
Introduction to Assembly Language Programming

Overview of Assembly Language

❑ Advantages:

- ✓ Faster as compared to programs written using high-level languages
- ✓ Efficient memory usage
- ✓ Control down to bit level

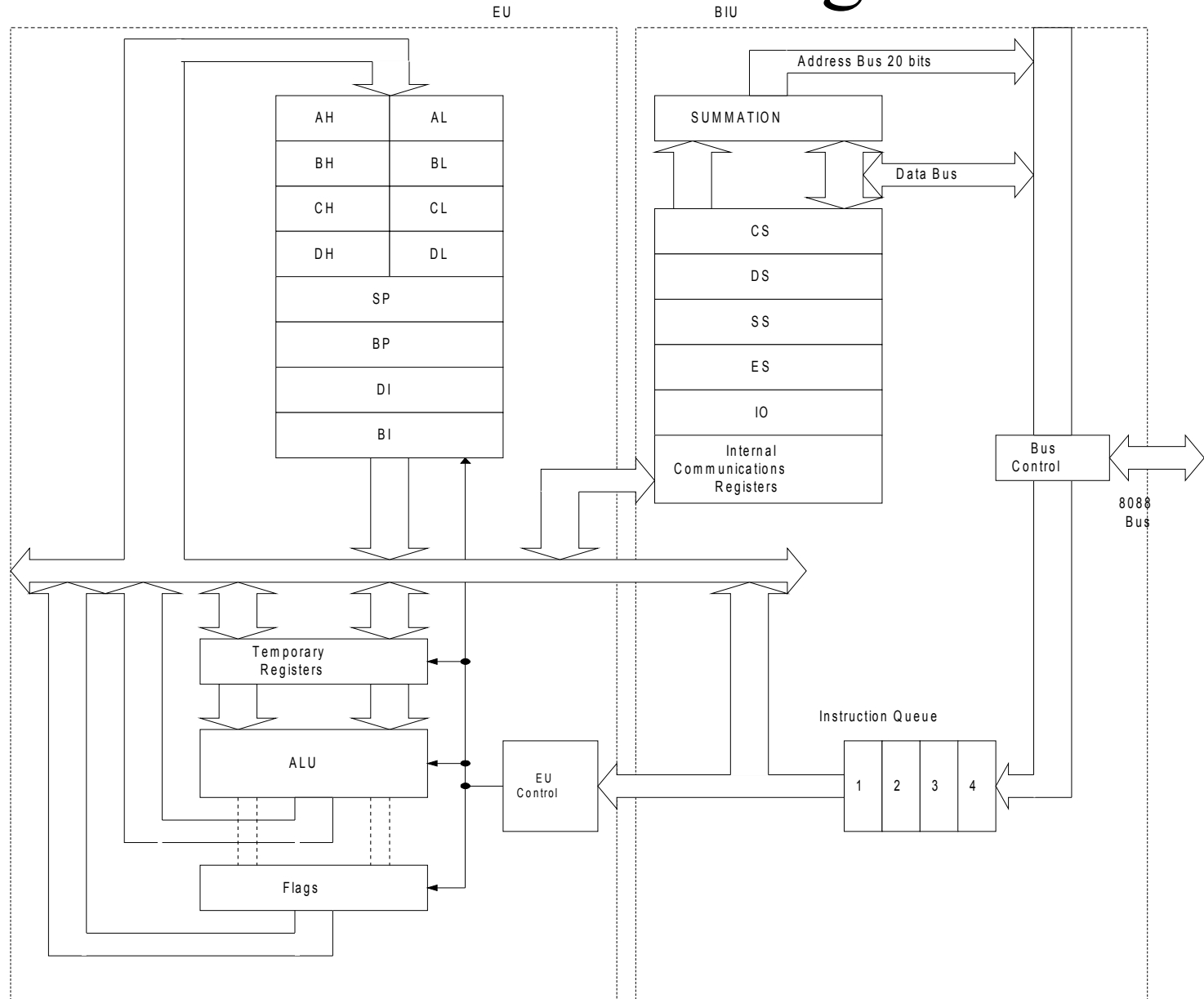
❑ Disadvantages:

- × Need to know detail hardware implementation
- × Not portable
- × Slow to development and difficult to debug

❑ Basic components in assembly Language:

Instruction, Directive, Label, and Comment

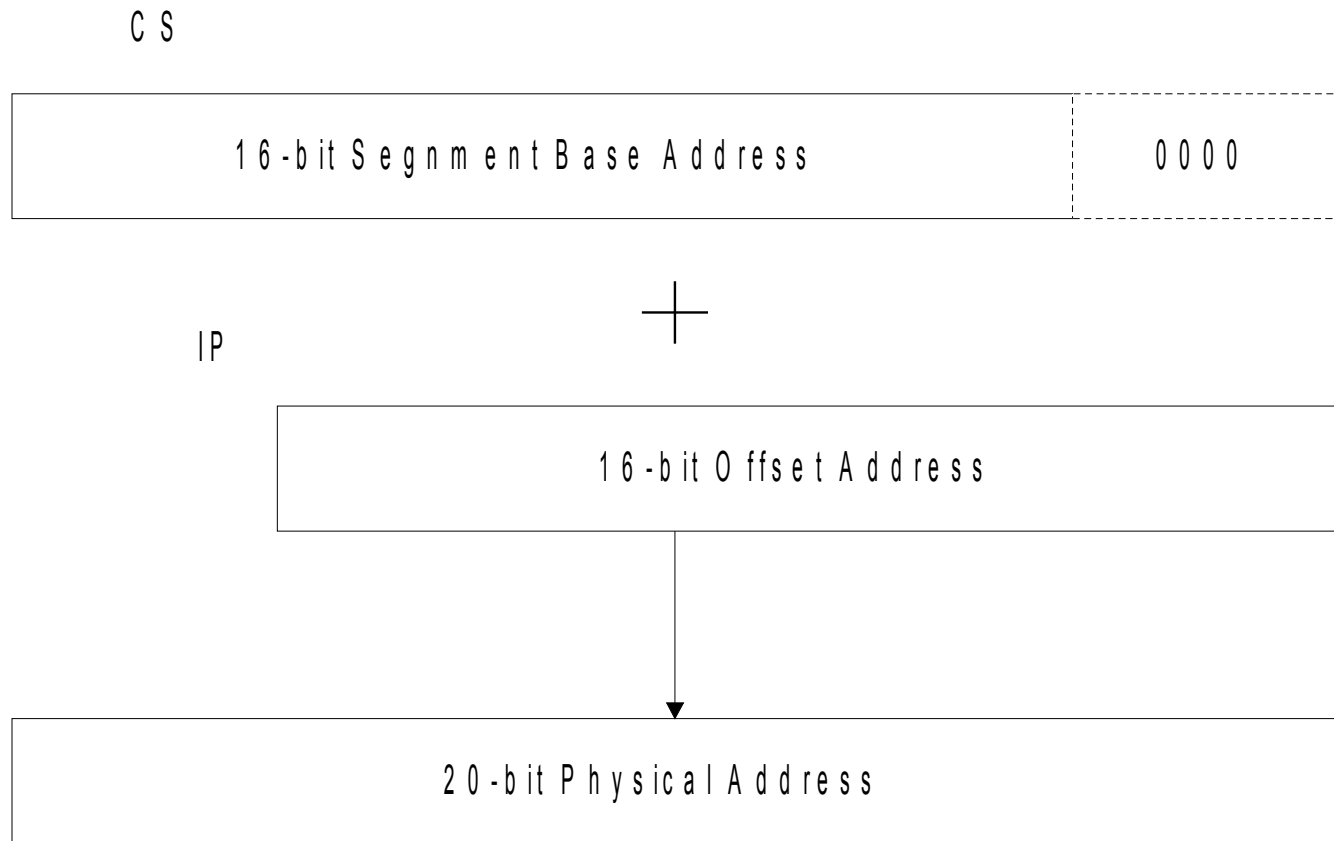
8086/8088 Internal Organisation



BIU Elements

- Instruction Queue: the next instructions or data can be fetched from memory while the processor is executing the current instruction
 - The memory interface is slower than the processor execution time so this speeds up overall performance
- Segment Registers:
 - CS, DS, SS and ES are 16b registers
 - Used with the 16b Base registers to generate the 20b address
 - Allow the 8086/8088 to address 1MB of memory
 - Changed under program control to point to different segments as a program executes
- Instruction Pointer (IP) contains the Offset Address of the next instruction, the distance in bytes from the address given by the current CS register

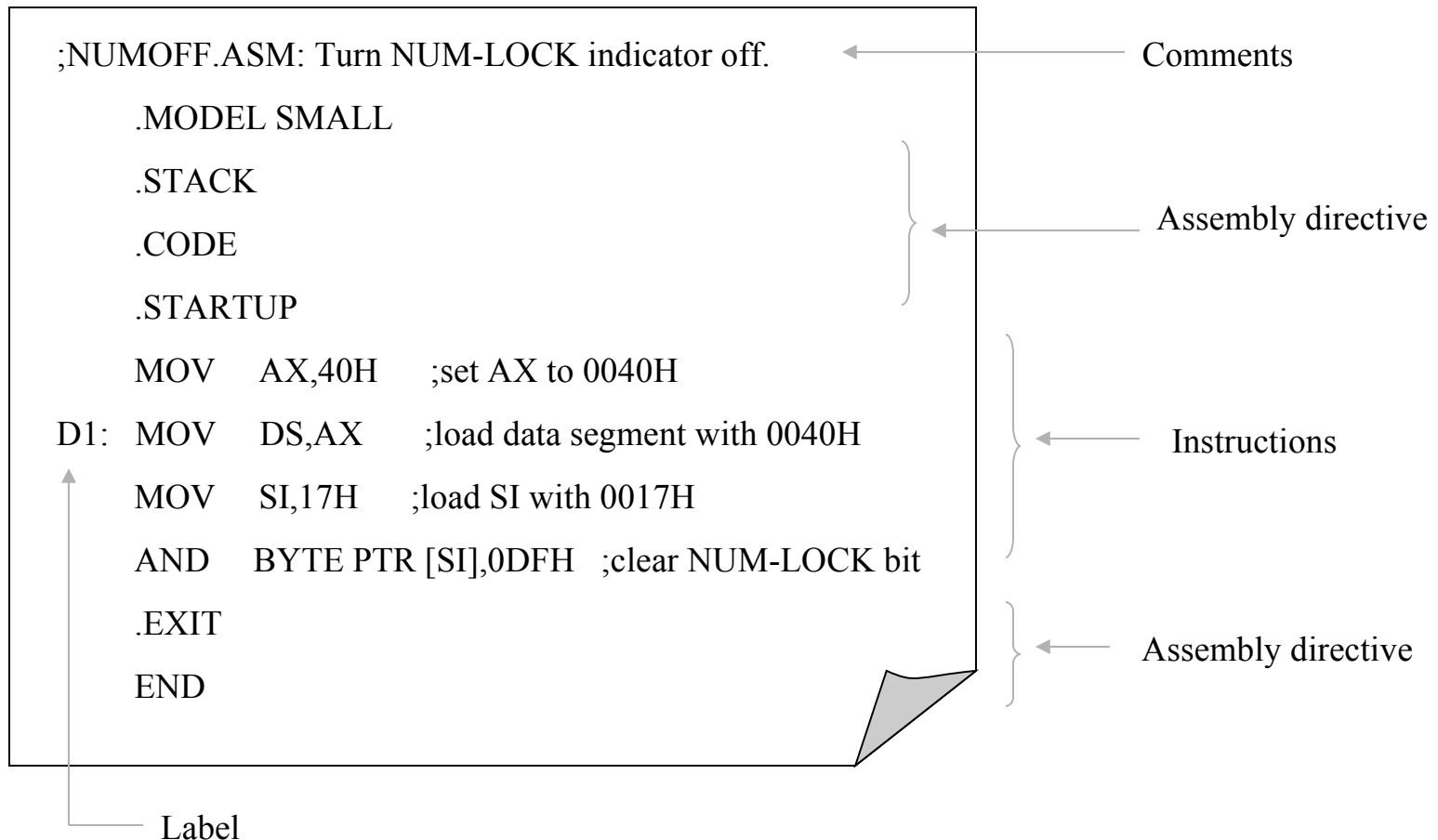
8086/8088 20-bit Addresses



Exercise: 20-bit Addressing

1. CS contains 0A820h, IP contains 0CE24h.
What is the resulting physical address?
2. CS contains 0B500h, IP contains 0024h.
What is the resulting physical address?

Example of Assembly Language Program



Instruction Format

❑ General Format of Instructions

Label: Opcode Operands ; Comment

- **Label:** It is optional. It provides a symbolic address that can be used in branch instructions
- **Opcode:** It specifies the type of instructions
- **Operands:** Instructions of 80x86 family can have one, two, or zero operand
- **Comments:** Only for programmers' reference

❑ Machine Code Format

Opcode	Mode	Operand1	Operand2
--------	------	----------	----------

MOV AL, BL ➡ 1 0 0 0 1 0 0 0 1 1 0 0 0 0 1 1

 MOV Register mode

What is the Meaning of Addressing Modes?

- When a CPU executes an instruction, it needs to know where to get data and where to store results. Such information is specified in the operand fields of the instruction.

1 0 0 0 1 0 0 0 1 1 0 0 0 0 1 1 \Longrightarrow MOV AL, BL

Opcode	Mode	Operand1	Operand2
--------	------	----------	----------

- An operand can be:
 - A datum
 - A register location
 - A memory location
- Addressing modes define how the CPU finds where to get data and where to store results

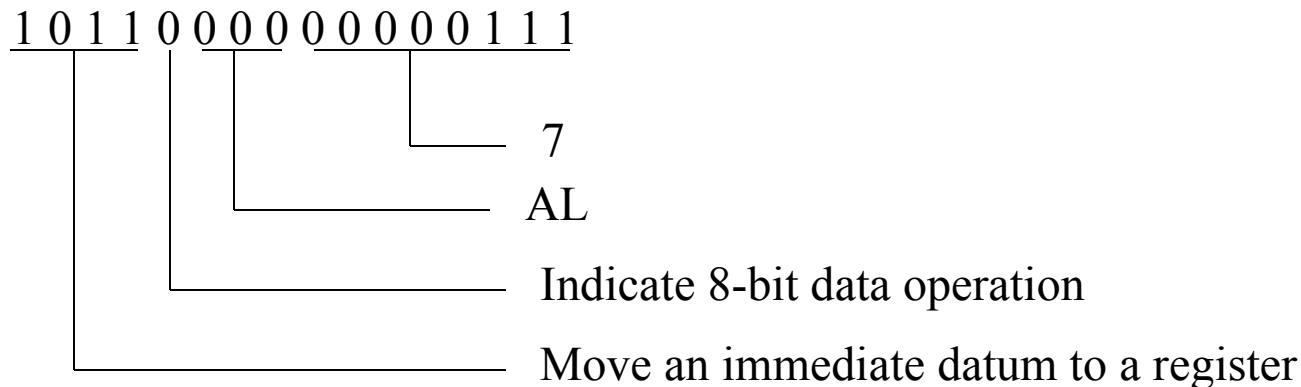
Immediate Addressing

- Data needed by the processor is contained in the instruction

For Example: *move 7 to register AL*



Machine code of MOV AL, 7



Register Addressing

- Operands of the instruction are the names of internal register
- The processor gets data from the register locations specified by instruction operands

For Example: *move the value of register BL to register AL*



- ❑ If AX = 1000H and BX=A080H, after the execution of MOV AL, BL what are the new values of AX and BX?

In immediate and register addressing modes, the processor does not access memory. Thus, the execution of such instructions are fast.

Direct Addressing

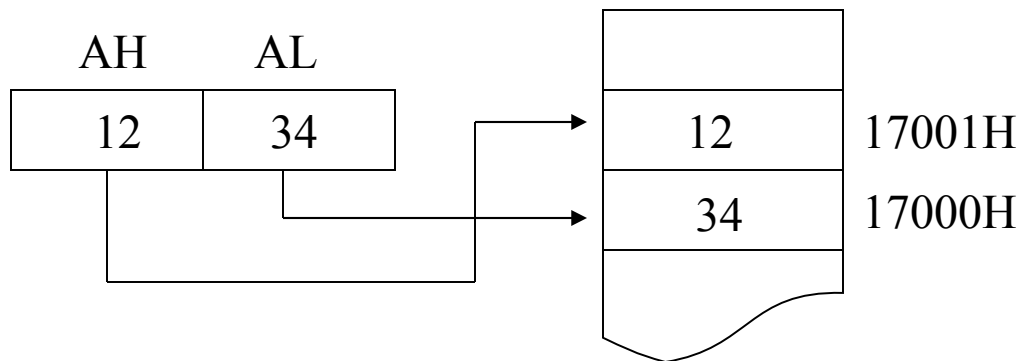
- The processor accesses a memory location
- The memory location is determined by the value of segment register DS and the displacement (offset) specified in the instruction operand field

$$DS \times 10H + \text{Displacement} = \text{Memory location}$$

— Example: *assume DS = 1000H, AX = 1234H*

MOV [7000H], AX

$$\begin{array}{r} \text{DS: } 1\ 0\ 0\ 0\ _ \\ + \text{ Disp: } 7\ 0\ 0\ 0 \\ \hline 1\ 7\ 0\ 0\ 0 \end{array}$$



Register Indirect Addressing

- One of the registers BX, BP, SI, DI appears in the instruction operand field. Its value is used as the memory displacement value.

