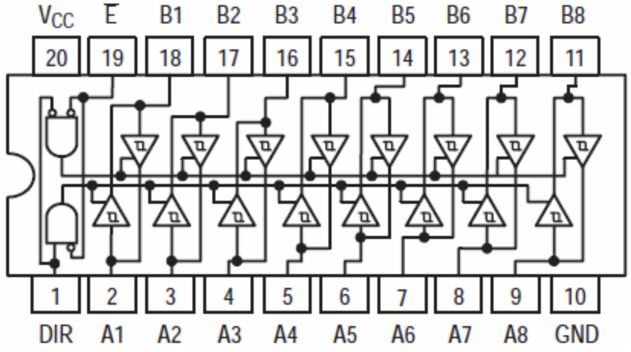
Function Table

Output Control	Enable		Output
	G	D	
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

74LS373 D Latch

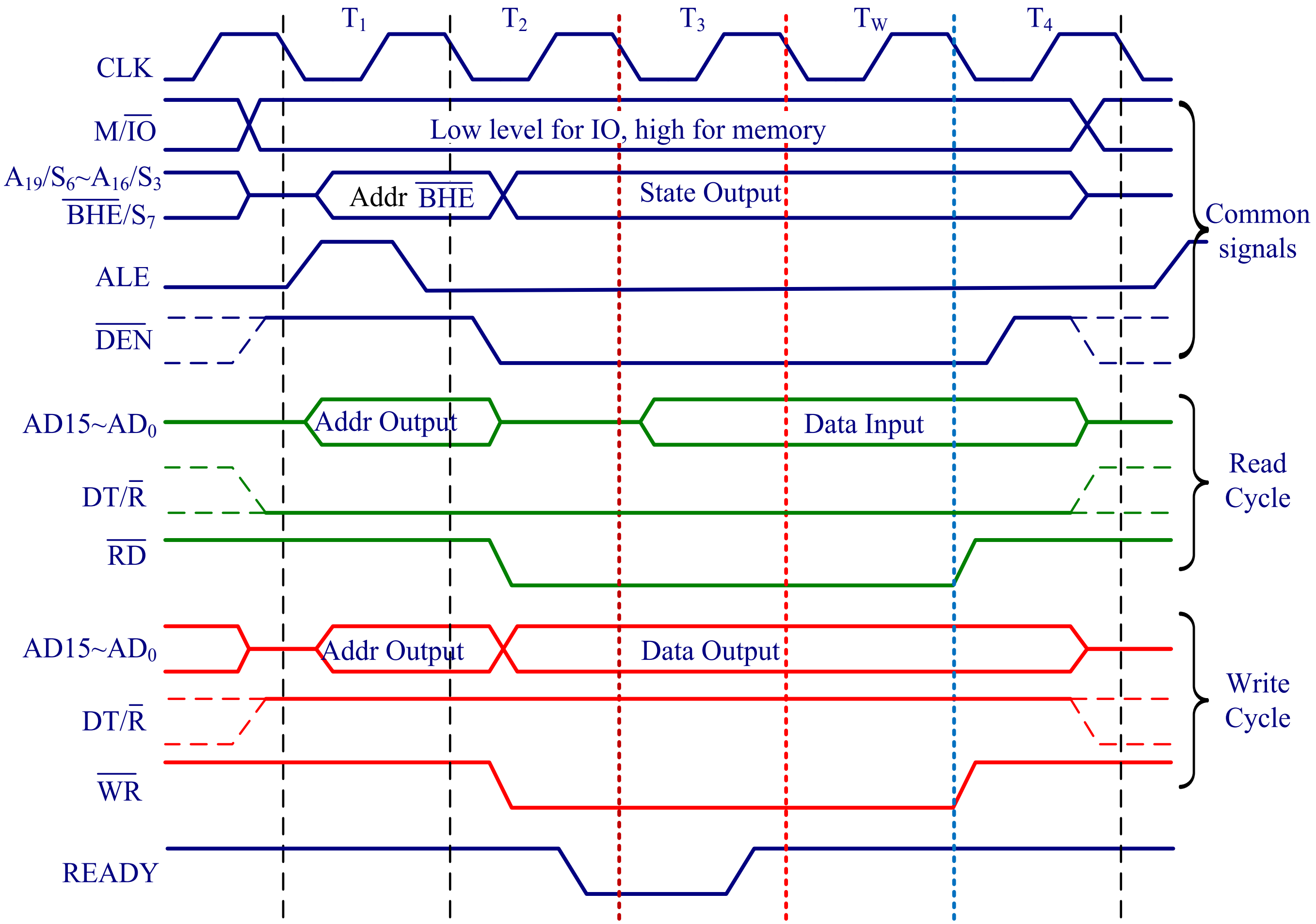
Data Bus Transceiver



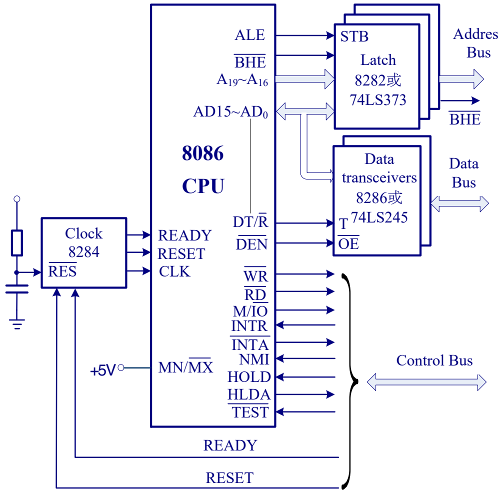


INPUTS		OUTPUT
E	DIR	
L	L	Bus B Data to Bus A
L	H	Bus A Data to Bus B
H	X	Isolation

8086/88 Bus Cycle (for data transfers)



At least 4 clock cycles



Key Content

- Internal organization of 8086
  - Registers, pipelining
- Chip interface of 8086
- Memory management in 8086
- Addressing mode in 8086

8086 Programming

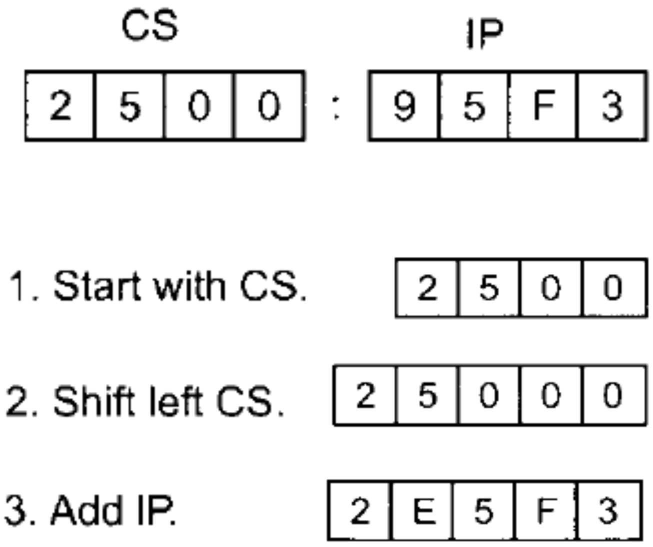
- A typical program on 8086 consists of at least three *segments*
  - code segment: contains instructions that accomplish certain tasks
  - data segment: stores information to be processed
  - stack segment: store information temporarily
- What is a segment?
  - A memory block includes up to 64KB. Why?
  - Begins on an address evenly divisible by 16, i.e., an address looks like in XXXX0H. Why?

Logical & Physical Address

- Physical address
  - 20-bit address that is actually put on the address bus
  - A range of 1MB from 00000H to FFFFFH
  - Actual physical location in memory
- Logical address
  - Consists of a *segment value* (determines the beginning of a segment) and an *offset address* (a relative location within a 64KB segment)
  - E.g., an instruction in the code segment has a logical address in the form of CS (code segment register):IP (instruction pointer)

Logical & Physical Address

- logical address -> physical address
  - Shift the segment value left one hex digit (or 4 bits)
  - Then adding the above value to the offset address
  - One logical -> only one physical
- Segment range representation
  - Maximum 64KB
  - logical *2500:0000 – 2500:FFFF*
  - Physical *25000H – 34FFFH* (*25000 + FFFF*)



CS IP

2 5 0 0 : 9 5 F 3

1. Start with CS. 2 5 0 0

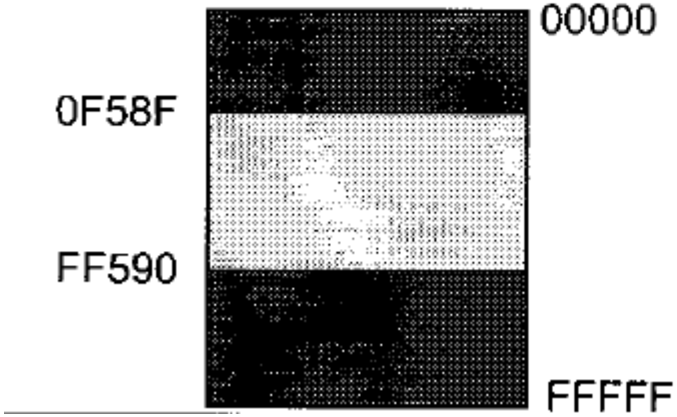
2. Shift left CS. 2 5 0 0 0

3. Add IP. 2 E 5 F 3

Physical Address Wrap-around

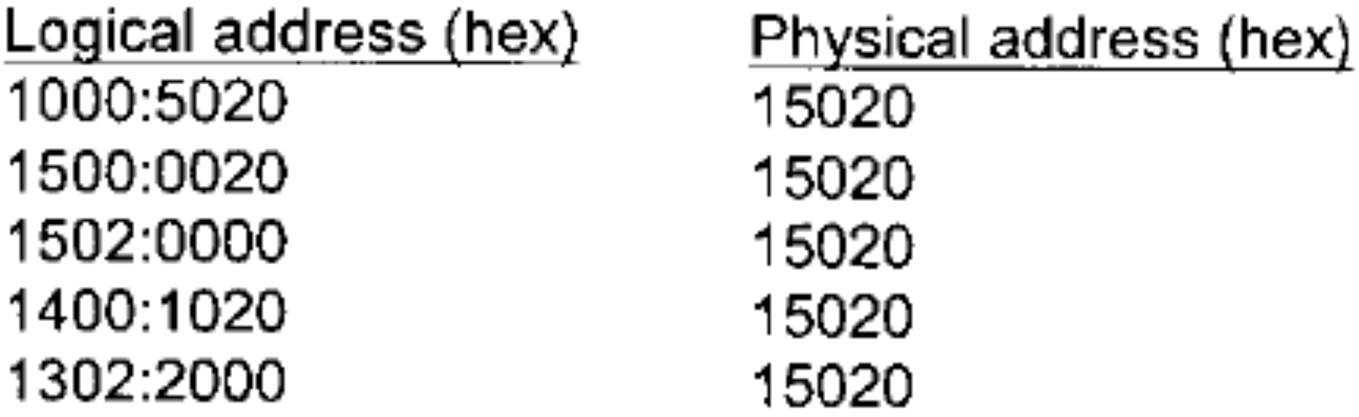
- When adding the offset to the shifted segment value results in an address beyond the maximum value *FFFFFH*
- *E.g., what is the range of physical addresses if CS=FF59H?*
  - Solution:

The low range is FF590H, and the range goes to FFFFFH and wraps around from 00000H to 0F58FH (FF590+FFFF).



Logical & Physical Address

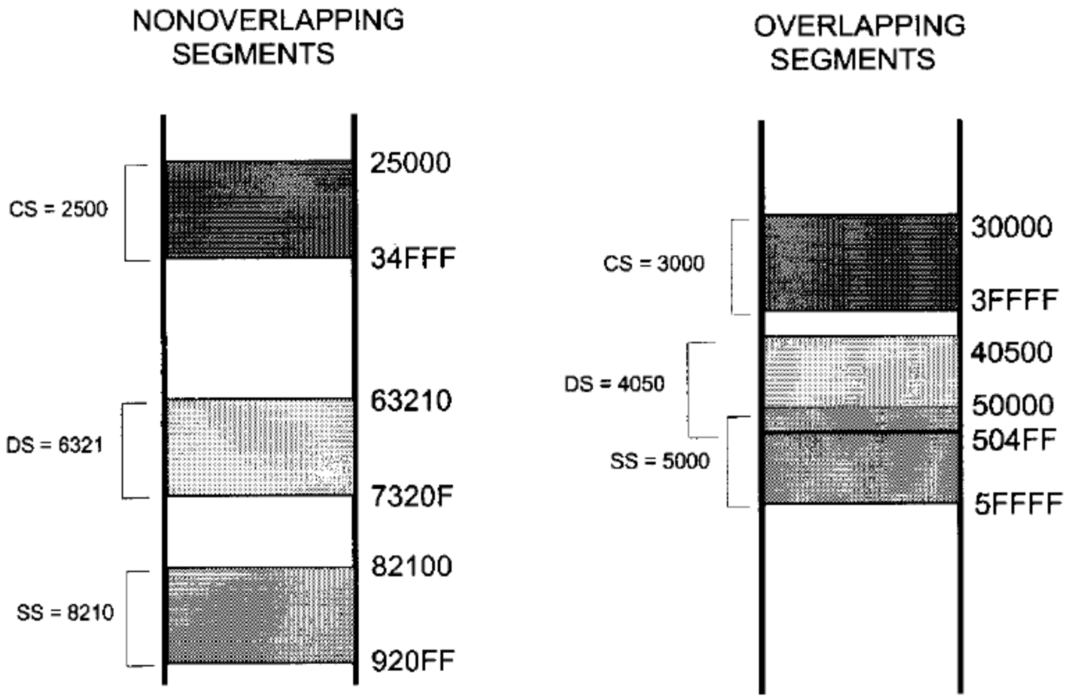
- Physical address -> logical address ?
  - One physical address can be derived from different logical addresses
  - E.g.,



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

Segment Overlapping

- Two segments can overlap
  - Dynamic behaviour of the segment and offset concept
  - May be desirable in some circumstances



Code Segment

- 8086 fetches instructions from the code segment
  - Logical address of an instruction: **CS:IP**
  - Physical address is generated to retrieve this instruction from memory
  - *What if desired instructions are physically located beyond the current code segment?*