For Example: MOV DL, [SI]

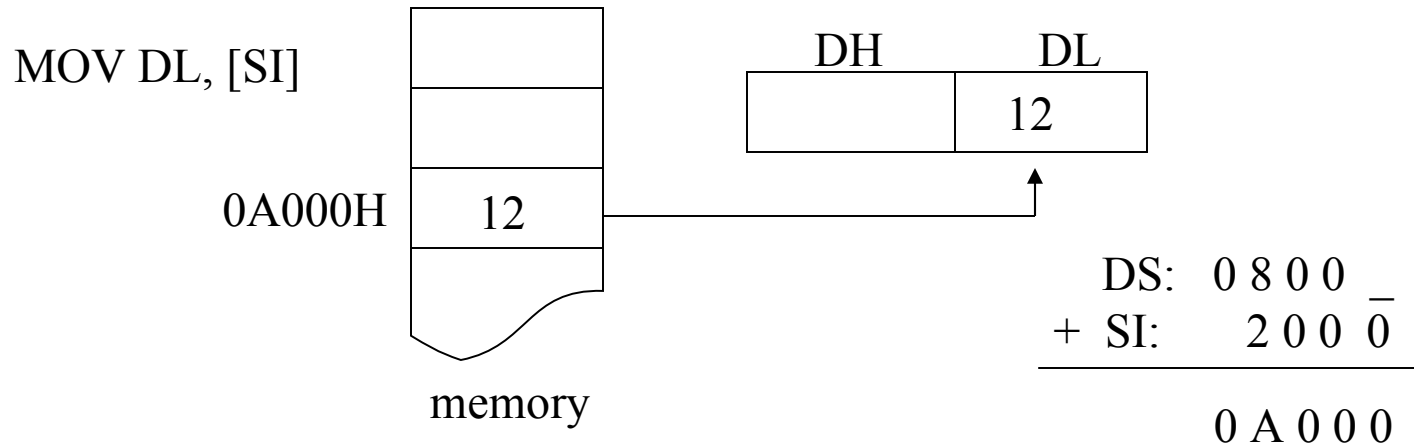
- Memory address is calculated as following:

$$\begin{bmatrix} \text{DS} \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \text{SI} \\ \text{DI} \\ \text{BP} \end{bmatrix} = \text{Memory address}$$

- ❑ If BX, SI, or DI appears in the instruction operand field, segment register DS is used in address calculation
- ❑ If BP appears in the instruction operand field, segment register SS is used in address calculation

Register Indirect Addressing

- Example 1: *assume DS = 0800H, SI=2000H*



- Example 2: *assume SS = 0800H, BP=2000H, DL = 7*

MOV [BP], DL



Based Addressing

- The operand field of the instruction contains a base register (BX or BP) and an 8-bit (or 16-bit) constant (displacement)

For Example: MOV AX, [BX+4]

- Calculate memory address

$$\begin{bmatrix} \text{DS} \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \text{BP} \end{bmatrix} + \text{Displacement} = \text{Memory address}$$

- ☐ If BX appears in the instruction operand field, segment register DS is used in address calculation
- ☐ If BP appears in the instruction operand field, segment register SS is used in address calculation

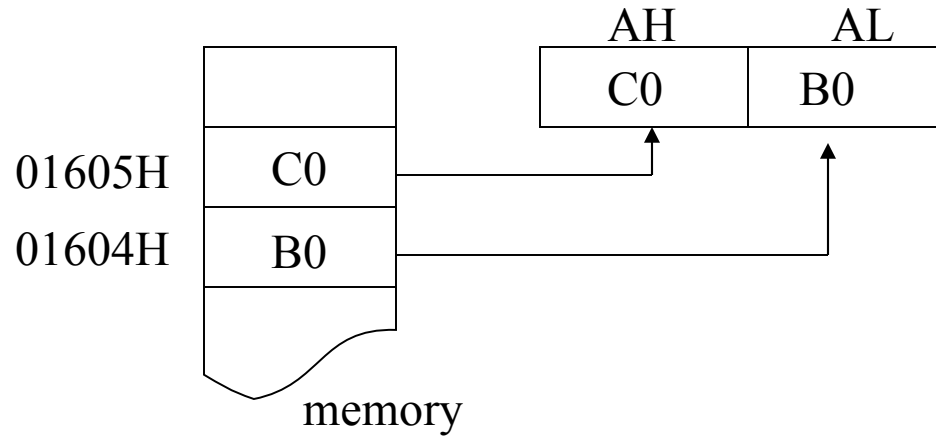
What's difference between register indirect addressing and based addressing?

Based Addressing

- Example 1: *assume DS = 0100H, BX=0600H*

MOV AX, [BX+4]

DS:	0	1	0	0	—
+ BX:	0	6	0	0	
+ Disp.:	0	0	0	4	
<hr/>					
	0	1	6	0	4



- Example 2: *assume SS = 0A00H, BP=0012H, CH = ABH*

MOV [BP-7], CH



Indexed Addressing

- The operand field of the instruction contains an index register (SI or DI) and an 8-bit (or 16-bit) constant (displacement)

For Example: MOV [DI-8], BL

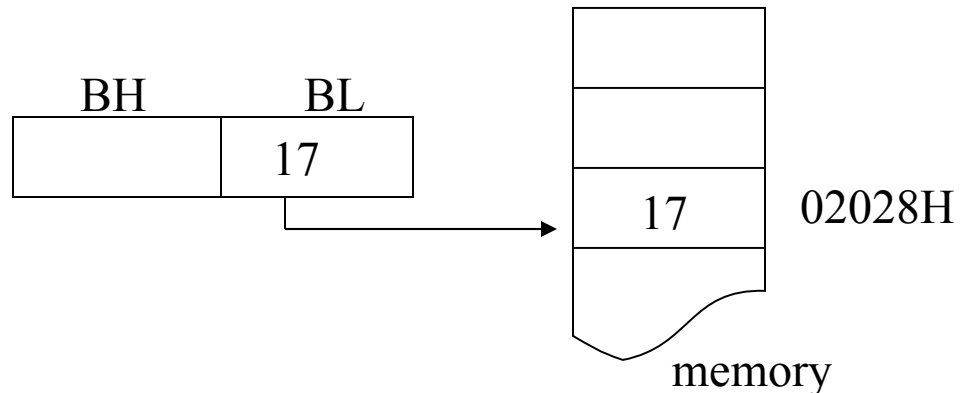
- Calculate memory address

$$DS \times 10H + \left[\begin{array}{c} SI \\ DI \end{array} \right] + \text{Displacement} = \text{Memory address}$$

- Example: *assume DS = 0200H, DI=0030H BL = 17H*

MOV [DI-8], BL

$$\begin{array}{r} DS: 0200\text{H} \\ + DI: 0030\text{H} \\ - Disp.: 0008\text{H} \\ \hline 02028\text{H} \end{array}$$



Based Indexed Addressing

- The operand field of the instruction contains a base register (BX or BP) and an index register

For Example: MOV [BP] [SI], AH
 or MOV [BP+SI], AH

- Calculate memory address

$$\begin{bmatrix} \text{DS} \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \text{BP} \end{bmatrix} + \{\text{SI or DI}\} = \text{Memory address}$$

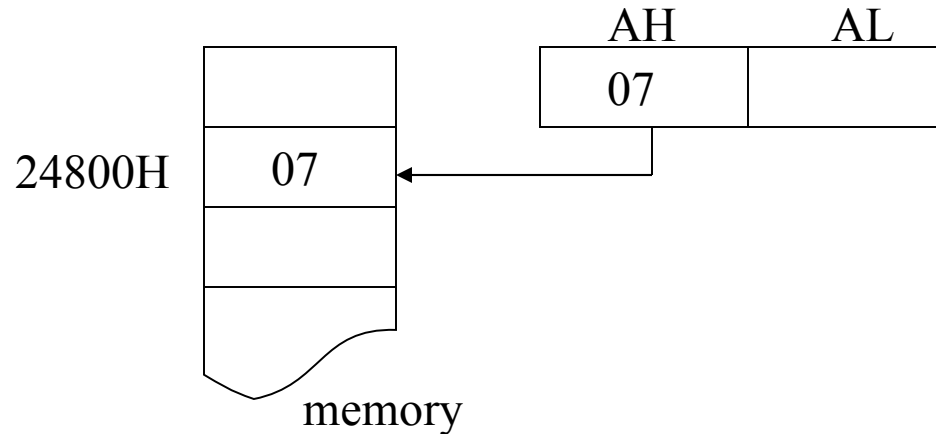
- ☐ If BX appears in the instruction operand field, segment register DS is used in address calculation
- ☐ If BP appears in the instruction operand field, segment register SS is used in address calculation

Based Indexed Addressing

- Example 1: *assume SS = 2000H, BP=4000H, SI=0800H, AH=07H*

MOV [BP] [SI], AH

SS:	2	0	0	0	̄
+ BP:	4	0	0	0	
+ SI:	0	8	0	0	
<hr/>					
	2	4	8	0	0



- Example 2: *assume DS = 0B00H, BX=0112H, DI = 0003H, CH=ABH*

MOV [BX+DI], CH



Based Indexed with Displacement Addressing

- The operand field of the instruction contains a base register (BX or BP), an index register, and a displacement

For Example: `MOV CL, [BX+DI+2080H]`

- Calculate memory address

$$\begin{bmatrix} \text{DS} \\ \text{SS} \end{bmatrix} \times 10\text{H} + \begin{bmatrix} \text{BX} \\ \text{BP} \end{bmatrix} + \{\text{SI or DI}\} + \text{Disp.} = \text{Memory address}$$

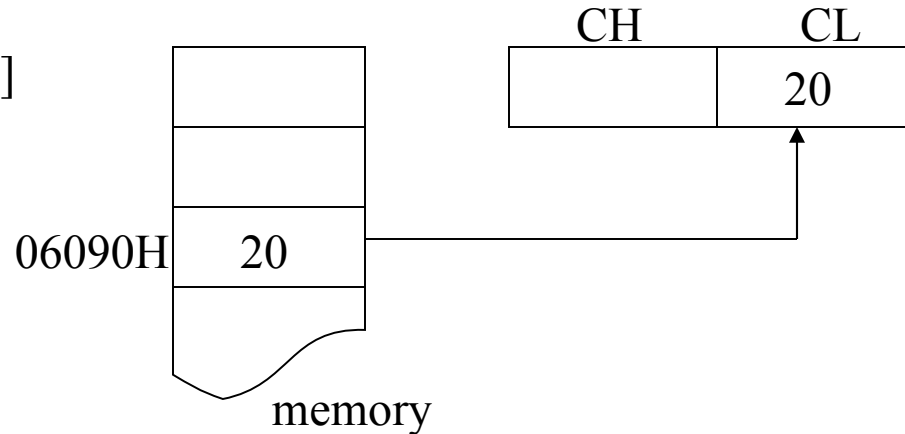
- ☐ If BX appears in the instruction operand field, segment register DS is used in address calculation
- ☐ If BP appears in the instruction operand field, segment register SS is used in address calculation

Based Indexed with Displacement Addressing

- Example 1: *assume DS = 0300H, BX=1000H, DI=0010H*

MOV CL, [BX+DI+2080H]

DS:	0	3	0	0	—
+ BX:	1	0	0	0	
+ DI:	0	0	1	0	
+ Disp.	2	0	8	0	
<hr/>					
	0	6	0	9	0



- Example 2: *assume SS = 1100H, BP=0110H, SI = 000AH, CH=ABH*

MOV [BP+SI+0010H], CH



Instruction Types

- ☐ Data transfer instructions
- ☐ String instructions
- ☐ Arithmetic instructions
- ☐ Bit manipulation instructions
- ☐ Loop and jump instructions
- ☐ Subroutine and interrupt instructions
- ☐ Processor control instructions

An excellent website about 80x86 instruction set: <http://www.penguin.cz/~literakl/intel/intel.html>
Another good reference is in the tutorial of 8086 emulator

Addressing Modes

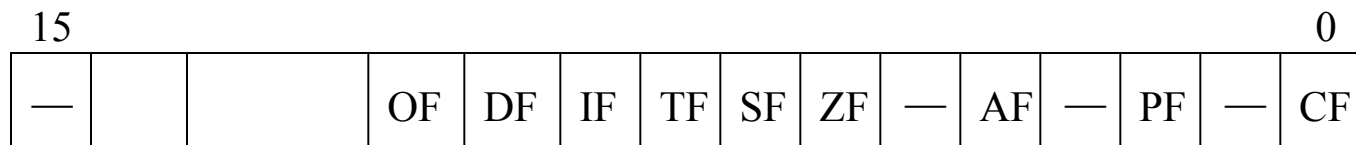
<i>Addressing Modes</i>	<i>Examples</i>
<input type="checkbox"/> Immediate addressing	MOV AL, 12H
<input type="checkbox"/> Register addressing	MOV AL, BL
<input type="checkbox"/> Direct addressing	MOV [500H], AL
<input type="checkbox"/> Register Indirect addressing	MOV DL, [SI]
<input type="checkbox"/> Based addressing	MOV AX, [BX+4]
<input type="checkbox"/> Indexed addressing	MOV [DI-8], BL
<input type="checkbox"/> Based indexed addressing	MOV [BP+SI], AH
<input type="checkbox"/> Based indexed with displacement addressing	MOV CL, [BX+DI+2]

Exceptions

- ☐ String addressing
- ☐ Port addressing (e.g. IN AL, 79H)

Flag Register

- ❑ Flag register contains information reflecting the current status of a microprocessor. It also contains information which controls the operation of the microprocessor.



➤ Control Flags

IF: Interrupt enable flag
DF: Direction flag
TF: Trap flag

➤ Status Flags

CF: Carry flag
PF: Parity flag
AF: Auxiliary carry flag
ZF: Zero flag
SF: Sign flag
OF: Overflow flag

Flags Commonly Tested During the Execution of Instructions

- ❑ There are five flag bits that are commonly tested during the execution of instructions
 - Sign Flag (Bit 7), SF: 0 for positive number and 1 for negative number
 - Zero Flag (Bit 6), ZF: If the ALU output is 0, this bit is set (1); otherwise, it is 0
 - Carry Flag (Bit 0), CF: It contains the carry generated during the execution
 - Auxiliary Carry, AF: (Bit 4) Depending on the width of ALU inputs, this flag bit contains the carry generated at bit 3 (or, 7, 15) of the 8088 ALU
 - Parity Flag (bit2), PF: It is set (1) if the output of the ALU has even number of ones; otherwise it is zero


Data Transfer Instructions

❑ *MOV Destination, Source*

- Move data from source to destination; *e.g.* **MOV [DI+100H], AH**
- It does not modify flags
- For 80x86 family, directly moving data from one memory location to another memory location is not allowed

MOV [SI], [5000H] 


- When the size of data is not clear, assembler directives are used

MOV [SI], 0 

- **BYTE PTR**
- **WORD PTR**
- **DWORD PTR**

MOV BYTE PTR [SI], 12H
MOV WORD PTR [SI], 12H
MOV DWORD PTR [SI], 12H

- You can not move an immediate data to segment register by MOV

MOV DS, 1234H 

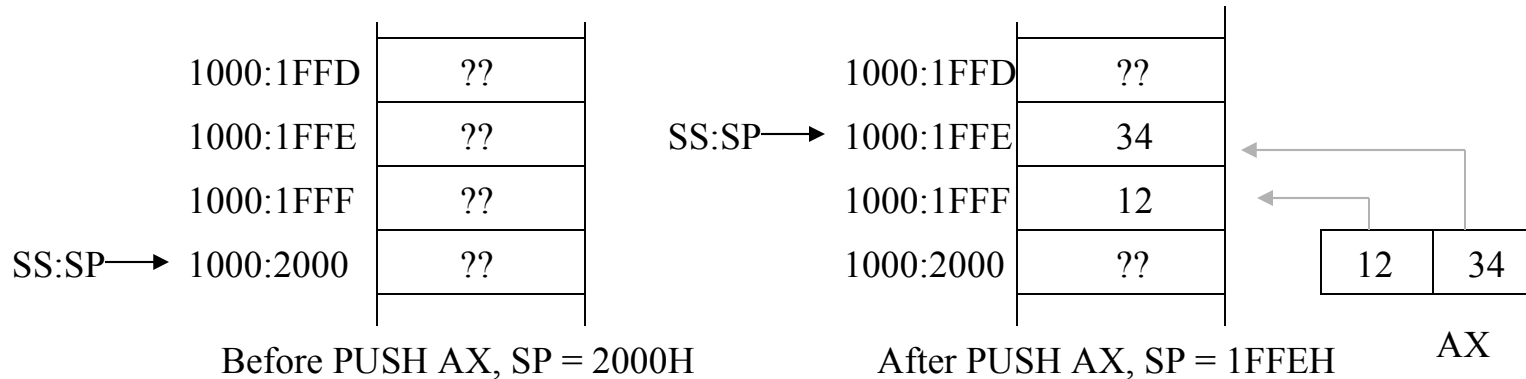
Instructions for Stack Operations

❑ What is a Stack ?

- A stack is a collection of memory locations. It always follows the rule of last-in-first-out
- Generally, SS and SP are used to trace where is the latest data written into stack

❑ PUSH *Source*

- Push data (**word**) onto stack
- It does not modify flags
- For Example: PUSH AX (assume ax=1234H, SS=1000H, SP=2000H before PUSH AX)



➤ Decrementing the stack pointer during a push is a standard way of implementing stacks in hardware

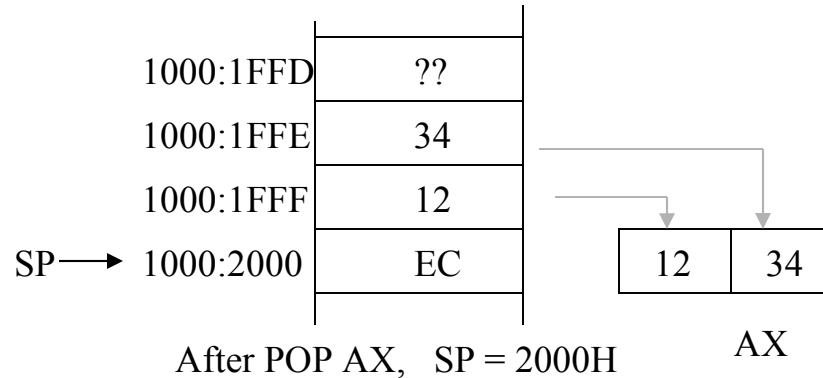
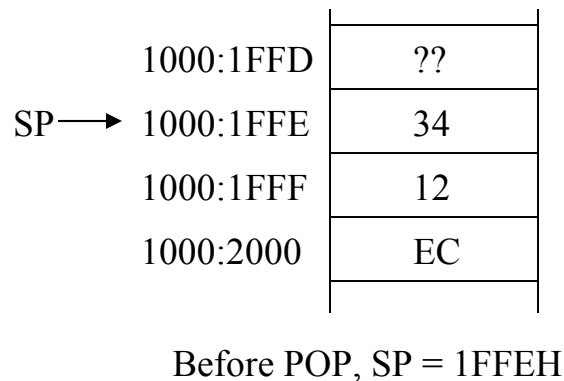
Instructions for Stack Operations

❑ PUSHF

- Push the values of the flag register onto stack
- It does not modify flags

❑ POP *Destination*

- Pop word off stack
- It does not modify flags
- For example: **POP AX**



❑ POPF

- Pop word from the stack to the flag register
- It modifies all flags

Data Transfer Instructions

❑ SAHF

- Store data in AH to the low 8 bits of the flag register
- It modifies flags: AF, CF, PF, SF, ZF

❑ LAHF

- Copies bits 0-7 of the flags register into AH
- It does not modify flags

❑ LDS *Destination Source*

- Load 4-byte data (pointer) in memory to two 16-bit registers
- Source operand gives the memory location
- The first two bytes are copied to the register specified in the destination operand; the second two bytes are copied to register DS
- It does not modify flags

❑ LES *Destination Source*

- It is identical to LDS except that the second two bytes are copied to ES
- It does not modify flags

Data Transfer Instructions

❑ LEA *Destination Source*

- Transfers the offset address of source (must be a memory location) to the destination register
- It does not modify flags

❑ XCHG *Destination Source*

- It exchanges the content of destination and source
- One operand must be a microprocessor register, the other one can be a register or a memory location
- It does not modify flags

❑ XLAT

- Replace the data in AL with a data in a user defined look-up table
- BX stores the beginning address of the table
- At the beginning of the execution, the number in AL is used as the index of the look-up table
- It does not modify flags

String Instructions

- ❑ String is a collection of bytes, words, or long-words that can be up to 64KB in length
- ❑ String instructions can have at most two operands. One is referred to as source string and the other one is called destination string
 - Source string must locate in Data Segment and SI register points to the current element of the source string
 - Destination string must locate in Extra Segment and DI register points to the current element of the destination string

DS : SI		
0510:0000	53	S
0510:0001	48	H
0510:0002	4F	O
0510:0003	50	P
0510:0004	50	P
0510:0005	45	E
0510:0006	52	R
Source String		

ES : DI		
02A8:2000	53	S
02A8:2001	48	H
02A8:2002	4F	O
02A8:2003	50	P
02A8:2004	50	P
02A8:2005	49	I
02A8:2006	4E	N
Destination String		

Repeat Prefix Instructions

❑ REP *String Instruction*

- The prefix instruction makes the microprocessor repeatedly execute the string instruction until CX decrements to 0 (During the execution, CX is decreased by one when the string instruction is executed one time).
- For Example:

MOV CX, 5
REP MOVSB

By the above two instructions, the microprocessor will execute MOVSB 5 times.