Solution: Change the CS value so that those instructions can be located using new logical addresses

Data Segment

- Information to be processed is stored in the data segment
  - Logical address of a piece of data: **DS:offset**
    - **Offset value**: e.g., 0000H, 23FFH
    - **Offset registers** for data segment: **BX**, **SI** and **DI**
  - Physical address is generated to retrieve data (8-bit or 16-bit) from memory
  - *What if desired data are physically located beyond the current data segment?*

Solution: Change the DS value so that those data can be located using new logical addresses

Data Representation in Memory

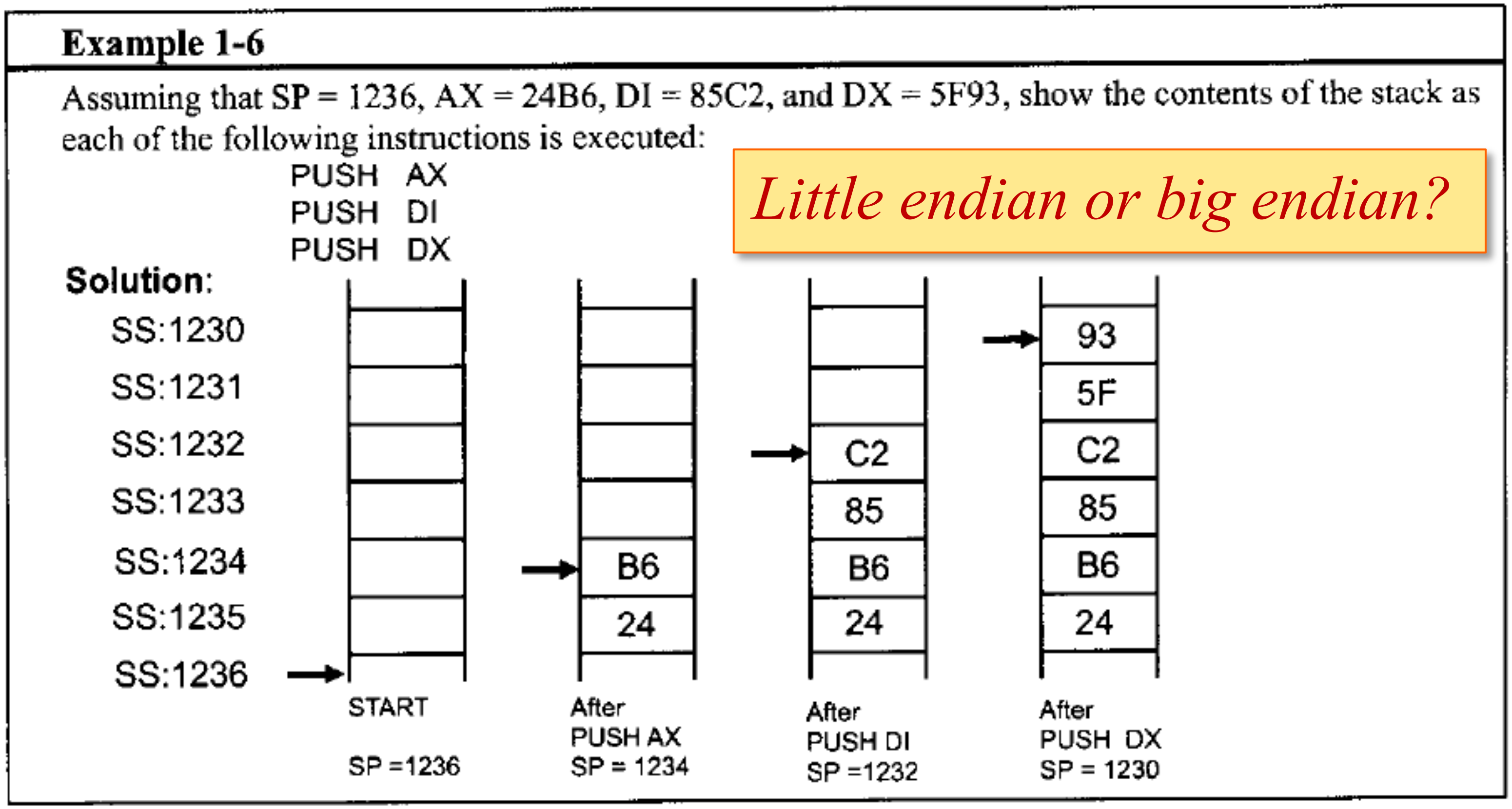
- Memory can be logically imagine as a consecutive block of bytes
- *How to store data whose size is larger than a byte?*
  - Little endian: the low byte of the data goes to the low memory location
  - Big endian: the high byte of the data goes to the low memory location
  - E.g., 2738H

Stack Segment

- A section of RAM memory used by the CPU to store information temporarily
  - Logical address of a piece of data: **SS:SP** (special applications with **BP**)
  - Most registers (except segment registers and SP) inside the CPU can be stored in the stack and brought back into the CPU from the stack using ***push*** and ***pop***, respectively
  - Grows **downward** from upper addresses to lower addresses in the memory allocated for a program
    - Why? To protect other programs from destruction
    - Note: Ensure that the code section and stack section would not write over each other

Push & Pop

- 16-bit operation



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:

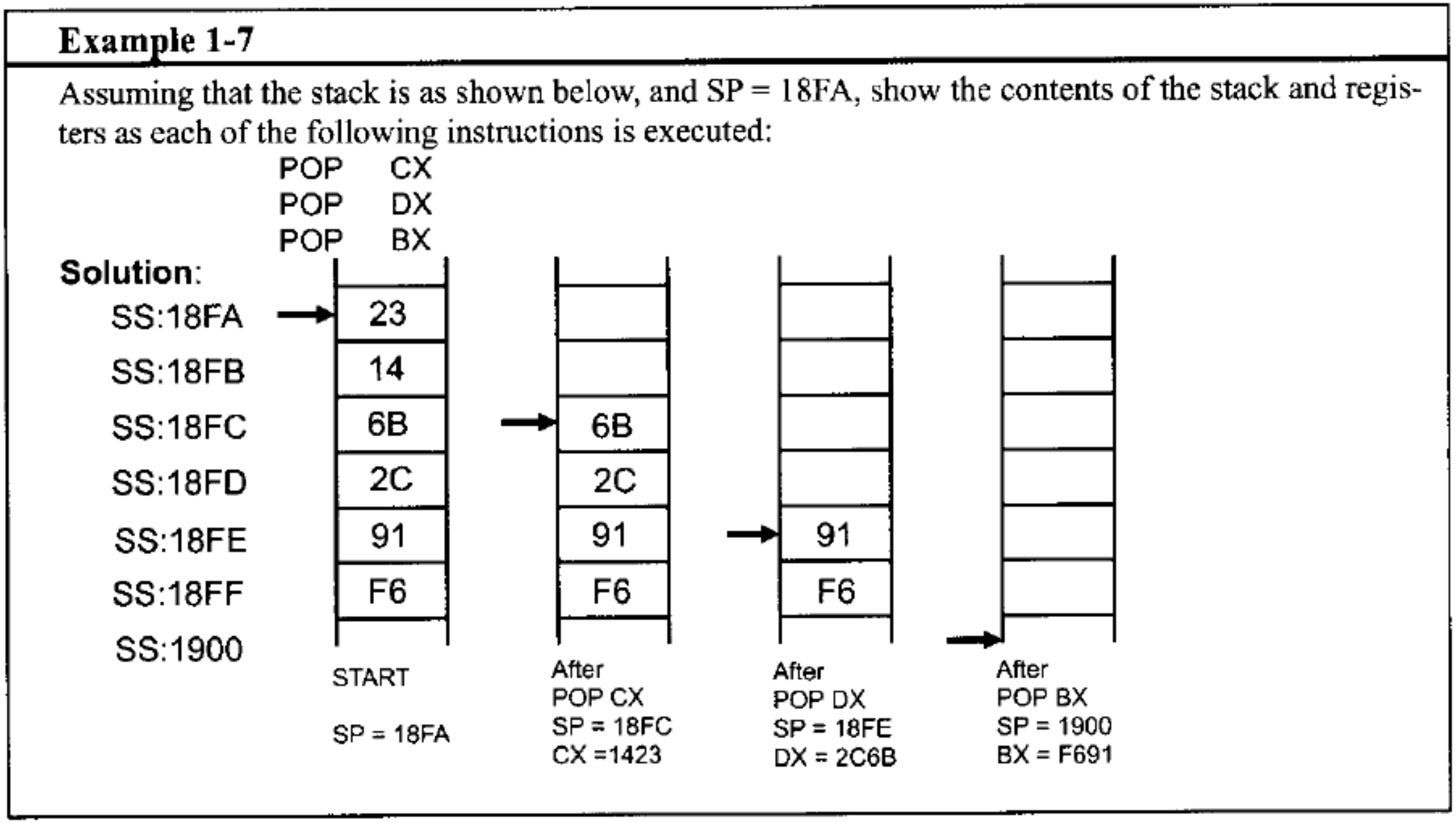
```
PUSH  AX
PUSH  DI
PUSH  DX
```

Little endian or big endian?

Solution:

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				
	START SP =1236	After PUSH AX SP = 1234	After PUSH DI SP =1232	After PUSH DX SP = 1230

Push & Pop



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				→

Extra Segment

- An extra data segment, essential for string operations
  - Logical address of a piece of data: **ES:offset**
    - **Offset value**: e.g., 0000H, 23FFH
    - **Offset registers** for data segment: **BX**, **SI** and **DI**
- **In Summary,**

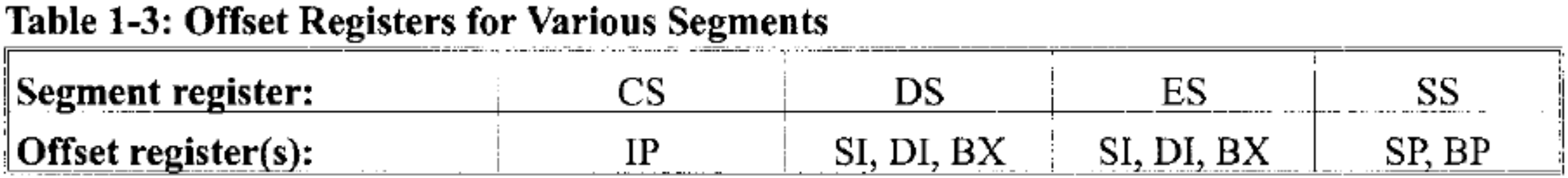
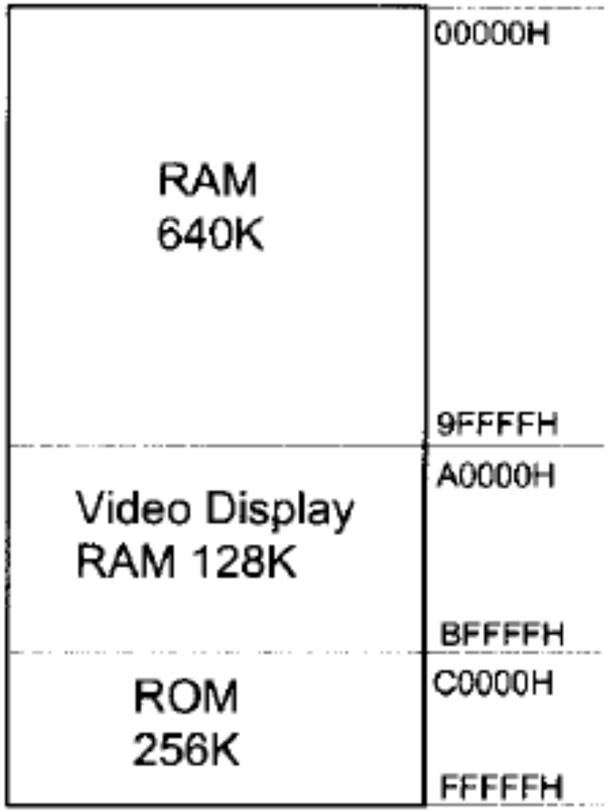


Table 1-3: Offset Registers for Various Segments

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

Memory map of the IBM PC

- 1MB logical address space
- 640K max RAM
  - In 1980s, 64kB-256KB
  - MS-DOS, application software
  - DOS does memory management; you do not set CS, DS and SS
- Video display RAM
- ROM
  - 64KB BIOS
  - Various adapter cards



BIOS Function

- Basic input-output system (BIOS)
  - Tests all devices connected to the PC when powered on and reports errors if any
  - Load DOS from disk into RAM
  - Hand over control of the PC to DOS

- Recall that after CPU being reset, what is the first instruction that CPU will execute?

Key Content

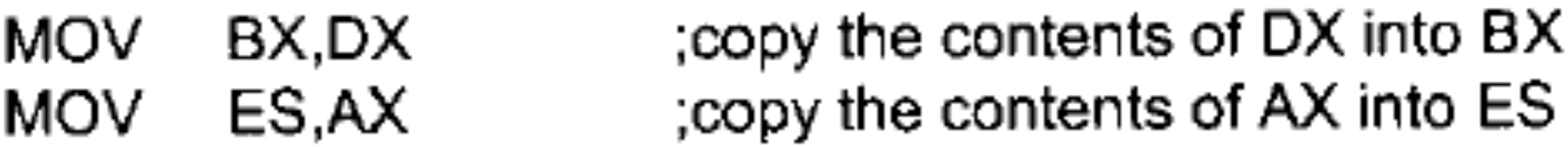
- Internal organization of 8086
  - Registers, pipelining
- Chip interface of 8086
- Memory management in 8086
- Addressing mode in 8086

80X86 Addressing Modes

- How CPU can access operands (data)
- 80X86 has seven distinct addressing modes
  - Register
  - Immediate
  - Direct
  - Register indirect
  - Based relative
  - Indexed relative
  - Based indexed relative
- Take the MOV instruction for example
  - MOV destination, source
  - Destination and source should have the same size