- Execution flow of REP MOVSB::

While (CX!=0)
{
CX = CX - 1;
MOVSB;
}

OR

Check_CX: If CX!=0 Then
CX = CX - 1;
MOVSB;
goto Check_CX;
end if

Repeat Prefix Instructions

❑ REPZ *String Instruction*

— Repeat the execution of the string instruction until CX=0 or zero flag is clear

❑ REPNZ *String Instruction*

— Repeat the execution of the string instruction until CX=0 or zero flag is set

❑ REPE *String Instruction*

— Repeat the execution of the string instruction until CX=0 or zero flag is clear

❑ REPNE *String Instruction*

— Repeat the execution of the string instruction until CX=0 or zero flag is set

Direction Flag

❑ Direction Flag (DF) is used to control the way SI and DI are adjusted during the execution of a string instruction

— DF=0, SI and DI will auto-increment during the execution; otherwise, SI and DI auto-decrement

— Instruction to set DF: **STD**; Instruction to clear DF: **CLD**

— Example:

CLD

MOV CX, 5

REP MOVSB

At the beginning of execution,
DS=0510H and SI=0000H

DS : SI				
0510:0000	53	S	←	SI _{0x5}
0510:0001	48	H	←	SI _{0x4}
0510:0002	4F	O	←	SI _{0x3}
0510:0003	50	P	←	SI _{0x2}
0510:0004	50	P	←	SI _{0x1}
0510:0005	45	E	←	SI _{0x0}
0510:0006	52	R		
		Source String		

String Instructions

❑ MOVSB (MOVSW)

- Move byte (word) at memory location DS:SI to memory location ES:DI and update SI and DI according to DF and the width of the data being transferred
- It does not modify flags
- Example:

	DS : SI		ES : DI	
MOV AX, 0510H	0510:0000	53	S	0300:0100
MOV DS, AX	0510:0001	48	H	
MOV SI, 0	0510:0002	4F	O	
MOV AX, 0300H	0510:0003	50	P	
MOV ES, AX	0510:0004	50	P	
MOV DI, 100H	0510:0005	45	E	
CLD	0510:0006	52	R	
MOV CX, 5				
REP MOVSB				
	Source String		Destination String	

String Instructions

❑ CMPSB (CMPSW)

- Compare bytes (words) at memory locations DS:SI and ES:DI;
update SI and DI according to DF and the width of the data being compared
- It modifies flags
- Example:

Assume: ES = 02A8H
DI = 2000H
DS = 0510H
SI = 0000H

CLD
MOV CX, 9
REPZ CMPSB

What's the values of CX after
The execution?

DS : SI		
0510:0000	53	S
0510:0001	48	H
0510:0002	4F	O
0510:0003	50	P
0510:0004	50	P
0510:0005	45	E
0510:0006	52	R
Source String		

ES : DI		
02A8:2000	53	S
02A8:2001	48	H
02A8:2002	4F	O
02A8:2003	50	P
02A8:2004	50	P
02A8:2005	49	I
02A8:2006	4E	N
Destination String		

String Instructions

❑ SCASB (SCASW)

- Move byte (word) in AL (AX) and at memory location ES:DI;
update DI according to DF and the width of the data being compared
- It modifies flags

❑ LODSB (LODSW)

- Load byte (word) at memory location DS:SI to AL (AX);
update SI according to DF and the width of the data being transferred
- It does not modify flags

❑ STOSB (STOSW)

- Store byte (word) at in AL (AX) to memory location ES:DI;
update DI according to DF and the width of the data being transferred
- It does not modify flags

Arithmetic Instructions

❑ *ADD Destination, Source*

- $\text{Destination} + \text{Source} \rightarrow \text{Destination}$
- Destination and Source operands can not be memory locations at the same time
- It modifies flags AF CF OF PF SF ZF

❑ *ADC Destination, Source*

- $\text{Destination} + \text{Source} + \text{Carry Flag} \rightarrow \text{Destination}$
- Destination and Source operands can not be memory locations at the same time
- It modifies flags AF CF OF PF SF ZF

❑ *INC Destination*

- $\text{Destination} + 1 \rightarrow \text{Destination}$
- It modifies flags AF OF PF SF ZF (**Note CF will not be changed**)

❑ *DEC Destination*

- $\text{Destination} - 1 \rightarrow \text{Destination}$
- It modifies flags AF OF PF SF ZF (**Note CF will not be changed**)

Arithmetic Instructions

- ❑ *SUB Destination, Source*
 - Destination - Source \rightarrow Destination
 - Destination and Source operands can not be memory locations at the same time
 - It modifies flags AF CF OF PF SF ZF

- ❑ *SBB Destination, Source*
 - Destination - Source - Carry Flag \rightarrow Destination
 - Destination and Source operands can not be memory locations at the same time
 - It modifies flags AF CF OF PF SF ZF

- ❑ *CMP Destination, Source*
 - Destination - Source (the result is not stored anywhere)
 - Destination and Source operands can not be memory locations at the same time
 - It modifies flags AF CF OF PF SF ZF **(if ZF is set, destination = source)**

Arithmetic Instructions

❑ *MUL Source*

- Perform unsigned multiply operation
- If source operand is a byte, **$AX = AL * Source$**
- If source operand is a word, **$(DX\ AX) = AX * Source$**
- Source operands can not be an immediate data
- It modifies CF and OF (AF,PF,SF,ZF undefined)

❑ *IMUL Source*

- Perform signed binary multiply operation
- If source operand is a byte, **$AX = AL * Source$**
- If source operand is a word, **$(DX\ AX) = AX * Source$**
- Source operands can not be an immediate data
- It modifies CF and OF (AF,PF,SF,ZF undefined)

➤ Examples:

```
MOV AL, 20H
MOV CL, 80H
MUL CL
```

```
MOV AL, 20H
MOV CL, 80H
IMUL CL
```


Arithmetic Instructions

□ *DIV Source*

- Perform unsigned division operation
- If source operand is a byte, **AL = AX / Source; AH = Remainder of AX / Source**
- If source operand is a word, **AX=(DX AX)/Source; DX=Remainder of (DX AX)/Source**
- Source operands can not be an immediate data

□ *IDIV Source*

- Perform signed division operation
- If source operand is a byte, **AL = AX / Source; AH = Remainder of AX / Source**
- If source operand is a word, **AX=(DX AX)/Source; DX=Remainder of (DX AX)/Source**
- Source operands can not be an immediate data

➤ Examples:

```
MOV AX, 5  
MOV BL, 2  
DIV BL
```

```
MOV AL, -5  
MOV BL, 2  
IDIV BL
```

Arithmetic Instructions

❑ NEG *Destination*

- 0 – Destination → Destination (the result is represented in 2's complement)
- Destination can be a register or a memory location
- It modifies flags AF CF OF PF SF ZF

❑ CBW

- Extends a signed 8-bit number in AL to a signed 16-bit data and stores it into AX
- It does not modify flags

❑ CWD

- Extends a signed 16-bit number in AX to a signed 32-bit data and stores it into DX and AX. DX contains the most significant word
- It does not modify flags

❖ Other arithmetic instructions:

DAA, DAS, AAA, AAS, AAM, AAD

Logical Instructions

❑ NOT *Destination*

- Inverts each bit of the destination operand
- Destination can be a register or a memory location
- It does not modify flags

❑ AND *Destination, Source*

- Performs logic AND operation for each bit of the destination and source; stores the result into destination
- Destination and source can not be both memory locations at the same time
- It modifies flags: CF OF PF SF ZF

❑ OR *Destination, Source*

- Performs logic OR operation for each bit of the destination and source; stores the result into destination
- Destination and source can not be both memory locations at the same time
- It modifies flags: CF OF PF SF ZF

Logical Instructions

❑ XOR *Destination, Source*

- Performs logic XOR operation for each bit of the destination and source; stores the result into destination
- Destination and source can not be both memory locations at the same time
- It modifies flags: CF OF PF SF ZF

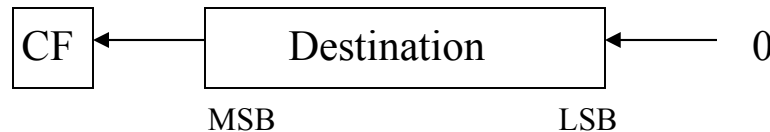
❑ TEST *Destination, Source*

- Performs logic AND operation for each bit of the destination and source
- Updates Flags depending on the result of AND operation
- Do not store the result of AND operation anywhere

Bit Manipulation Instructions

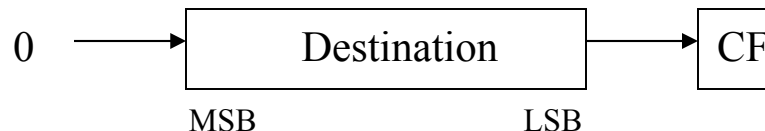
❑ SHL(SAL) *Destination, Count*

- Left shift destination bits; the number of bits shifted is given by operand Count
- During the shift operation, the MSB of the destination is shifted into CF and zero is shifted into the LSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF PF SF ZF



❑ SHR *Destination, Count*

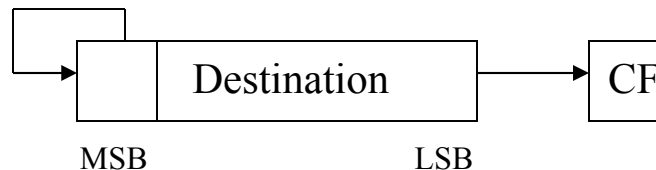
- Right shift destination bits; the number of bits shifted is given by operand Count
- During the shift operation, the LSB of the destination is shifted into CF and zero is shifted into the MSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF PF SF ZF



Bit Manipulation Instructions

❑ SAR *Destination, Count*

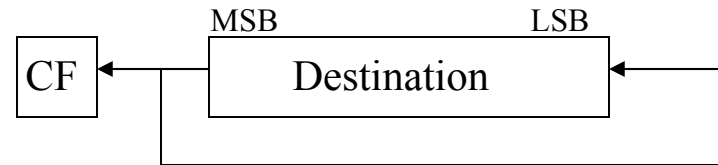
- Right shift destination bits; the number of bits shifted is given by operand Count
- The LSB of the destination is shifted into CF and the MSB of the destination remains the same
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF PF SF ZF



Bit Manipulation Instructions

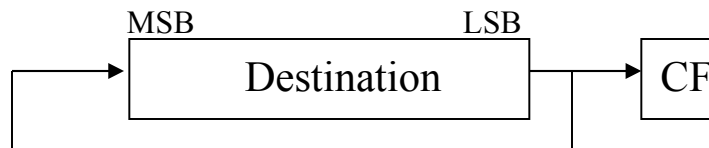
❑ ROL *Destination, Count*

- Left shift destination bits; the number of bits shifted is given by operand Count
- The MSB of the destination is shifted into CF, it also goes to the LSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF



❑ ROR *Destination, Count*

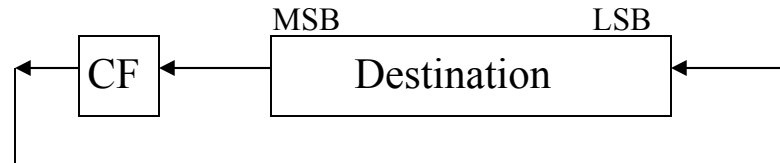
- Right shift destination bits; the number of bits shifted is given by operand Count
- The LSB of the destination is shifted into CF, it also goes to the MSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF



Bit Manipulation Instructions

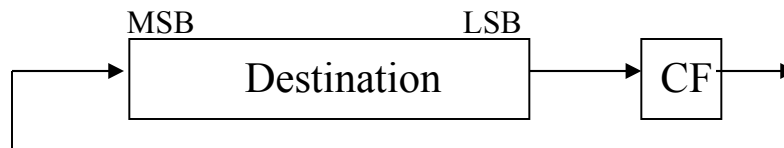
❑ RCL *Destination, Count*

- Left shift destination bits; the number of bits shifted is given by operand Count
- The MSB of the destination is shifted into CF; the old CF value goes to the LSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF PF SF ZF



❑ RCR *Destination, Count*

- Right shift destination bits; the number of bits shifted is given by operand Count
- The LSB of the destination is shifted into CF, the old CF value goes to the MSB of the destination
- Operand Count can be either an immediate data or register CL
- Destination can be a register or a memory location
- It modifies flags: CF OF PF SF ZF

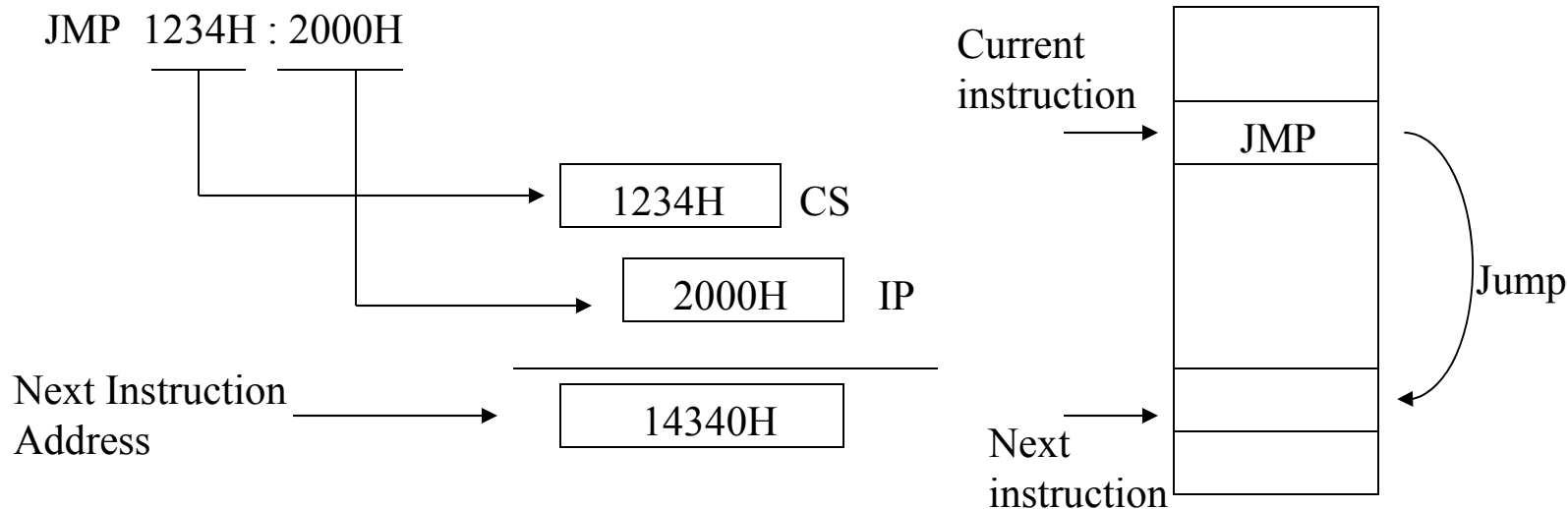


Program Transfer Instructions

❑ *JMP Target*

- Unconditional jump
- It moves microprocessor to execute another part of the program
- Target can be represented by a label, immediate data, registers, or memory locations
- It does not affect flags

➤ The execution of JMP instruction



Program Transfer Instructions

➤ Intrasegment transfer v.s. Intersegment transfer

- Intrasegment transfer: the microprocessor jumps to an address within the same segment
- Intersegment transfer: the microprocessor jumps to an address in a different segment
- Use assembler directive **near** and **far** to indicate the types of JMP instructions
- For intrasegment transfer, we can provide only new IP value in JMP instructions.
For Example: JMP 1000H
- For intersegment transfer, we need provide both new CS and IP values in JMP instructions
For Example: JMP 2000H : 1000H

➤ Direct Jump v.s. Indirect Jump

- Direct Jump: the target address is directly given in the instruction
- Indirect Jump: the target address is contained in a register or memory location

➤ Short Jump

- If the target address is within +127 or -128 bytes of the current instruction address, the jump is called a short jump
- For short jumps, instead of specifying the target address, we can specify the relative offset (the distance between the current address and the target address) in JMP instructions.

Program Transfer Instructions

➤ Conditional Jumps

- JZ: *Label_1*

- If ZF =1, jump to the target address labeled by *Label_1*; otherwise, do not jump

- JNZ: *Label_1*

- If ZF =0, jump to the target address labeled by *Label_1*; otherwise, do not jump

➤ Other Conditional Jumps

JNC	JAЕ	JNB	JC	JB	JNAЕ	JNG
JNE	JE	JNS	JS	JNO	JO	JNP
JPO	JP	JPE	JA	JBNE	JBE	JNA
JGE	JNL	JL	JNGE	JG	JNLE	JLE

- JCXZ: *Label_1*

- If CX =0, jump to the target address labeled by *Label_1*; otherwise, do not jump

Program Transfer Instructions

❑ LOOP *Short_Label*

- It is limited for short jump
- Execution Flow:

```
CX = CX - 1  
If CX != 0 Then  
    JMP Short_Label  
End IF
```

❑ LOOPE/LOOPZ *Short_Label*

```
CX = CX - 1  
If CX != 0 & ZF=1 Then  
    JMP Short_Label  
End IF
```

❑ LOOPNE/LOOPNZ *Short_Label*

```
CX = CX - 1  
If CX != 0 & ZF=0 Then  
    JMP Short_Label  
End IF
```

Processor Control Instructions

<input type="checkbox"/> CLC	<i>Clear carry flag</i>
<input type="checkbox"/> STC	<i>Set carry flag</i>
<input type="checkbox"/> CMC	<i>Complement carry flag</i>
<input type="checkbox"/> CLD	<i>Clear direction flag</i>
<input type="checkbox"/> STD	<i>Set direction flag</i>
<input type="checkbox"/> CLI	<i>Clear interrupt-enable flag</i>
<input type="checkbox"/> STI	<i>Set interrupt-enable flag</i>
<input type="checkbox"/> HLT	<i>Halt microprocessor operation</i>
<input type="checkbox"/> NOP	<i>No operation</i>
<input type="checkbox"/> LOCK	<i>Lock Bus During Next Instruction</i>

Subroutine Instructions

- A subroutine is a collection of instructions that can be called from one or more other locations within a program

❑ *CALL Procedure-Name*

— Example

...