Register Addressing Mode

- Data are held within registers
  - No need to access memory
  - E.g.,

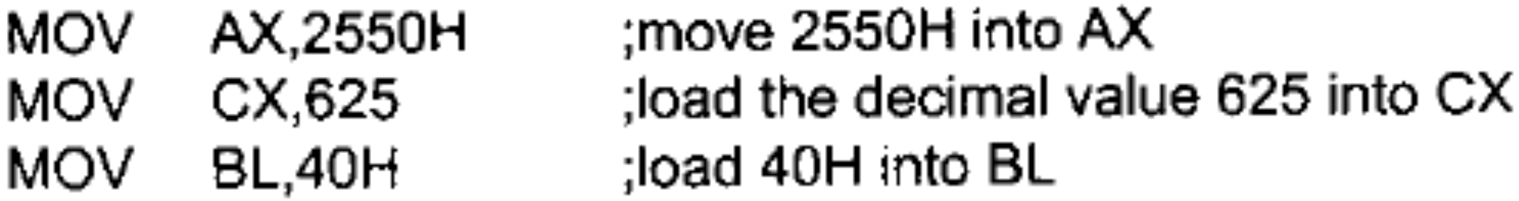


```
MOV     BX,DX            ;copy the contents of DX into BX
MOV     ES,AX            ;copy the contents of AX into ES
```

Data can be moved among ALL registers except CS (can not be set) and IP (cannot be accessed by MOV)

Immediate Addressing Mode

- The source operand is a constant
  - Embedded in instructions
  - No need to access memory
  - E.g.,



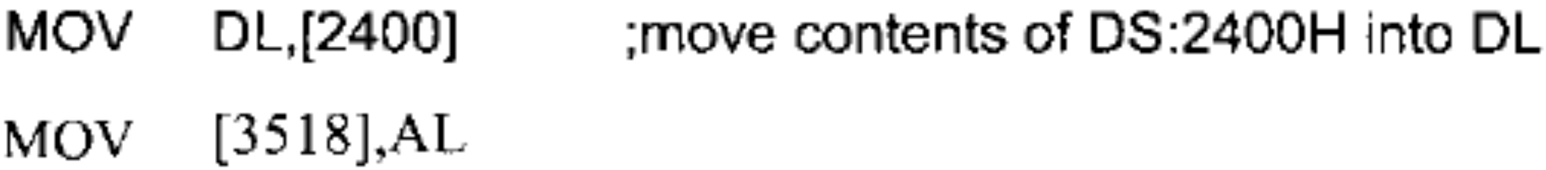
```
MOV    AX,2550H        ;move 2550H into AX
MOV    CX,625          ;load the decimal value 625 into CX
MOV    BL,40H          ;load 40H into BL
```

Immediate numbers CANNOT be moved to segment registers

How to change the value of a segment register?

Direct Addressing Mode

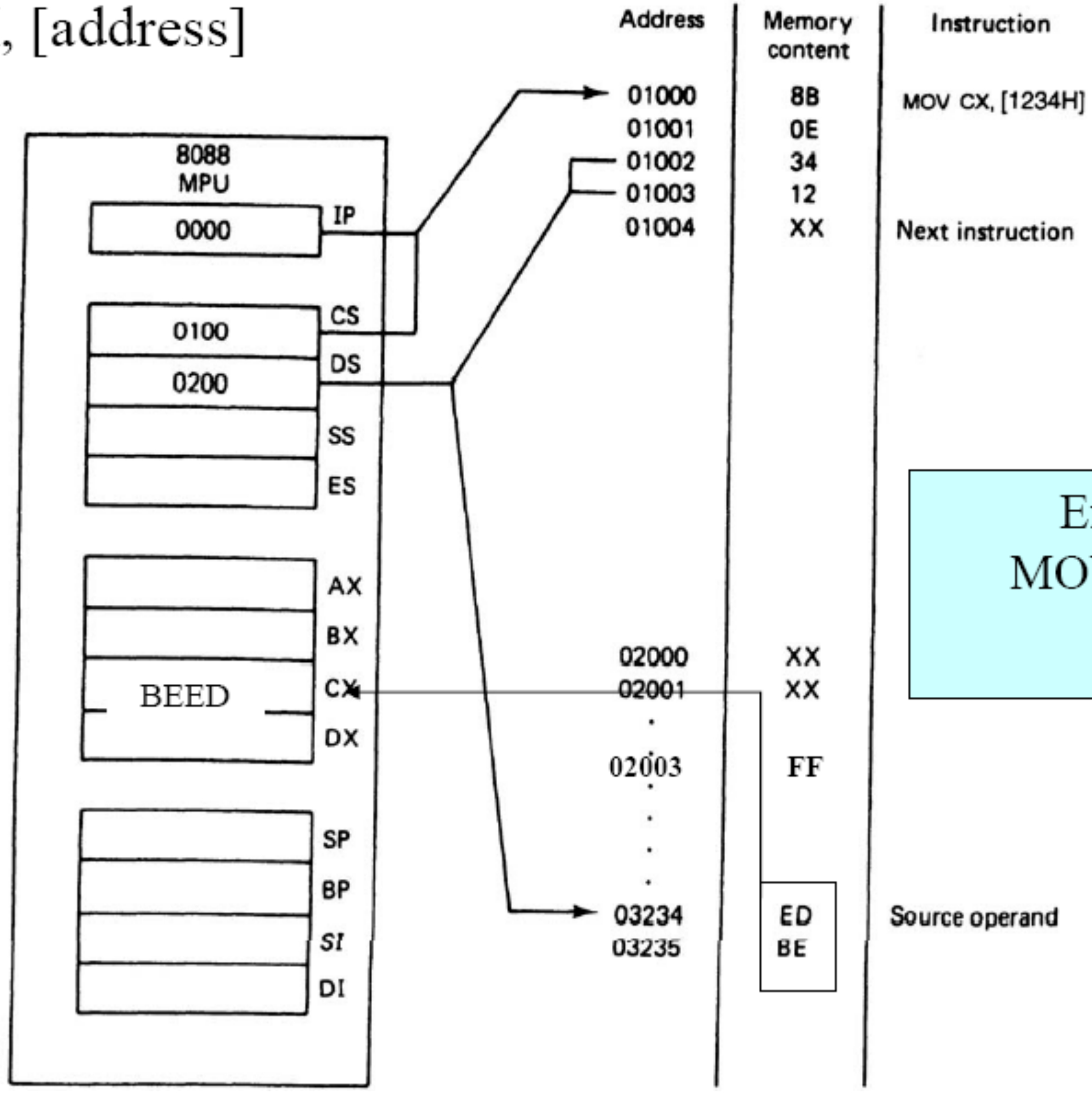
- Data is stored in memory and the address is given in instructions
  - Offset address in the data segment (**DS**) by default
  - Need to access memory to gain the data
  - E.g.,



```
MOV    DL,[2400]        ;move contents of DS:2400H into DL
MOV    [3518],AL
```

Direct Addressing Mode

MOV CX, [address]

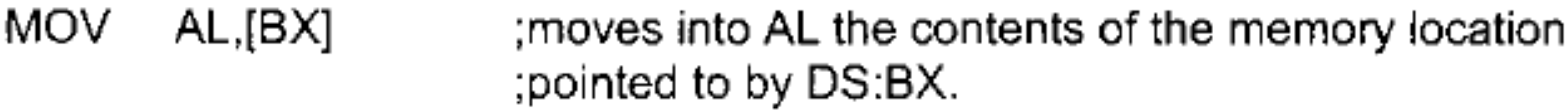


Example:
MOV AL,[03]

AL=?

Register Indirect Addressing Mode

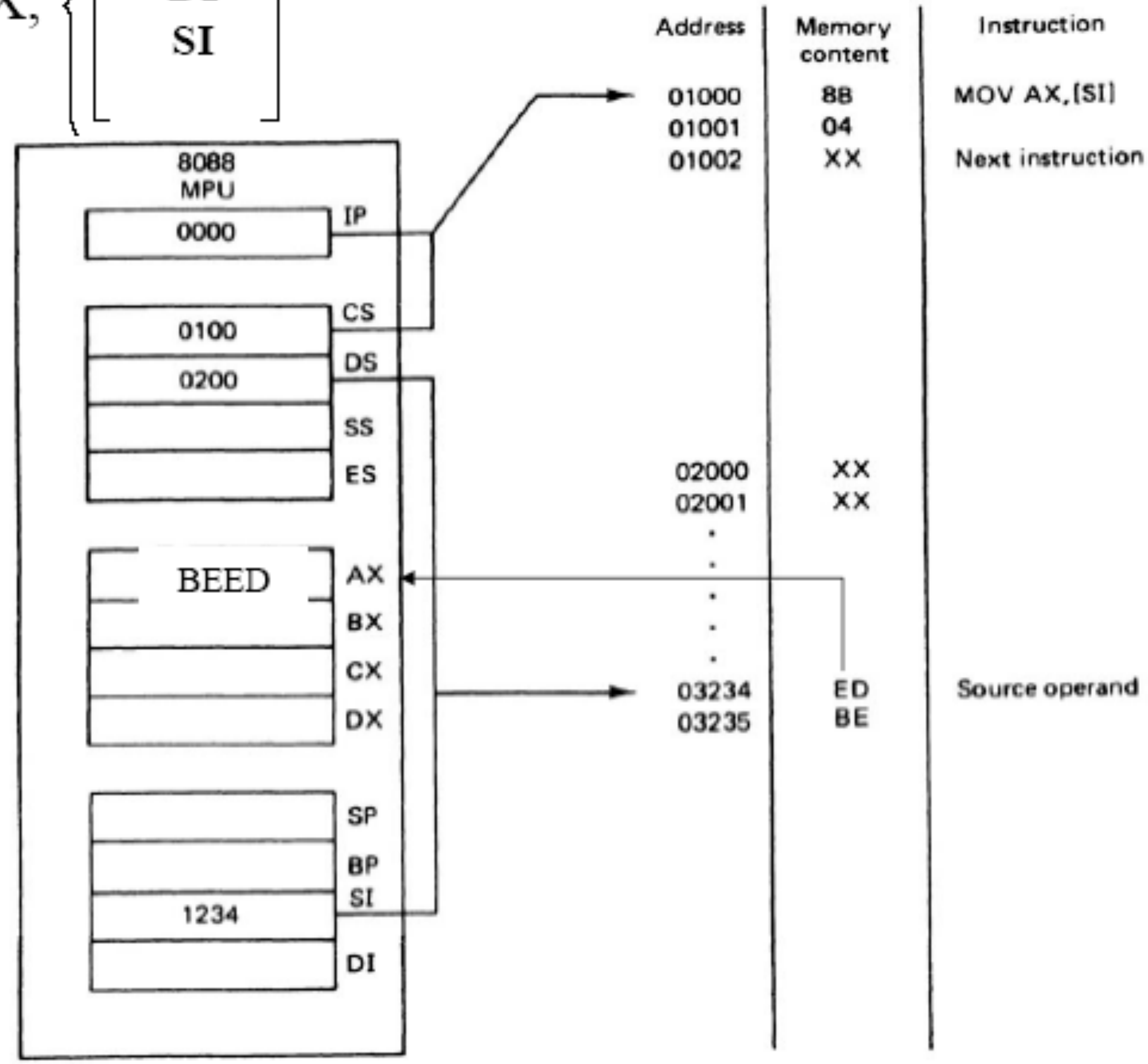
- Data is stored in memory and the address is held by a register
  - Offset address in the data segment (**DS**) by default
  - Registers for this purpose are **SI**, **DI**, and **BX**
  - Need to access memory to gain the data
  - E.g.,



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

Register Indirect Addressing Mode

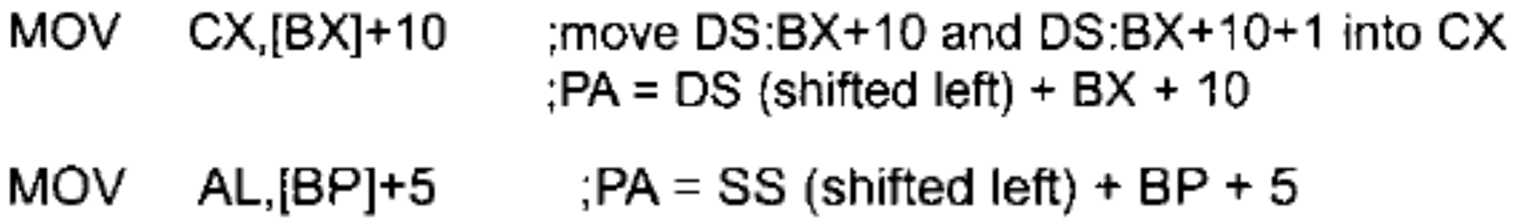
MOV AX, [BX
DI
SI]



Address	Memory content	Instruction
01000	8B	MOV AX,[SI]
01001	04	
01002	XX	Next instruction
02000	XX	
02001	XX	
.		
.		
.		
.		
.		
03234	ED	Source operand
03235	BE	

Based Relative Addressing Mode

- Data is stored in memory and the address can be calculated with base registers **BX** and **BP** as well as a displacement value
  - The default segment is data segment (**DS**) for **BX**, stack segment (**SS**) for **BP**
  - Need to access memory to gain the data
  - E.g.,