MOV AL, 1

CALL M1

MOV BL, 3

...

M PROC

MOV CL, 2

RET

M ENDP

The order of execution:

MOV AL, 1

MOV CL, 2

MOV BL, 3

- Intersegment CALL: the subroutine is located in a different code segment
- Intrasegment CALL: the subroutine is located in the same code segment
- Use assembler directives **far** and **near** to distinguish intersegment and intrasegment CALL

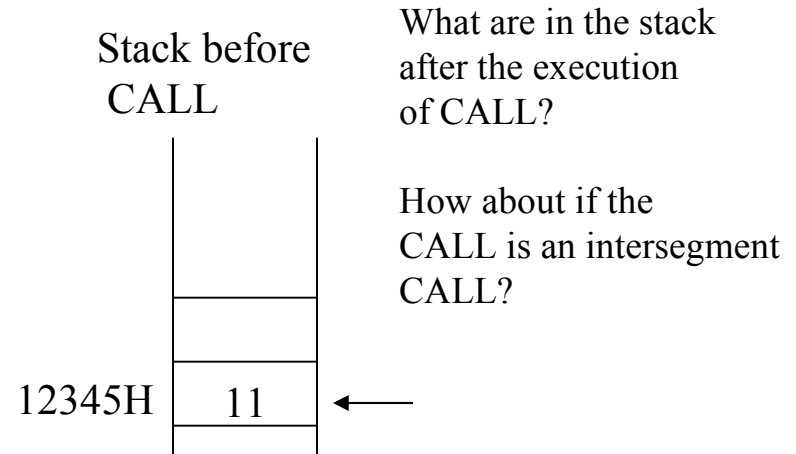
Subroutine Instructions

- What does the microprocessor do when it encounters a CALL instruction?
1. Push the values of CS and IP (which specify the address of the instruction immediately following the CALL instruction) into stack. If it is an intrasegment CALL, just push the value of IP into stack.
 2. Load the new values to CS and IP such that the next instruction that the microprocessor will fetch is the first instruction of the subroutine

— Example:

```
0000          .model small
0000          .code
0000 B0 02     MOV AL, 2
0002 E8 0002   CALL m1
0005 B3 03     MOV BL, 3

0007          m1 Proc
0007 B7 05     MOV BH, 5
0009 C3       RET
000A          m1 ENDP
              end
```



Subroutine Instructions

❑ RET

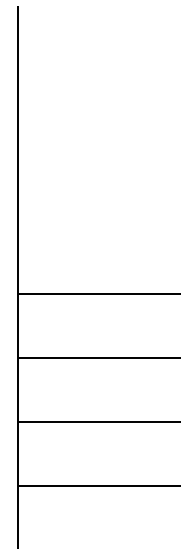
- It lets the microprocessor exit from a subroutine
- If it is used in a FAR procedure, RET pops two words from the stack. The first one goes to IP register. The second one goes to CS register
- If it is used in a NEAR procedure, RET pops one word from stack to IP register

— Example:

```
1234:2345    ...  
1234:2348    CALL FAR PTR M1  
1234:234D    ...
```

```
        M1 PROC FAR  
3456:0120    MOV AL, 0  
        ...  
        RET  
M1 ENDP
```

01022



What data are pushed into and popped from the stack during the execution of CALL and RET?

Interrupt Instructions

□ INT *Interrupt-Type*

- This instruction causes the microprocessor to execute an interrupt service routine. The *Interrupt-Type* is an immediate data (0-255) which specifies the type of interrupt
- It results in the following operations:

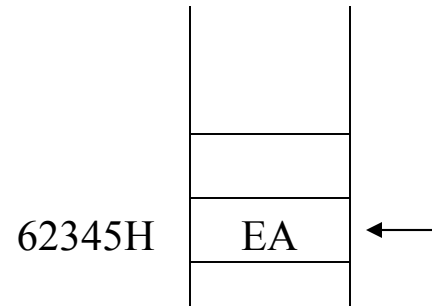
1. Push flag register into stack
2. Clear trace flag and interrupt-enable flag
3. Push CS and IP into stack
4. Load new CS and IP values from the interrupt vector table

— Example:

...

1230:6789 INT 20H

...

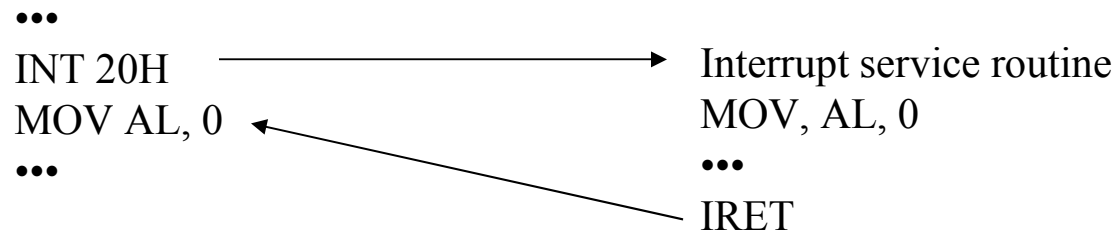


After the execution of INT 20H, what are the data pushed into the stack?

Interrupt Instructions

❑ IRET

- It is used at the end of an interrupt service routine to make the microprocessor jump back to the instruction that immediately follows the INT instruction



- It results in the following operations

1. Restore the original CS and IP values by popping them from stack
2. Restore the original flag register value by popping it from stack

Hardware and Software Interrupts

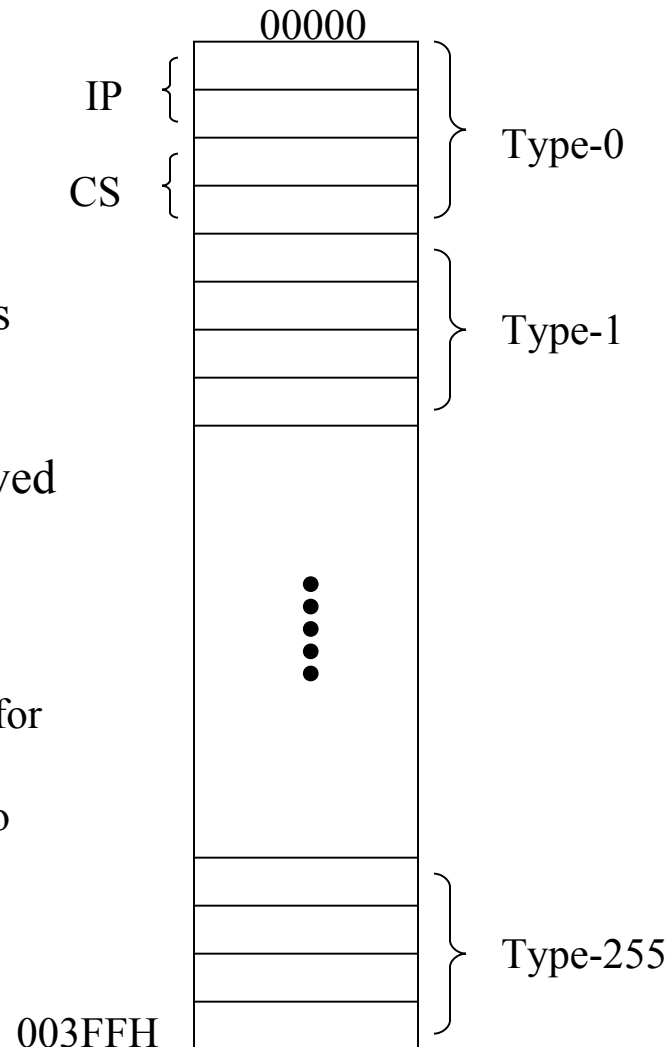
- ❑ An interrupt is an event that causes the processor to stop its current program execution and switch to performing an interrupt service routine.
- ❑ Hardware and Software Interrupts
 - Hardware Interrupts are caused by proper inputs at NMI or INTR input pin
 - Software Interrupts are caused by executing programs
- ❑ Interrupt Priority
 - When multiple interrupts occur at the same time, the interrupt with the highest priority will be served
- ❑ Interrupt Type
 - Interrupt type is used as the table index to search the address of interrupt service routine from the interrupt vector table

Interrupt Vector Table

- ❑ Interrupt vector table is used to store the addresses of interrupt service routine
- ❑ Interrupt vector table contains 256 table entries. Each table entry takes 4 bytes; two bytes are for IP values and two bytes are for CS values
- ❑ Interrupt vector table locates at the reserved memory space from 00000H to 003FFH

— Example:

Assume that the interrupt service routine for the type-40 interrupt is located at address 28000H. How do you write this address to the vector table?



Interrupt Processing Sequence

1. Get Vector Number (get the interrupt type)
 - Caused by NMI, it is type 2
 - Caused by INTR, the type number will be fed to the processor through data bus
 - Caused by executing INT instructions, the type number is given by the operand
 - ...
2. Save Processor Information
 1. Push flag register into stack
 2. Clear trace flag and interrupt-enable flag
 3. Push CS and IP into stack
3. Fetch New Instruction Pointer
 - Load new CS and IP values from the instruction vector table
4. Execute interrupt service routine
5. Return from interrupt service routine
 1. Pop flag register from stack
 2. Pop CS and IP from stack

Interrupt Service Routine

- ❑ An Interrupt Service Routine (ISR) is a section code that take care of processing a specific interrupt
- ❑ Some ISRs also contain instructions that save and restore restore general purpose registers

— Example:

```
1234:00AE PUSH AX
          PUSH DX

          MOV AX, 5
          MUL BL
          MOV [DI], AX
          MOV [DI+2], DX

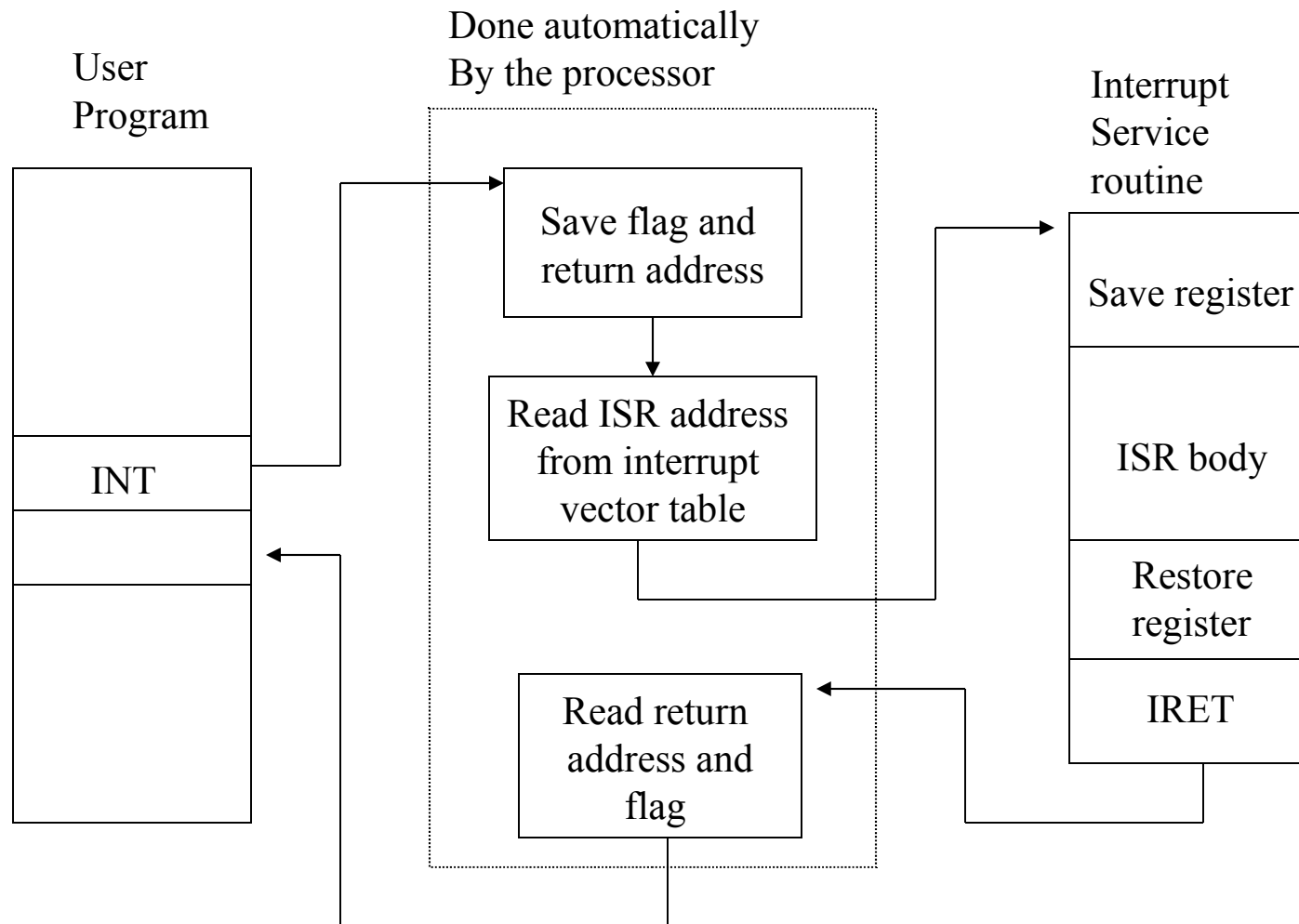
          POP DX
          POP AX

          IRET
```

Interrupt Vector Table

000A4	AE	} INT ?
000A5	00	
000A6	34	
000A7	12	

Storing Environment During Interrupt Processing



Special Interrupts

❑ Divide-Error

— Type-0 interrupt. It has the highest interrupt priority

❑ Single-Step

— Type-1 interrupt. It is generated after each instruction if the trace flag is set

❑ NMI

— Type-2 interrupt

❑ Breakpoint

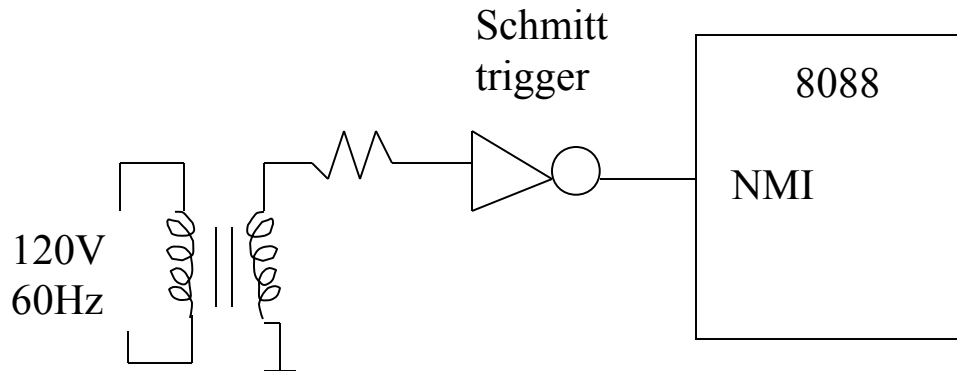
— Type-3 interrupt. It is used for debug purposes

❑ Overflow

— Type-4 interrupt. It is generated by INTO when the overflow flag is set

Interrupt Example

□ An NMI Time Clock



— Instructions for update
Interrupt Vector Table

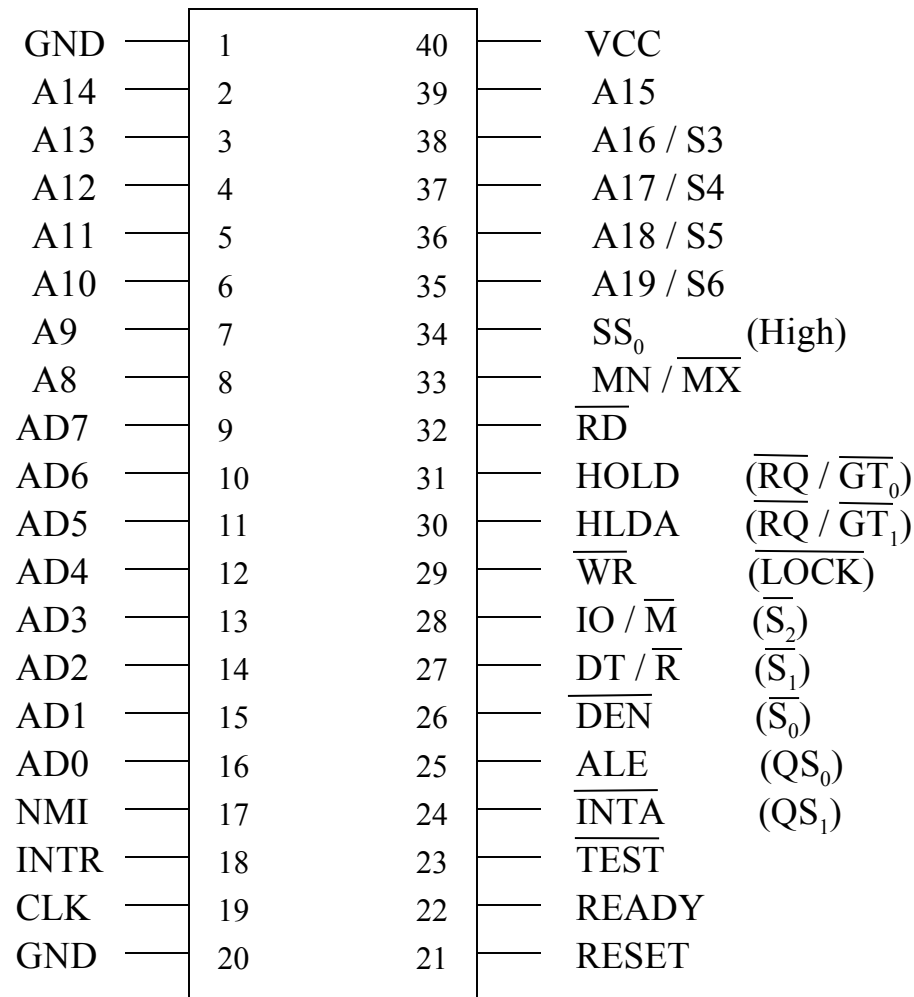
— ISR

```
NMITIME: DEC COUNT
          JNZ EXIT
          MOV COUNT, 60
          CALL FAR PTR ONESEC
EXIT:     IRET
```

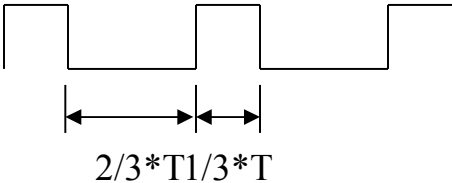
```
MOV COUNT, 60
PUSH DS
SUB AX, AX
MOV DS, AX
LEA AX, NMITIME
MOV [8], AX
MOV AX, CS
MOV [0AH], AX
POP DS
```

Hardware Interface

8088 Pin Configuration

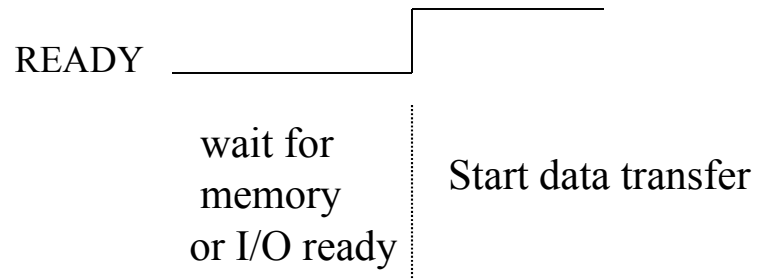
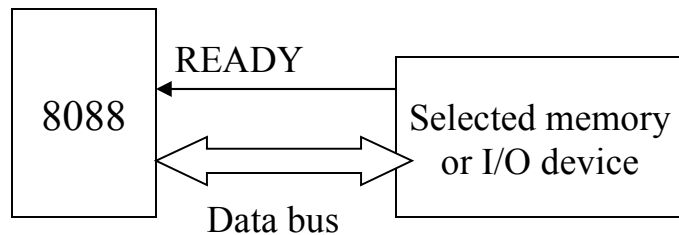


8088 Pin Description

Pin Name	Pin Number	Direction	Description
GND:	1 & 20		Both need to be connected to ground
VCC:	21		VCC = 5V
CLK:	19	Input	33% duty cycle 
MN/ $\overline{\text{MX}}$:	33	Input	High → Minimum mode Low → Maximum mode
RESET:	21	Input	Reset 8088 <ul style="list-style-type: none">➤ Duration of logic high must be greater than $4*T$➤ After reset, 8088 fetches instructions starting from memory address FFFF0H

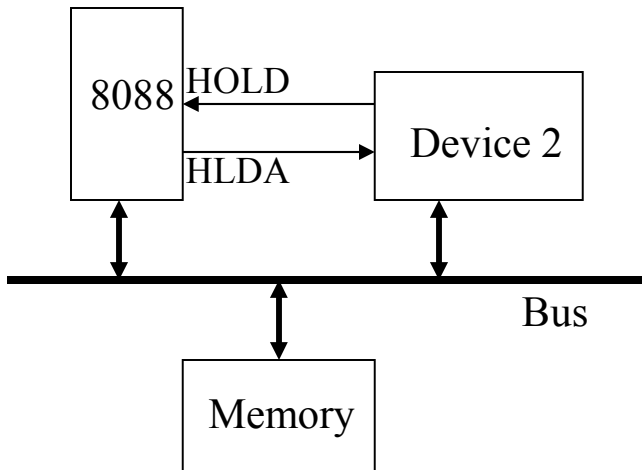
8088 Pin Description

Pin Name	Pin Number	Direction	Description
READY	22	Input	Informs the processor that the selected memory or I/O device is ready for a data transfer



8088 Pin Description

Pin Name	Pin Number	Direction	Description
HOLD	31	Input	The execution of the processor is suspended as long as HOLD is high
HLDA	30	Output	Acknowledges that the processor is suspended

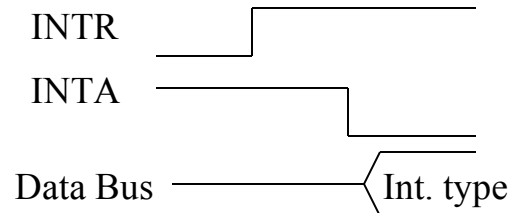
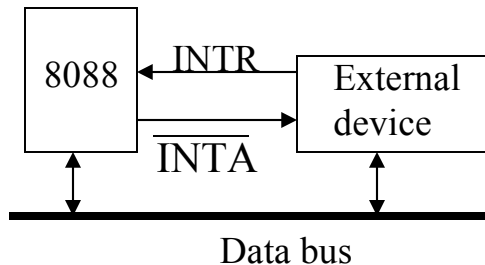


- Procedure for Device 2 to use bus
 - Drive the HOLD signal of 8088 high
 - Wait for the HLDA signal of 8088 becoming high
 - Now, Device2 can send data to bus

8088 Pin Description

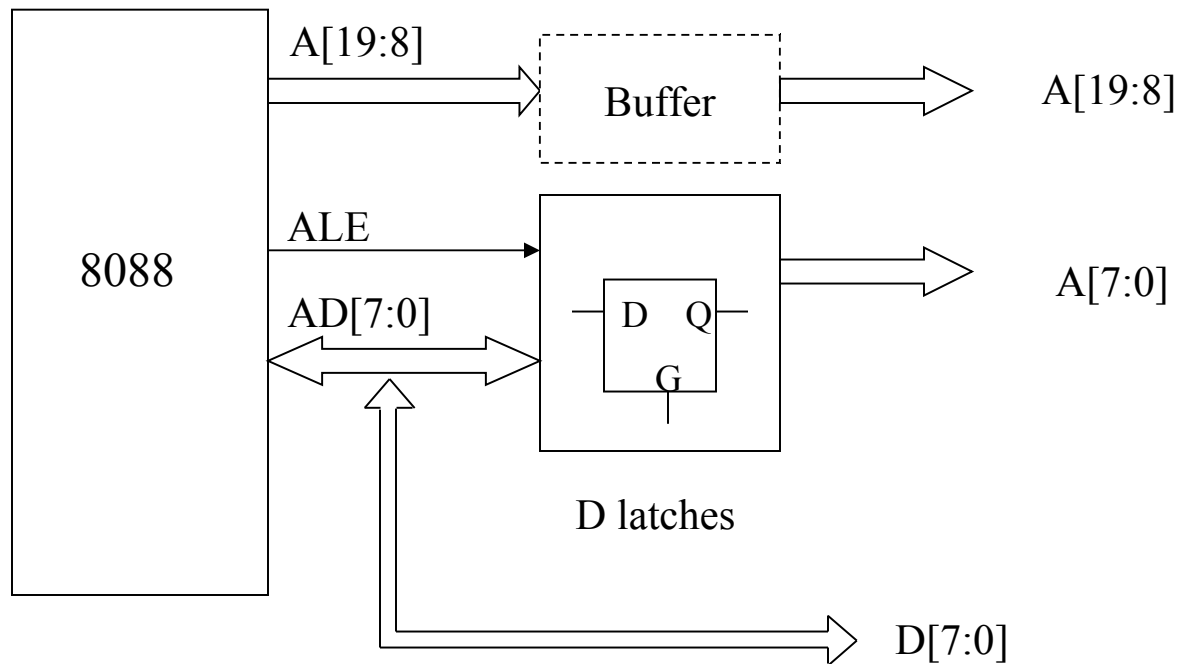
Pin Name	Pin Number	Direction	Description
NMI	17	Input	Causes a non-maskable type-2 interrupt
INTR	18	Input	Indicates a maskable interrupt request
$\overline{\text{INTA}}$	24	Output	Indicates that the processor has received an INTR request and is beginning interrupt processing

- **NMI (non-maskable interrupt):** a rising edge on NMI causes a type-2 interrupt
- **INTR:** logic high on INTR poses an interrupt request. However, this request can be masked by IF (**Interrupt enable Flag**). The type of interrupt caused by INTR is read from data bus
- **$\overline{\text{INTA}}$:** control when the interrupt type should be loaded onto the data bus



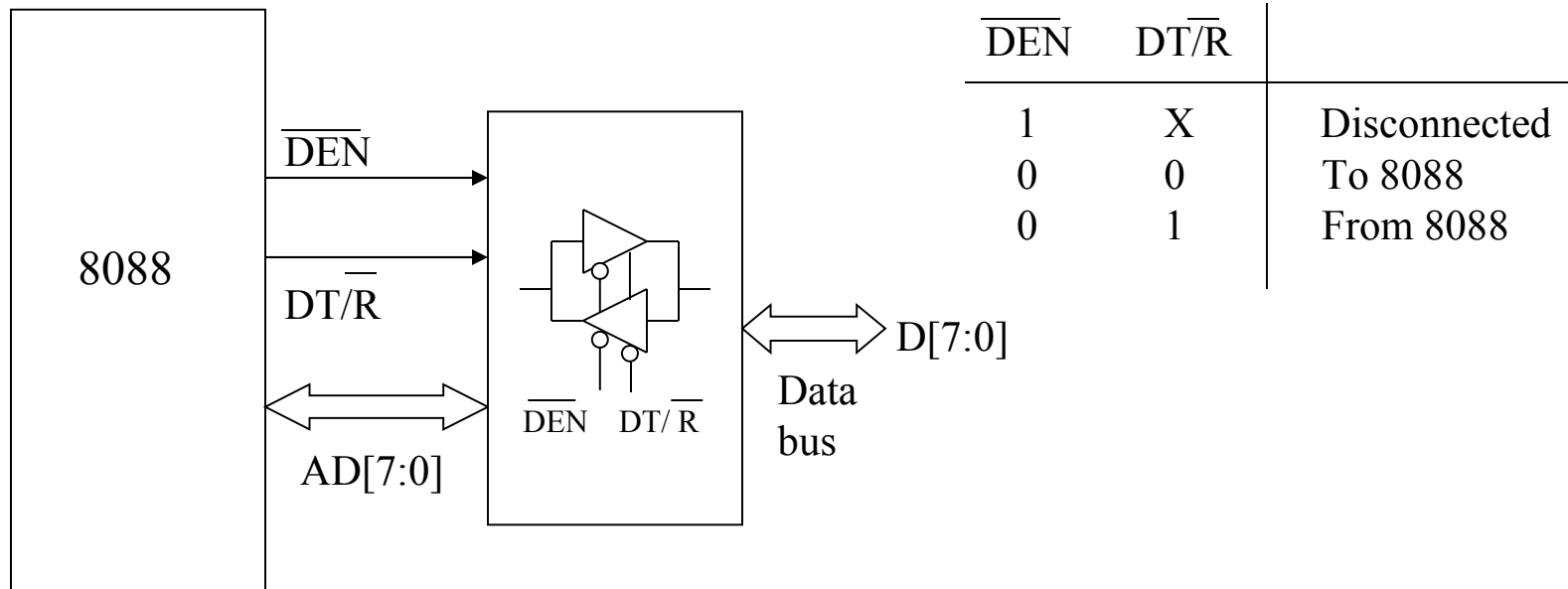
8088 Pin Description

Pin Name	Pin Number	Direction	Description
ALE	25	Output	Indicates the current data on 8088 address/data bus are address



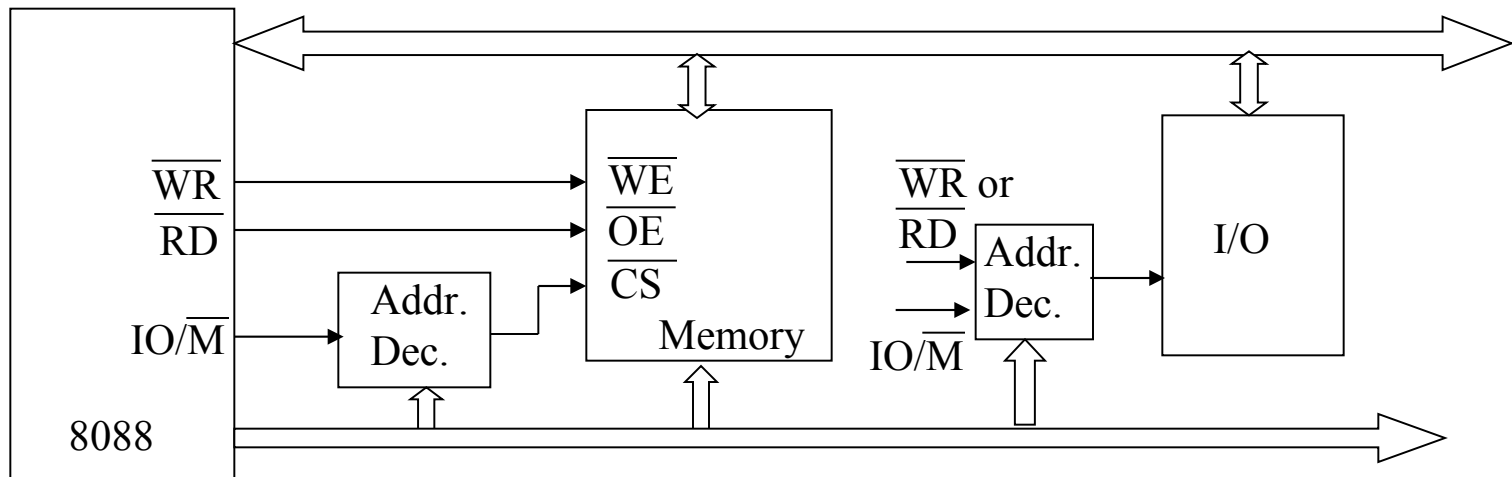
8088 Pin Description

Pin Name	Pin Number	Direction	Description
$\overline{\text{DEN}}$	26	Output	Disconnects data bus connection
$\text{DT} / \overline{\text{R}}$	27	Output	Indicates the direction of data transfer



8088 Pin Description

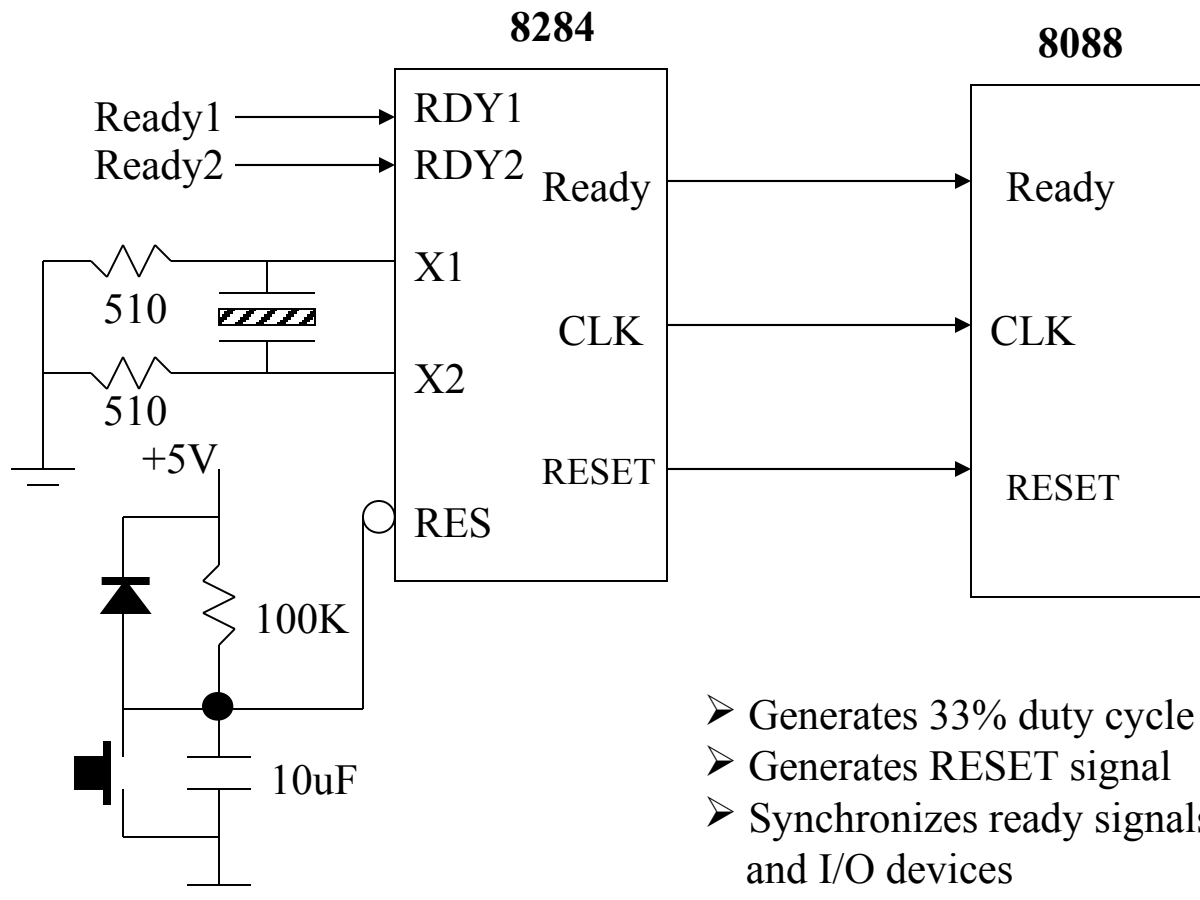
Pin Name	Pin Number	Direction	Description
$\overline{\text{WR}}$	29	Output	Indicates that the processor is writing to memory or I/O devices
$\overline{\text{RD}}$	32	Output	Indicates that the processor is reading from memory or I/O devices
$\text{IO}/\overline{\text{M}}$	28	Output	Indicates that the processor is accessing whether memory ($\text{IO}/\overline{\text{M}}=0$) or I/O devices ($\text{IO}/\overline{\text{M}}=1$)



8088 Pin Description

Pin Name	Pin Number	Direction	Description
AD[7:0]	9-16	I/O	Address / Data bus
A[19:8]	2-8, 35-39	Input	Address bus
LOCK	29	Input	Lock output is used to lock peripherals off the system. Activated by using the LOCK: prefix on any instruction.

8284 Clock Generator



Basic functions:

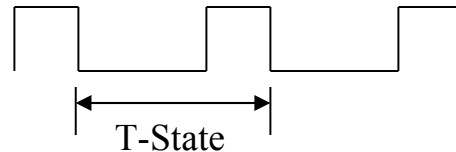
- ✓ Clock generation.
- ✓ RESET synchronization.
- ✓ READY synchronization.
- ✓ Peripheral clock signal.