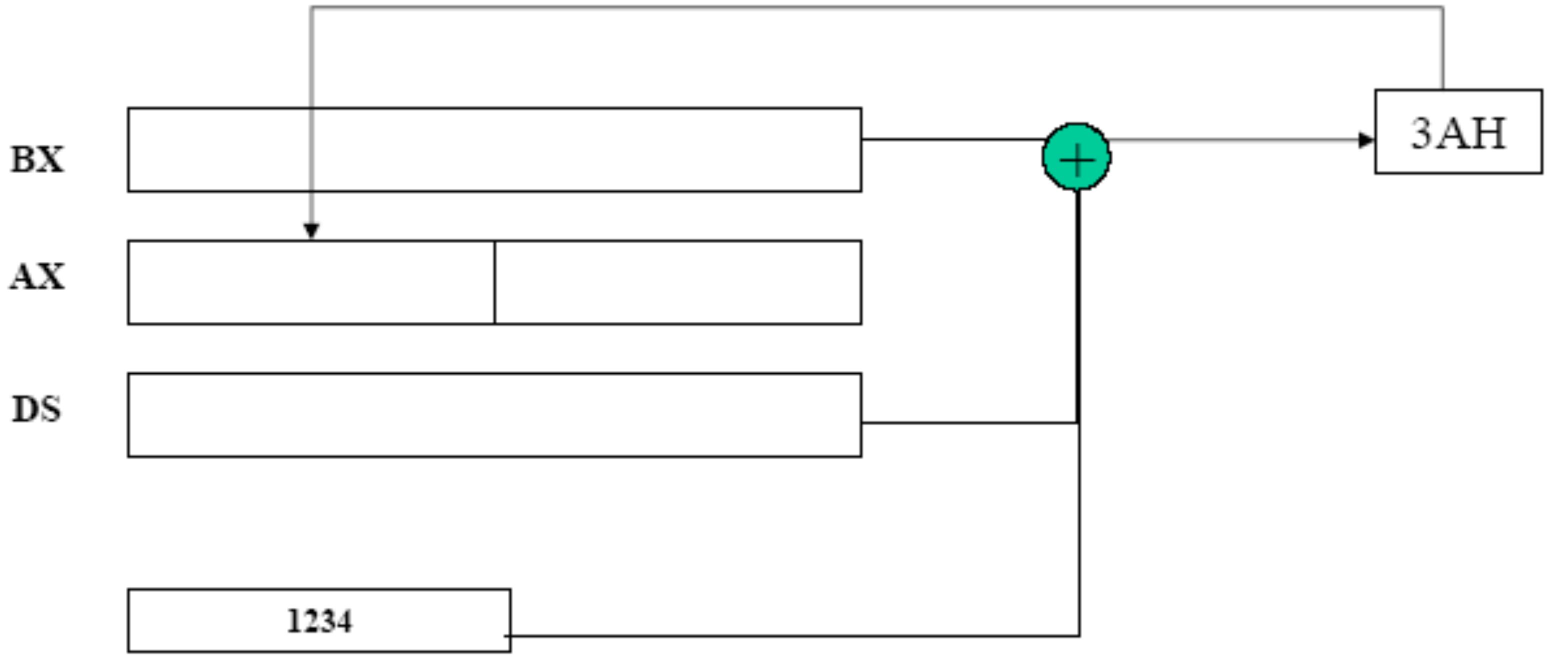


```
MOV    CX,[BX]+10      ;move DS:BX+10 and DS:BX+10+1 into CX
                       ;PA = DS (shifted left) + BX + 10

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

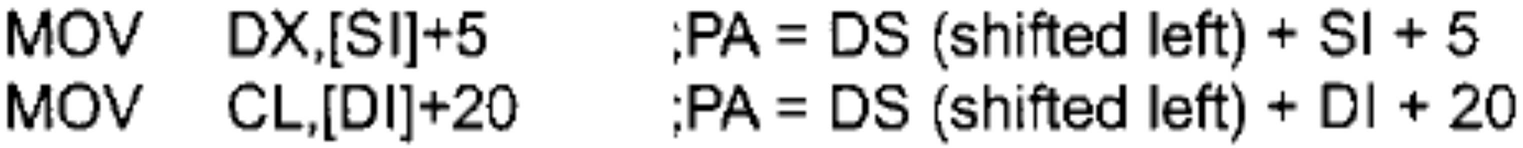
Based-Relative Addressing Mode

$$\text{MOV AH, } \left[\begin{matrix} \textbf{DS:BX} \\ \textbf{SS:BP} \end{matrix} \right] + \text{1234h}$$



Indexed Relative Addressing Mode

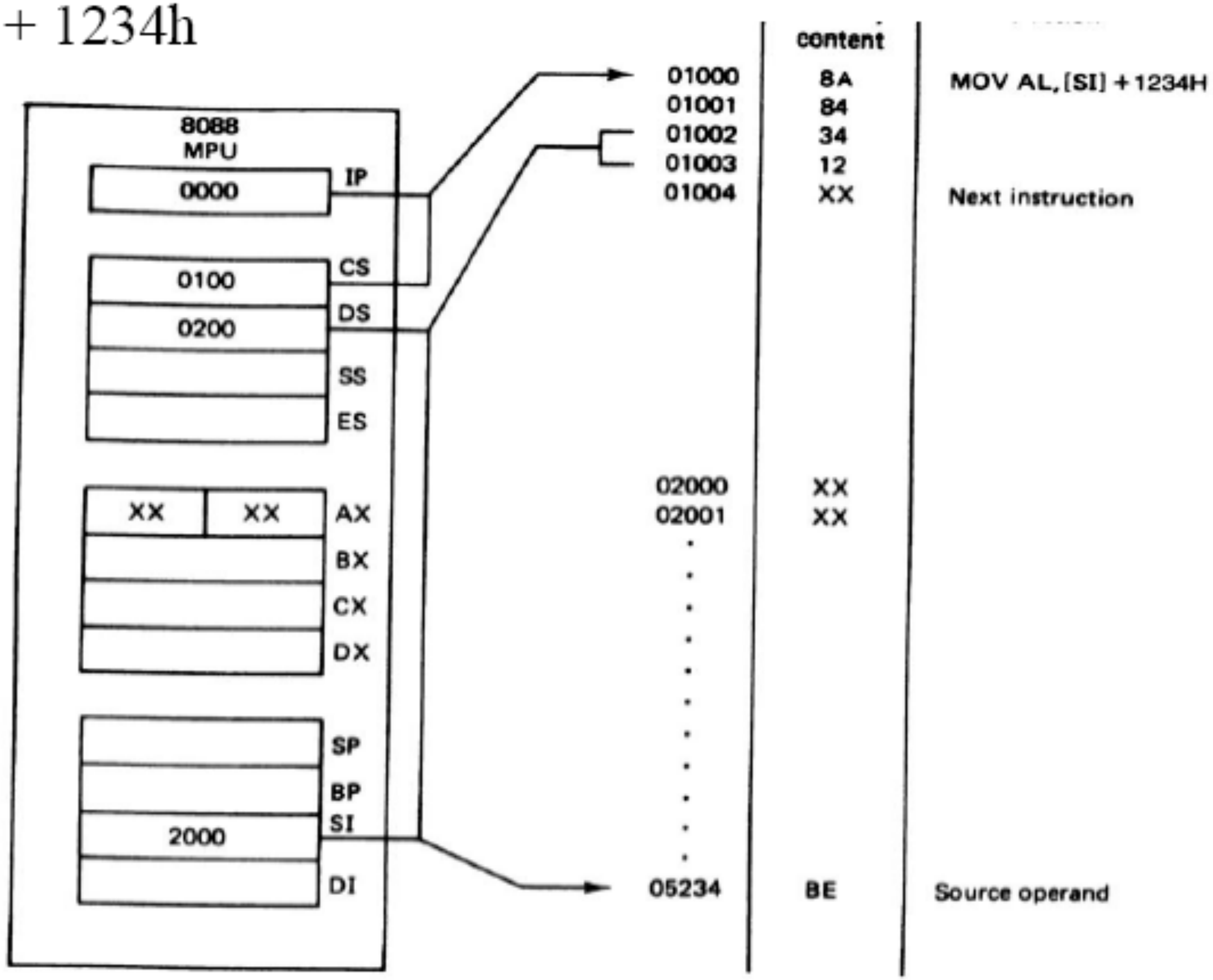
- Data is stored in memory and the address can be calculated with index registers **DI** and **SI** as well as a displacement value
  - The default segment is data segment (**DS**)
  - Need to access memory to gain the data
  - E.g.,



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

Indexed Relative Addressing Mode

MOV AH, [$\begin{smallmatrix}\textbf{SI}\\\textbf{DI}\end{smallmatrix}$] + 1234h

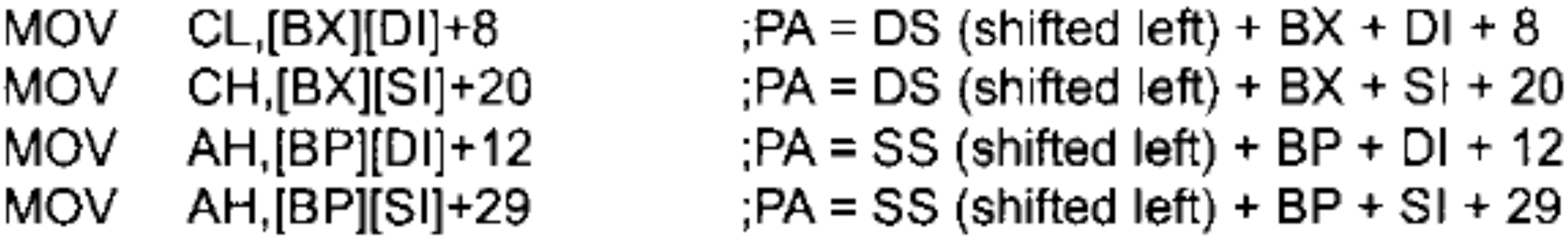


	content	
01000	8A	MOV AL,[SI] +1234H
01001	84	
01002	34	
01003	12	
01004	XX	Next instruction
02000	XX	
02001	XX	
⋮		
05234	BE	Source operand

Example: What is the physical address MOV [DI-8],BL if DS=200 & DI=30h ?
DS:200 shift left once 2000 + DI + -8 = 2028

Based Indexed Relative Addressing Mode

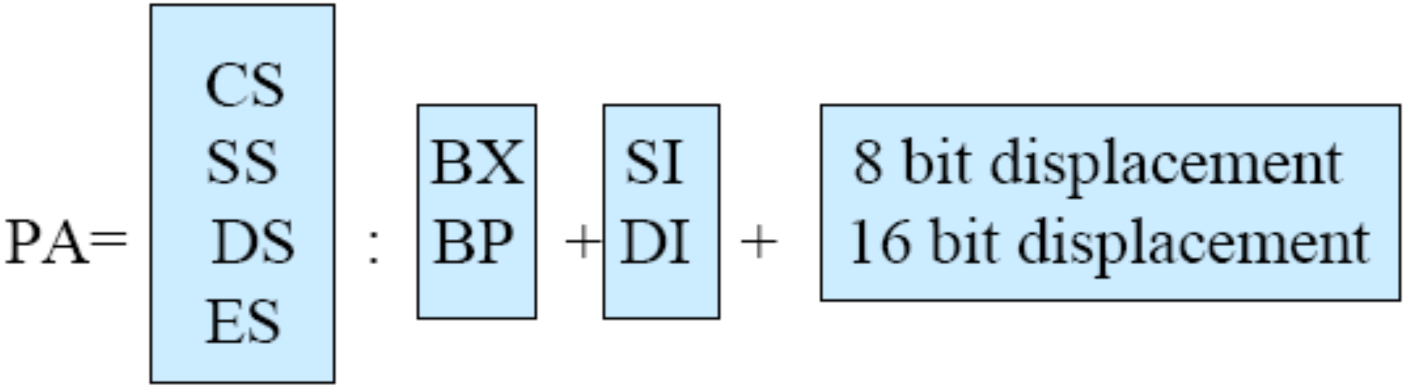
- Combines based and indexed addressing modes, one base register and one index register are used
  - The default segment is data segment (**DS**) for **BX**, stack segment (**SS**) for **BP**
  - Need to access memory to gain the data
  - E.g.,