- Generates 33% duty cycle clock signal
- Generates RESET signal
- Synchronizes ready signals from memory and I/O devices

System Timing Diagrams

□ T-State:

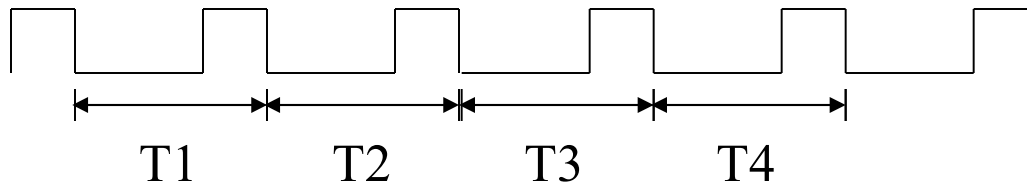
- One clock period is referred to as a T-State



- An operation takes an integer number of T-States

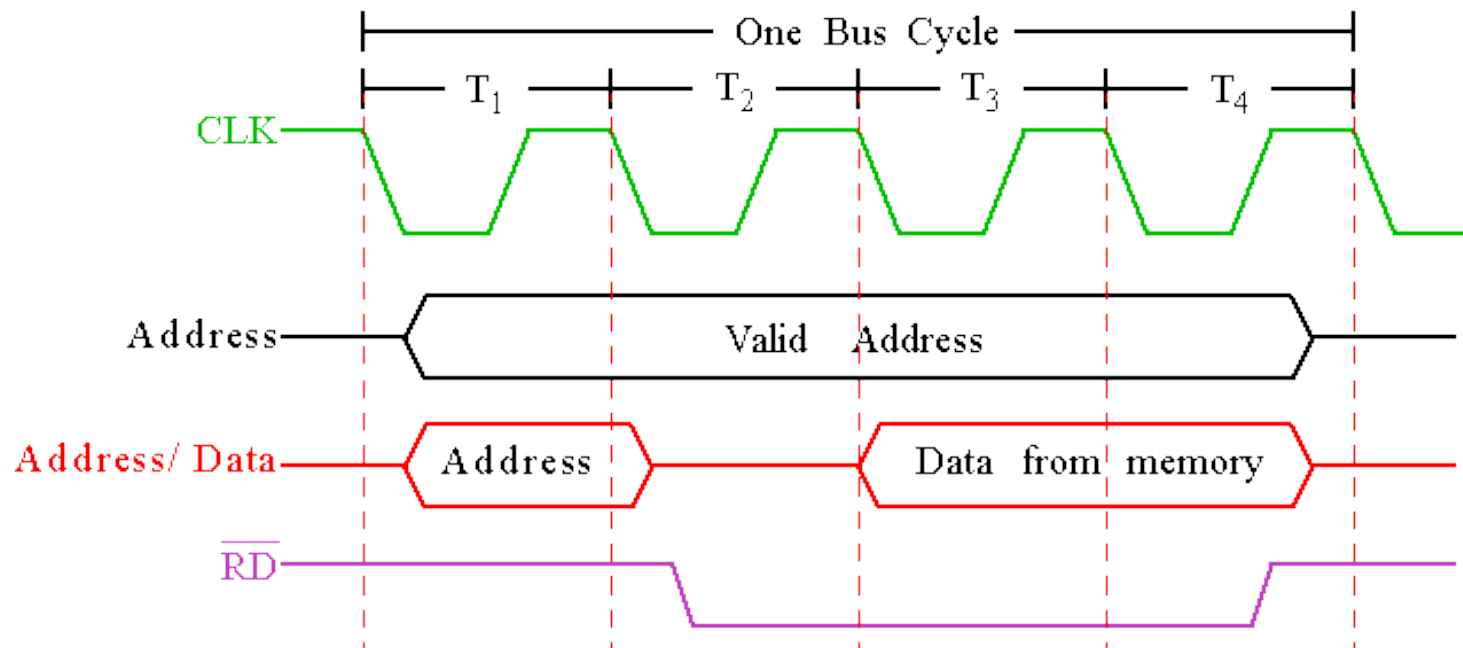
□ CPU Bus Cycle:

- A bus cycle consists of 4 or more T-States



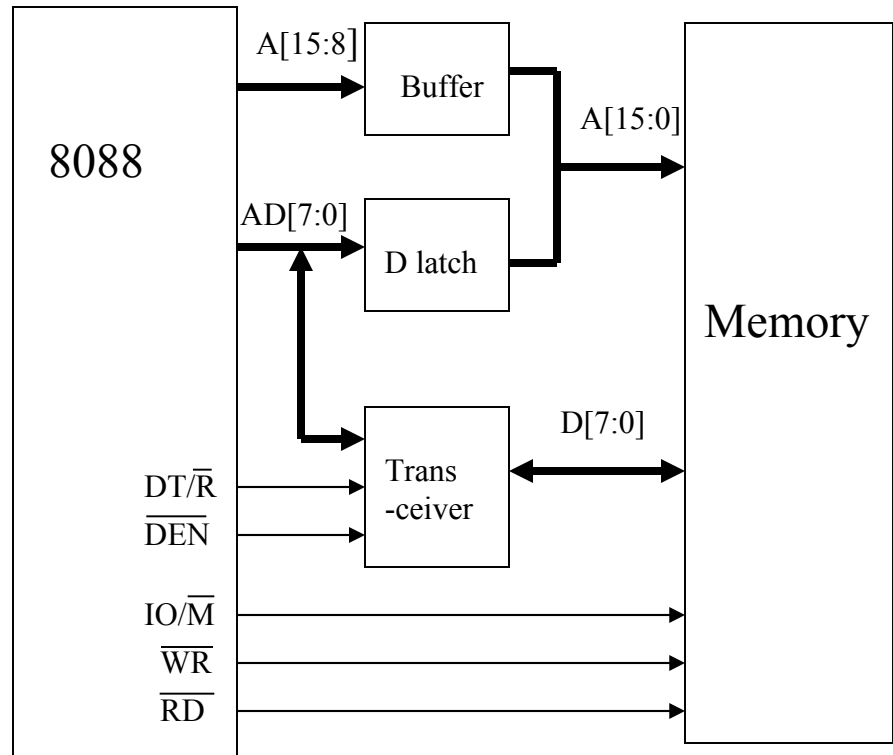
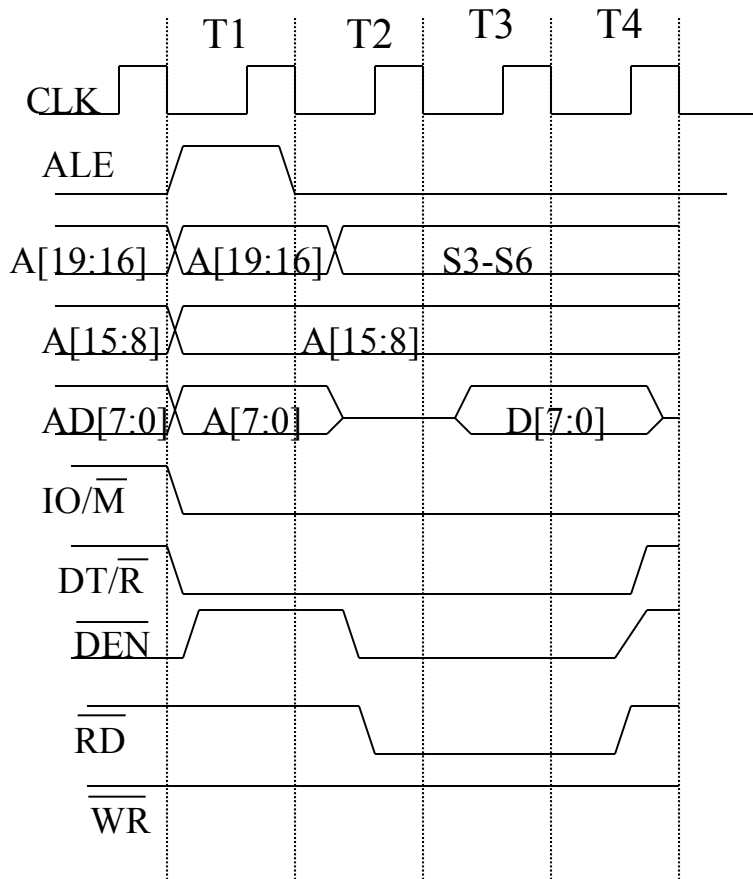
Memory Read Timing Diagrams

- *Dump address on address bus.*
- *Issue a read (RD) and set M/IO to 1.*
- *Wait for memory access cycle.*



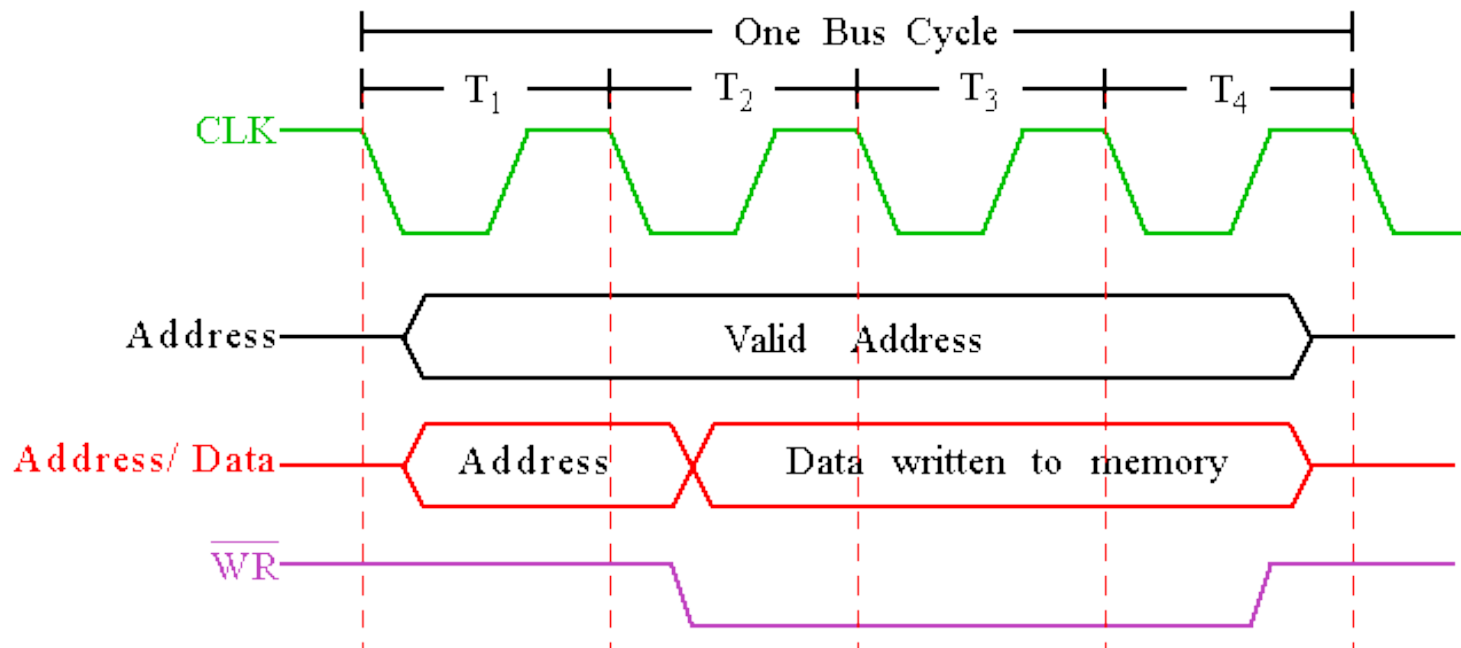
Simplified 8086 Read Bus Cycle

Memory Read Timing Diagrams



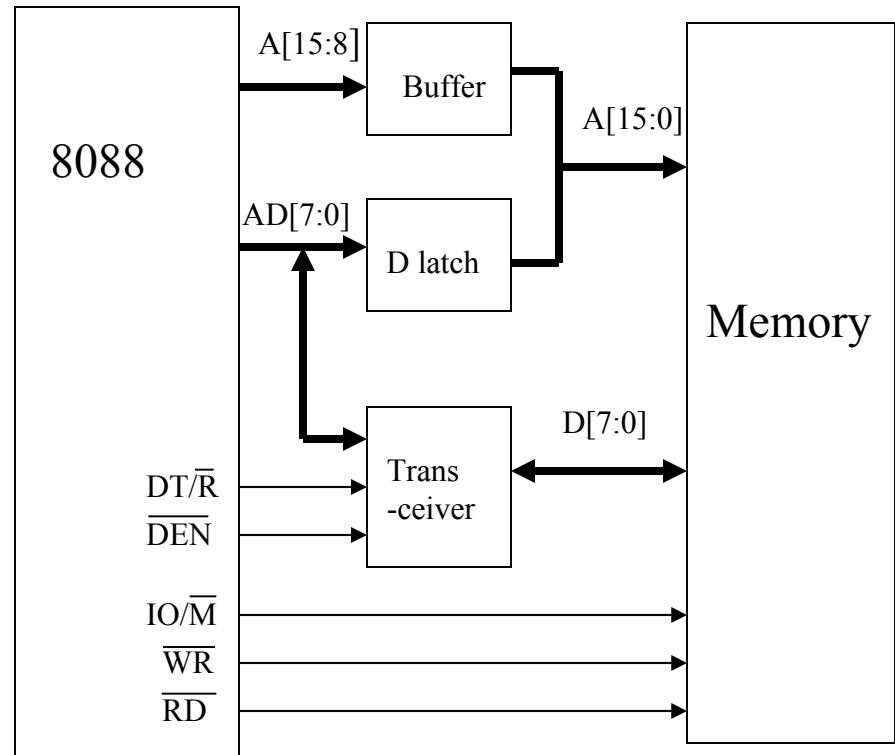
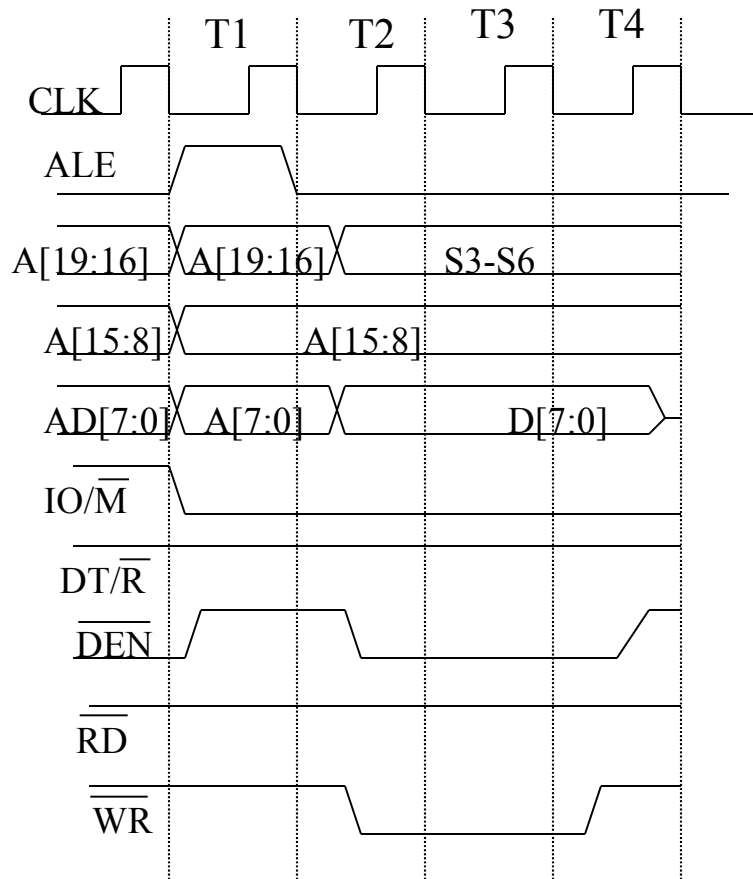
Memory Write Timing Diagrams

- *Dump address on address bus.*
- *Dump data on data bus.*
- *Issue a write (WR) and set M/ IO to 1.*



Simplified 8086 Write Bus Cycle

Memory Write Timing Diagrams



Bus Timing

During T 1 :

- *The address is placed on the Address/Data bus.*
- *Control signals M/ IO , ALE and DT/ R specify memory or I/O, latch the address onto the address bus and set the direction of data transfer on data bus.*

During T 2 :

- *8086 issues the RD or WR signal, DEN , and, for a write, the data.*
 - *DEN enables the memory or I/O device to receive the data for writes and the 8086 to receive the data for reads.*

During T 3 :

- *This cycle is provided to allow memory to access data.*
- *READY is sampled at the end of T 2 .*
 - *If low, T 3 becomes a wait state.*
 - *Otherwise, the data bus is sampled at the end of T 3 .*

During T 4 :

- *All bus signals are deactivated, in preparation for next bus cycle.*
- *Data is sampled for reads, writes occur for writes.*

Bus Timing

Timing:

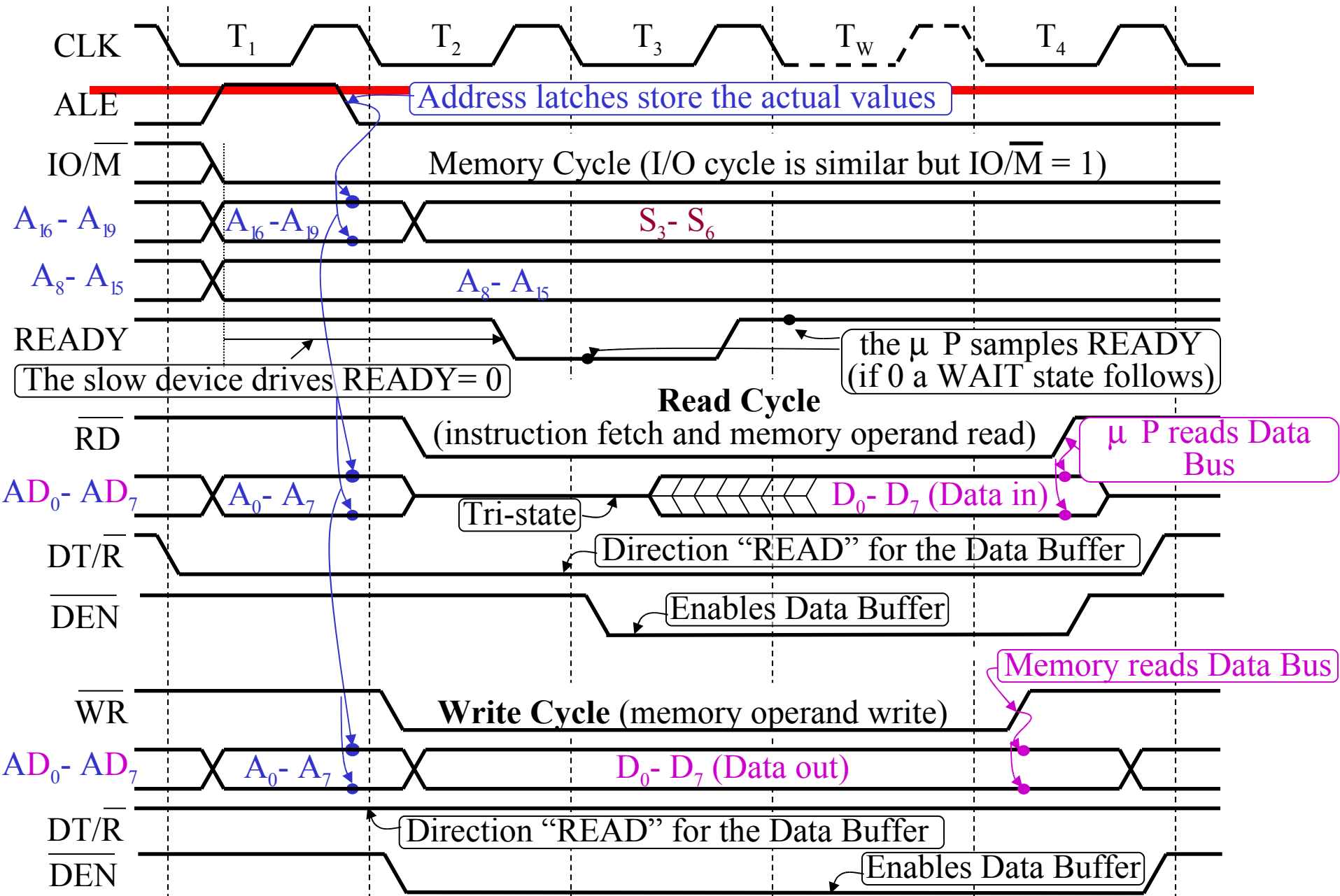
- Each BUS CYCLE on the 8086 equals **four** system clocking periods (*T* states).
- The clock rate is **5MHz** , therefore one Bus Cycle is 800ns .
- The transfer rate is **1.25MHz** .

Memory specs (memory access time) must match constraints of system timing.

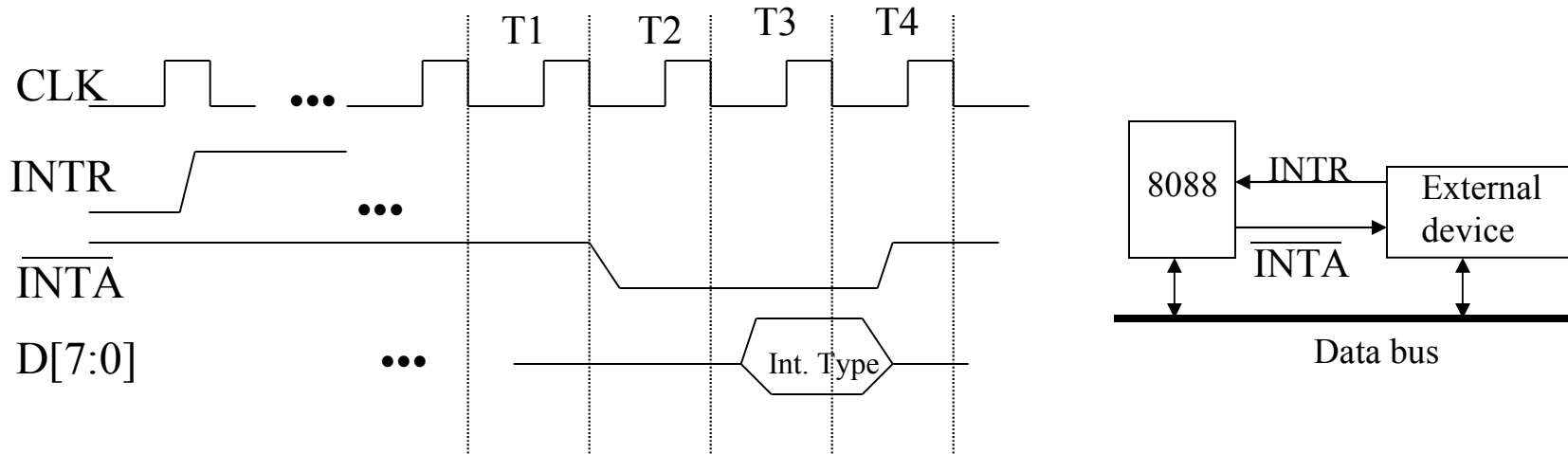
For example, bus timing for a read operation shows almost 600ns are needed to read data.

- *However, memory must access faster due to setup times, e.g. Address setup and data setup.*
- *This subtracts off about 150ns .*
- *Therefore, memory must access in at least 450ns minus another 30-40ns guard band for buffers and decoders.*
- *420ns DRAM required for the 8086.*

10.6 System Time Diagrams - CPU Bus Cycle

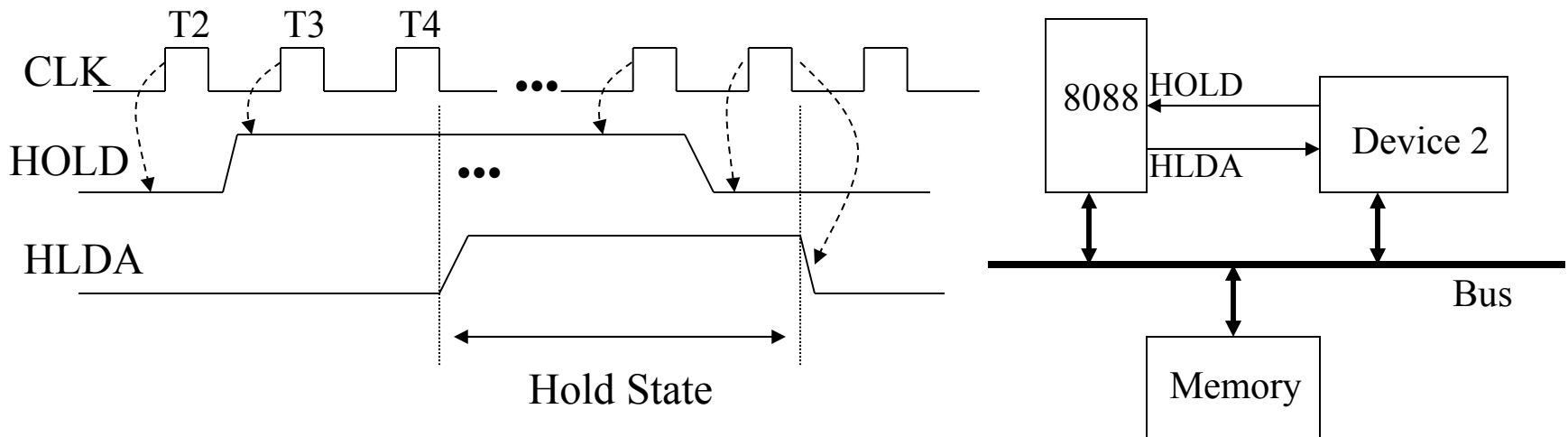


Interrupt Acknowledge Timing Diagrams



- ❑ It takes one bus cycle to perform an interrupt acknowledge
- ❑ During T1, the process tri-states the address bus
- ❑ During T2, \overline{INTA} is pulled low and remains low until it becomes inactive in T4
- ❑ The interrupting devices places an 8-bit interrupt type during \overline{INTA} is active

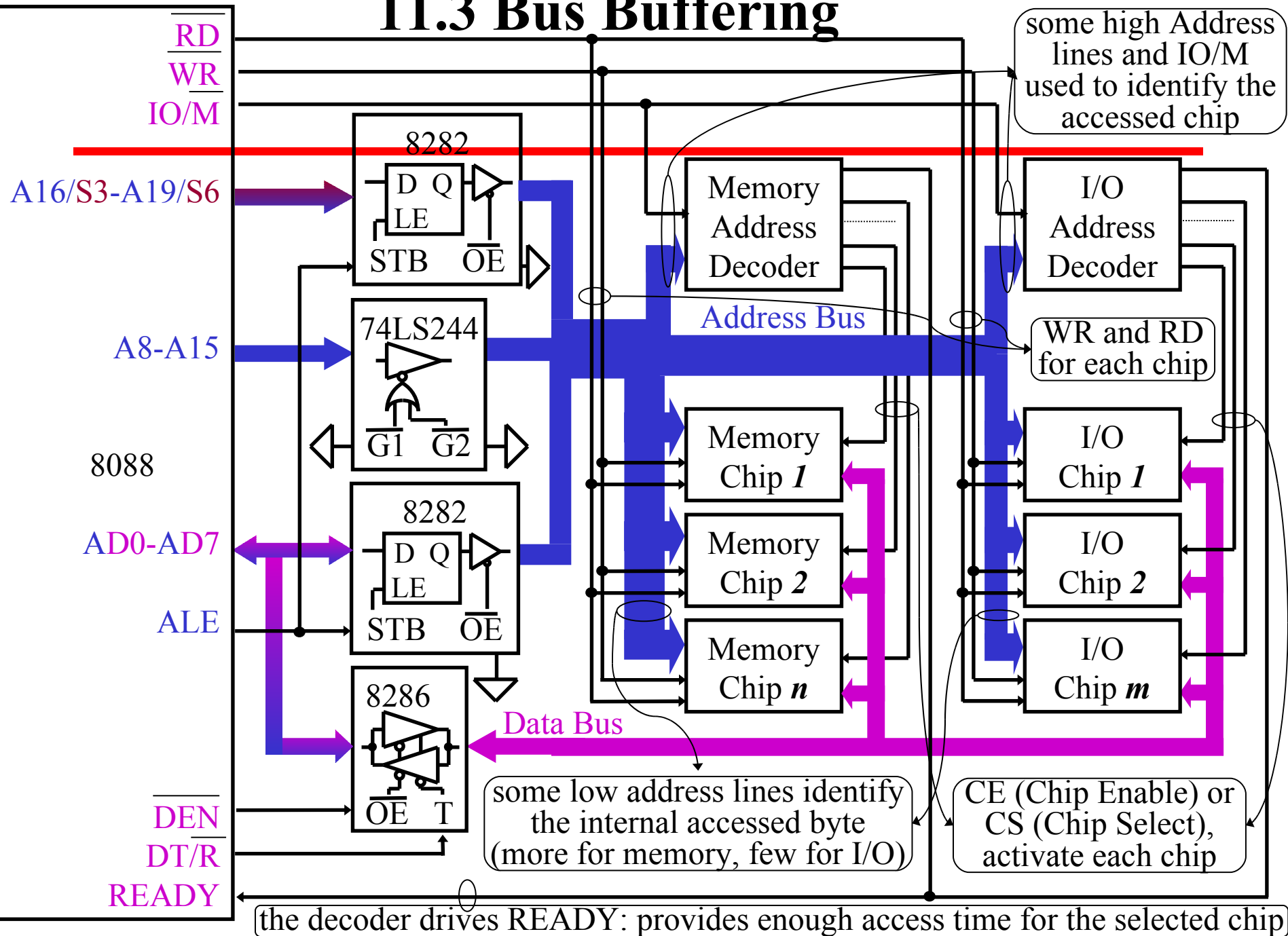
HOLD/HLDA Timing Diagrams



- ❑ The processor will examine HOLD signal at every rising clock edge
- ❑ If HOLD=1, the processor will pull HLDA high at the end of T4 state (end of the execution of the current instruction) and suspend its normal operation
- ❑ If HOLD=0, the processor will pull down HLDA at the falling clock edge and resume its normal operation

Memory Interface

11.3 Bus Buffering



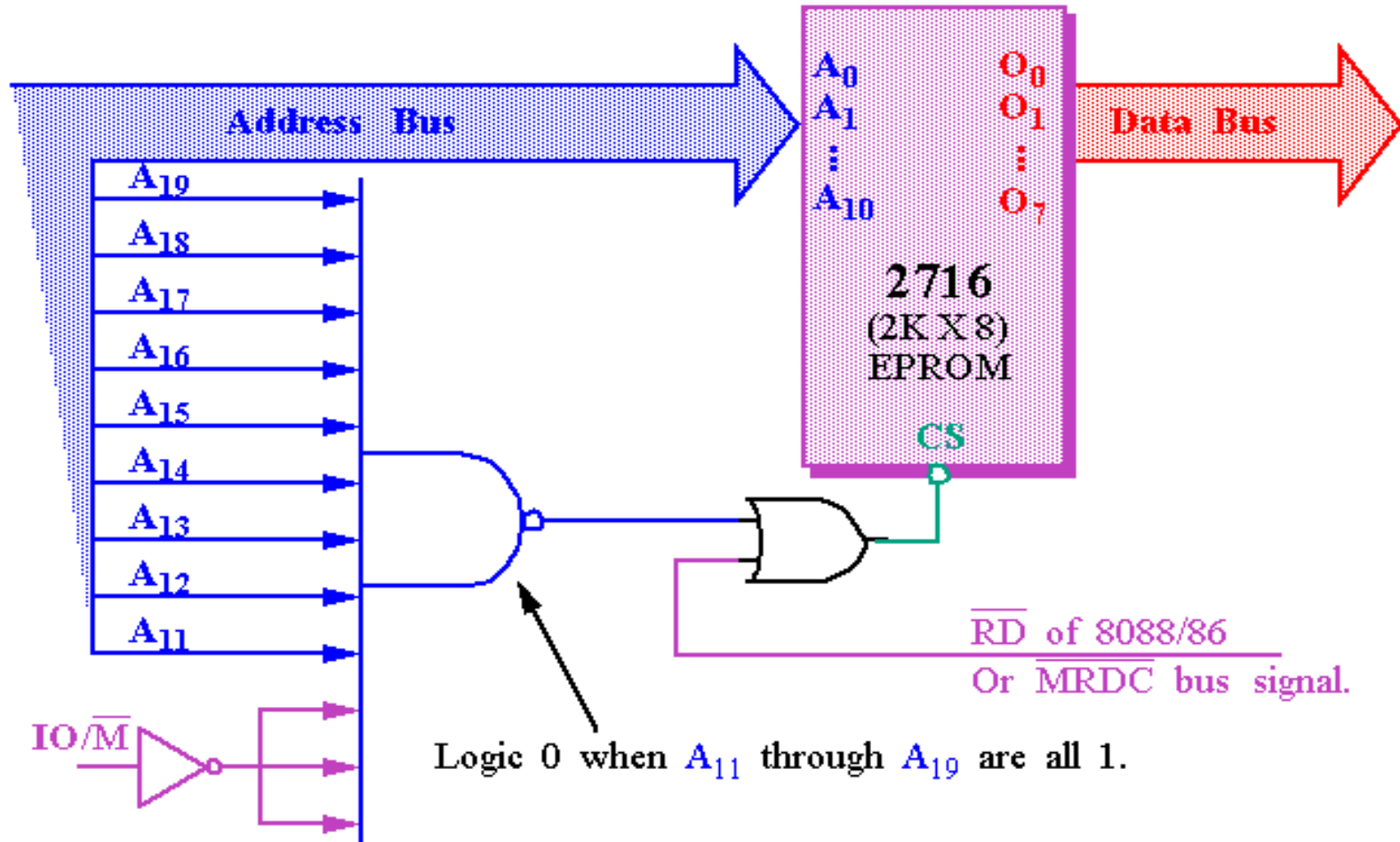
Memory Chips

- *The number of address pins is related to the number of memory locations .*
 - *Common sizes today are **1K** to **256M** locations. (10 and 28 address pins are present.)*
- *The data pins are typically bi-directional in read-write memories.*
 - *The number of data pins is related to the size of the memory location .*
 - *For example, an 8-bit wide (byte-wide) memory device has **8** data pins.*
 - *Catalog listing of 1K X 8 indicate a byte addressable 8K memory.*
- *Each memory device has at least one chip select (CS) or chip enable (CE) or select (S) pin that enables the memory device.*
- *Each memory device has at least one control pin.*
 - *For ROMs, an output enable (OE) or gate (G) is present.*
 - *The OE pin enables and disables a set of tristate buffers.*
 - *For RAMs, a read-write (R/W) or write enable (WE) and read enable (OE) are present.*
 - *For dual control pin devices, it must be hold true that both are not 0 at the same time.*

Memory Address Decoding

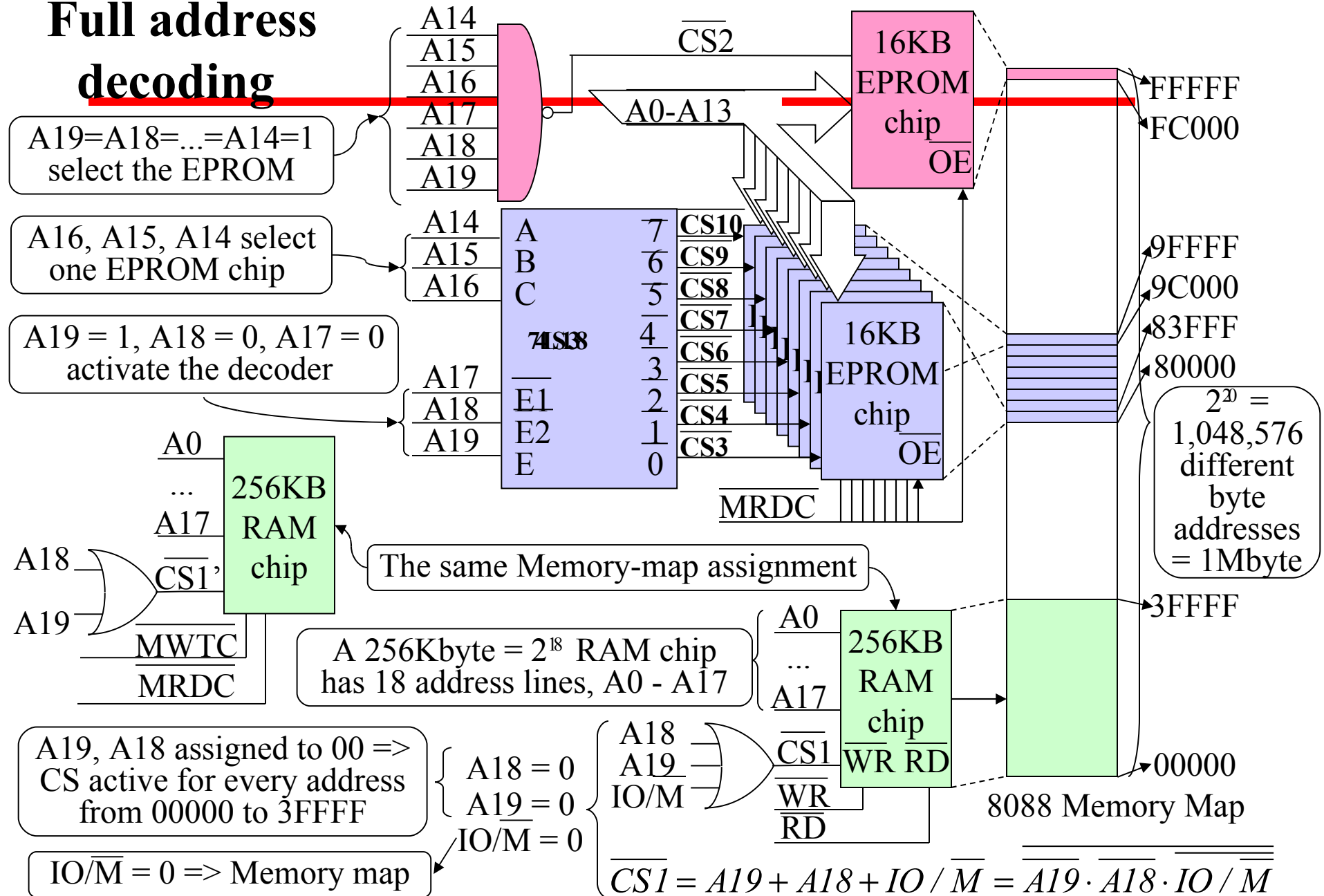
- *The processor can usually address a memory space that is much larger than the memory space covered by an individual memory chip.*
- *In order to splice a memory device into the address space of the processor, decoding is necessary.*
- *For example, the 8088 issues 20-bit addresses for a total of **1MB** of memory address space.*
- *However, the BIOS on a 2716 EPROM has only 2KB of memory and 11 address pins.*
- *A decoder can be used to decode the additional 9 address pins and allow the EPROM to be placed in **any** 2KB section of the 1MB address space.*