```
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
```

Based-Indexed Relative Addressing Mode

- Based Relative + Indexed Relative
- We must calculate the PA (physical address)



PA=

CS
SS
DS
ES

:

BX
BP

+

SI
DI

+

8 bit displacement
16 bit displacement

```
MOV AH,[BP+SI+29]
or
MOV AH,[SI+29+BP]
or
MOV AH,[SI][BP]+29
```

The register order does not matter

Segment Overrides

- Offset registers are used with default segment registers

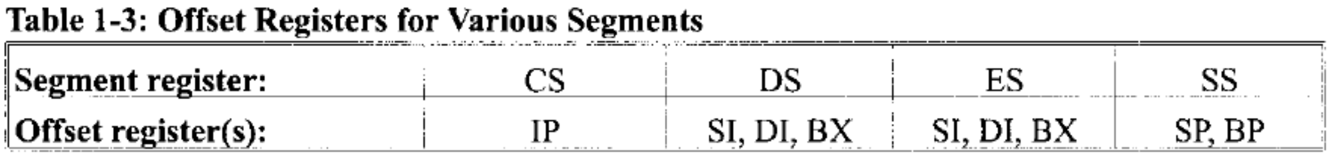
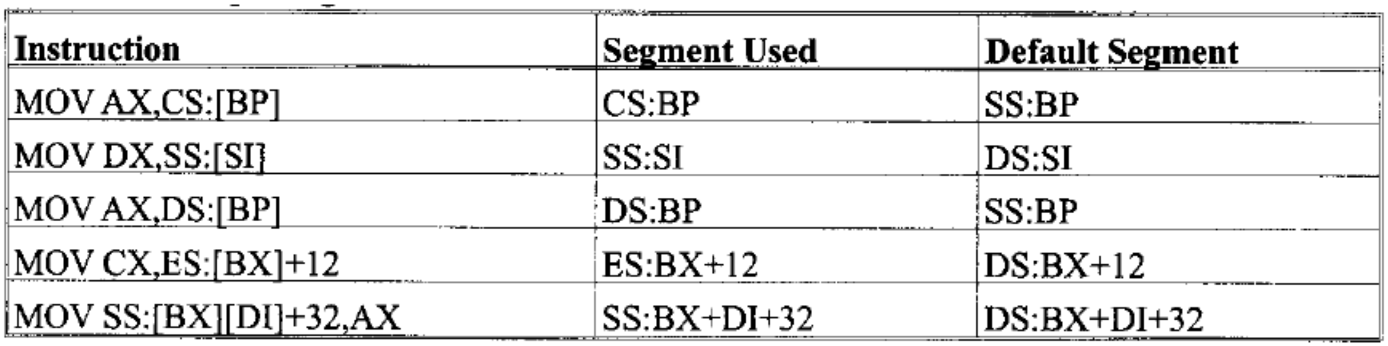


Table 1-3: Offset Registers for Various Segments

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

- 80X86 allows the program to override the default segment registers
  - Specify the segment register in the code



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