Memory Address Decoding

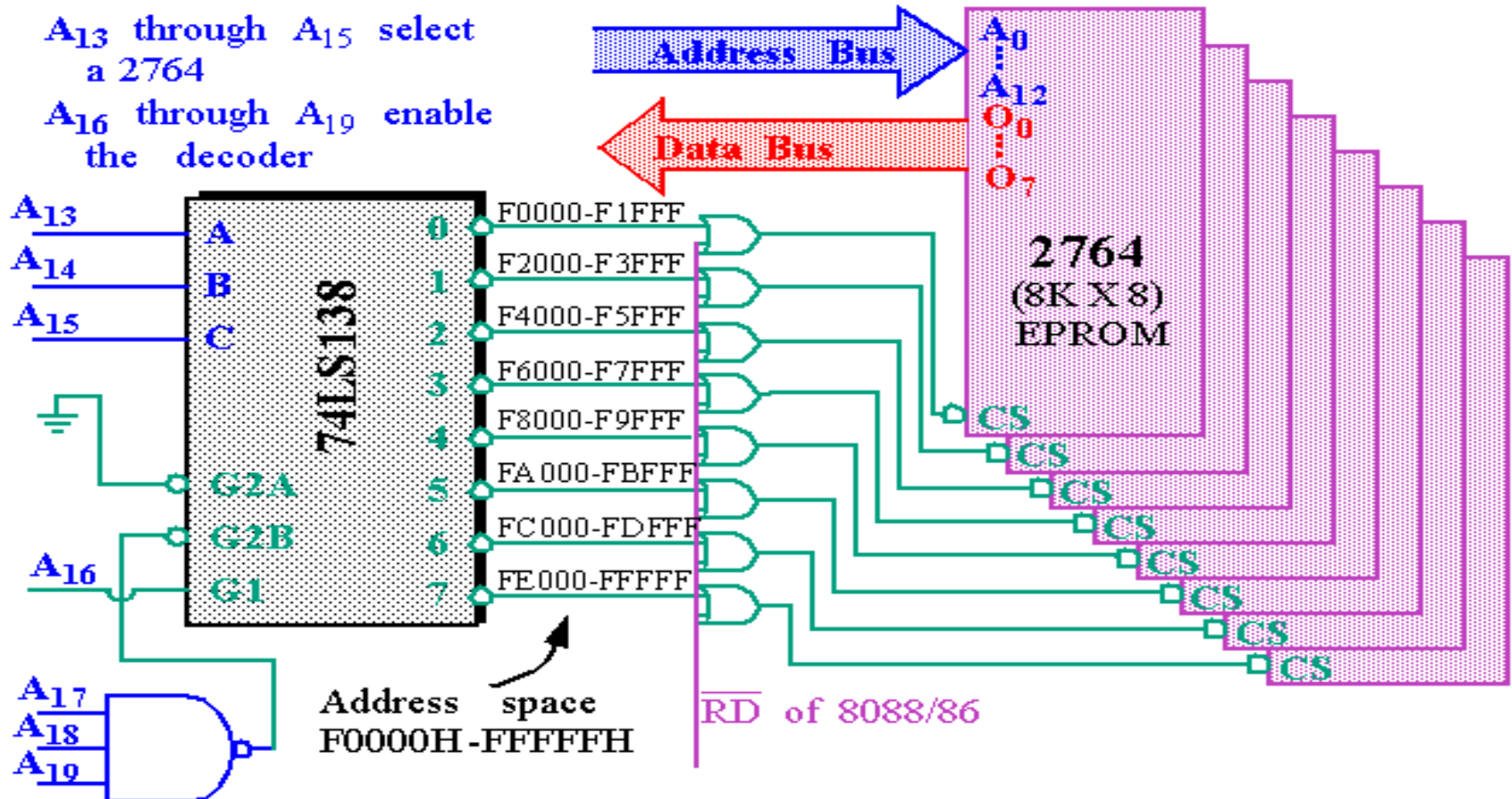


Memory Map - Full address decoding

All the address lines used by the decoder or memory chip => each byte is uniquely addressed = **full address decoding**

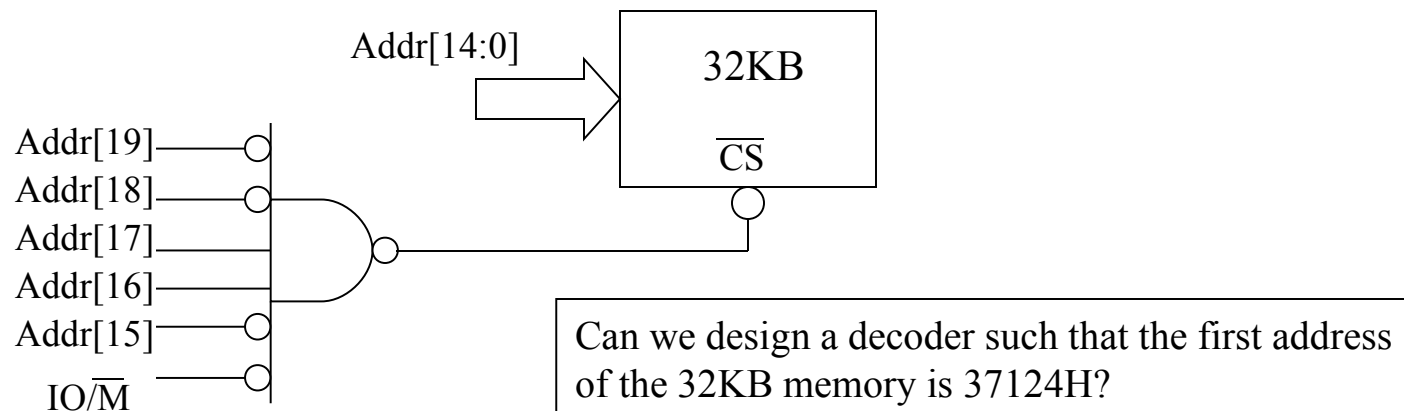
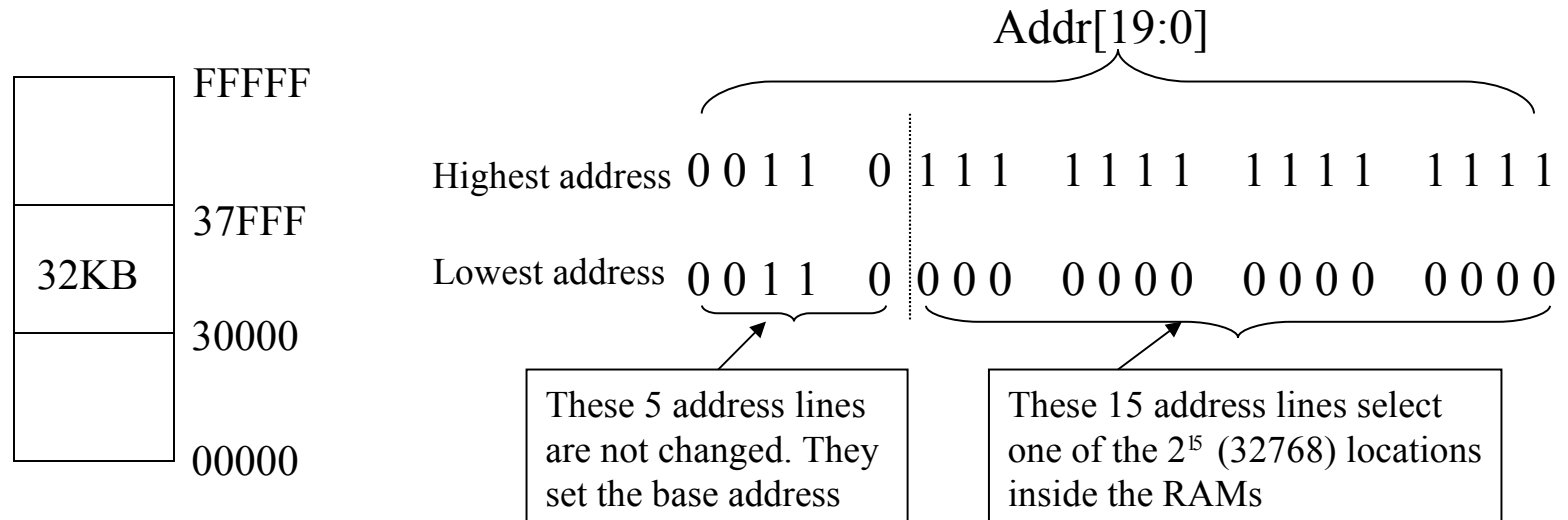


Decoding Circuits



Memory Address Decoding

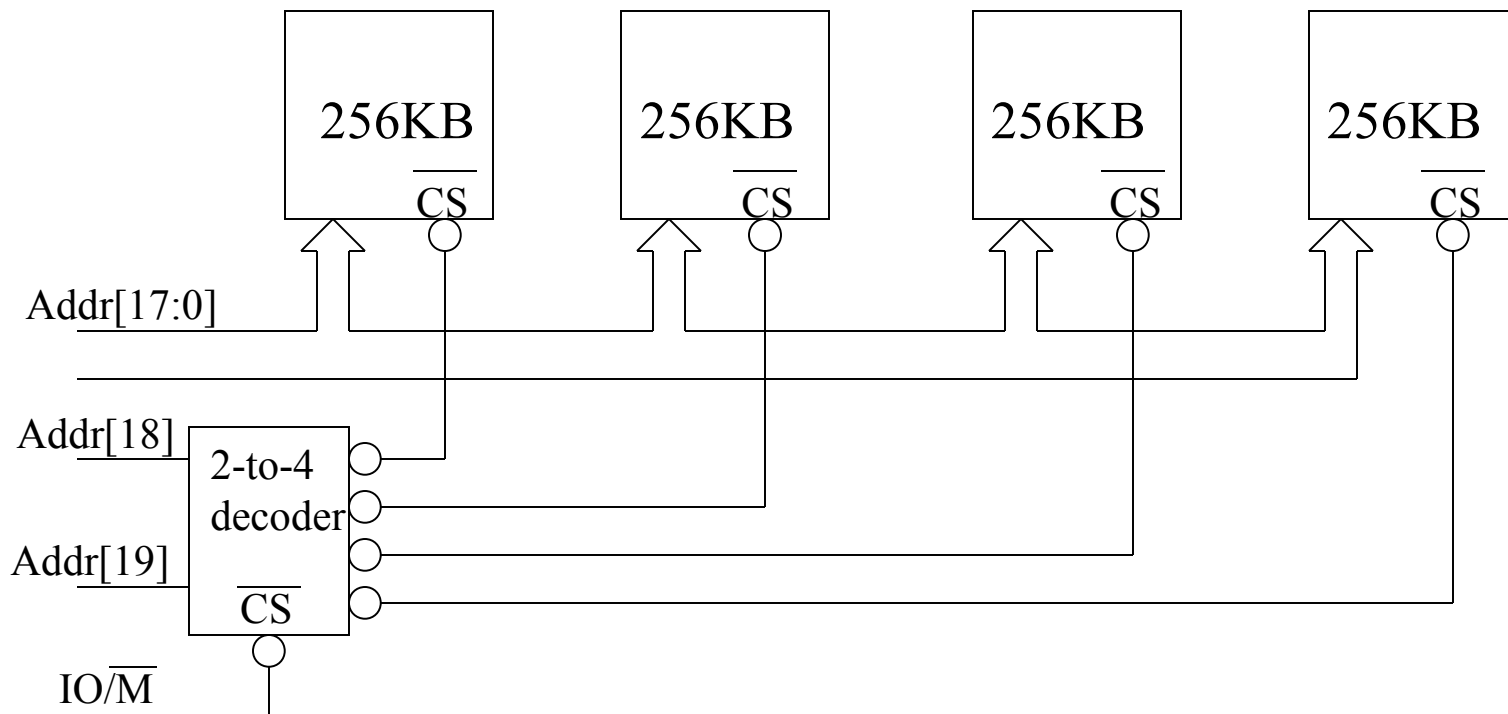
❑ Using Full memory addressing space



Memory Address Decoding

- Design a 1MB memory system consisting of multiple memory chips

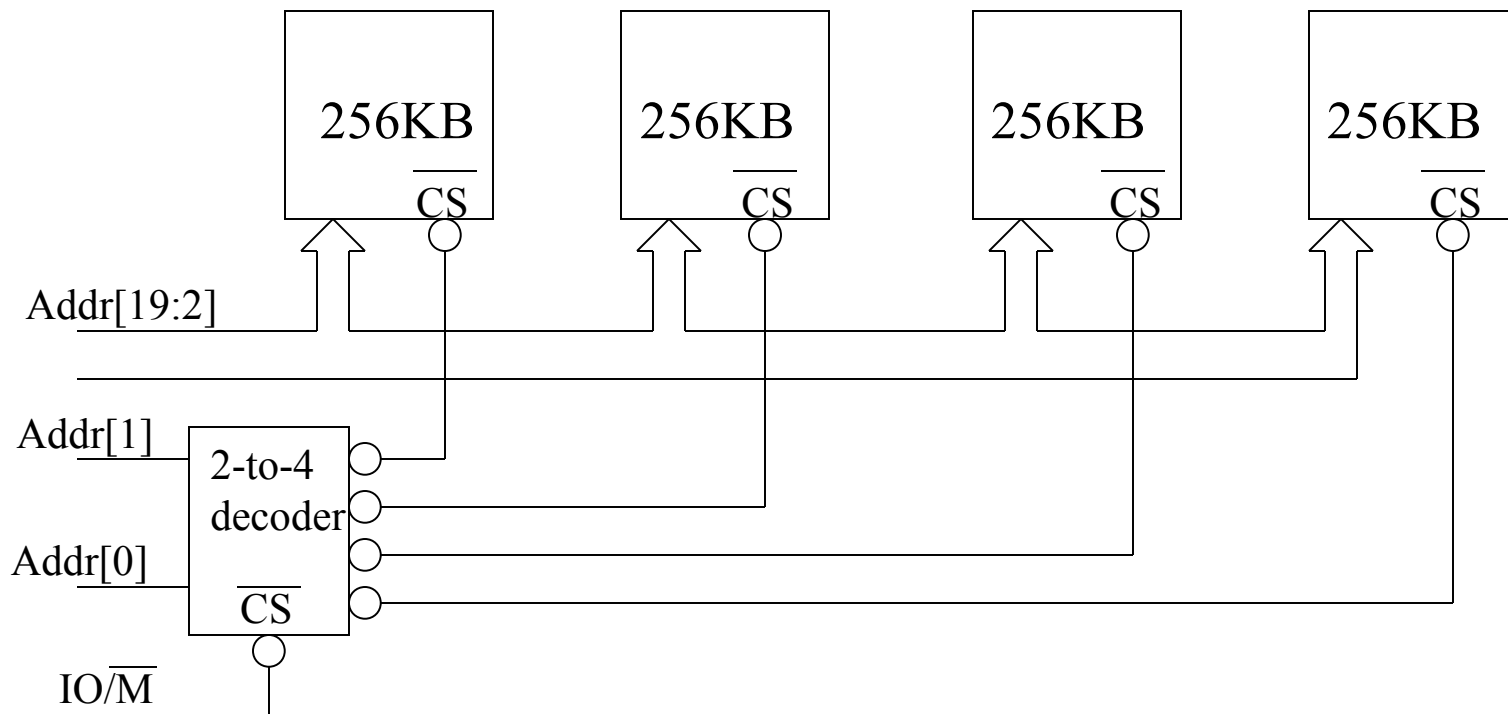
— *Solution 1:*



Memory Address Decoding

- Design a 1MB memory system consisting of multiple memory chips

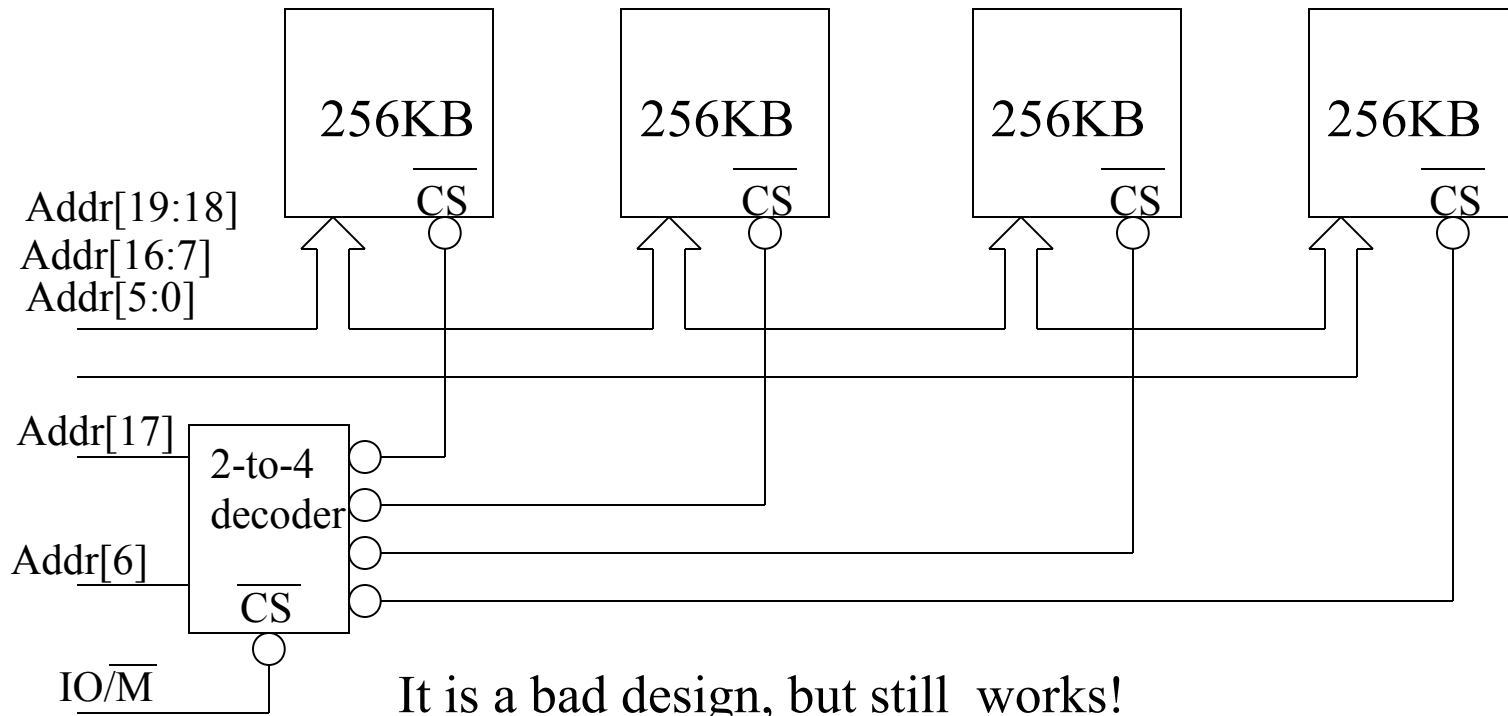
— *Solution 2:*



Memory Address Decoding

- Design a 1MB memory system consisting of multiple memory chips

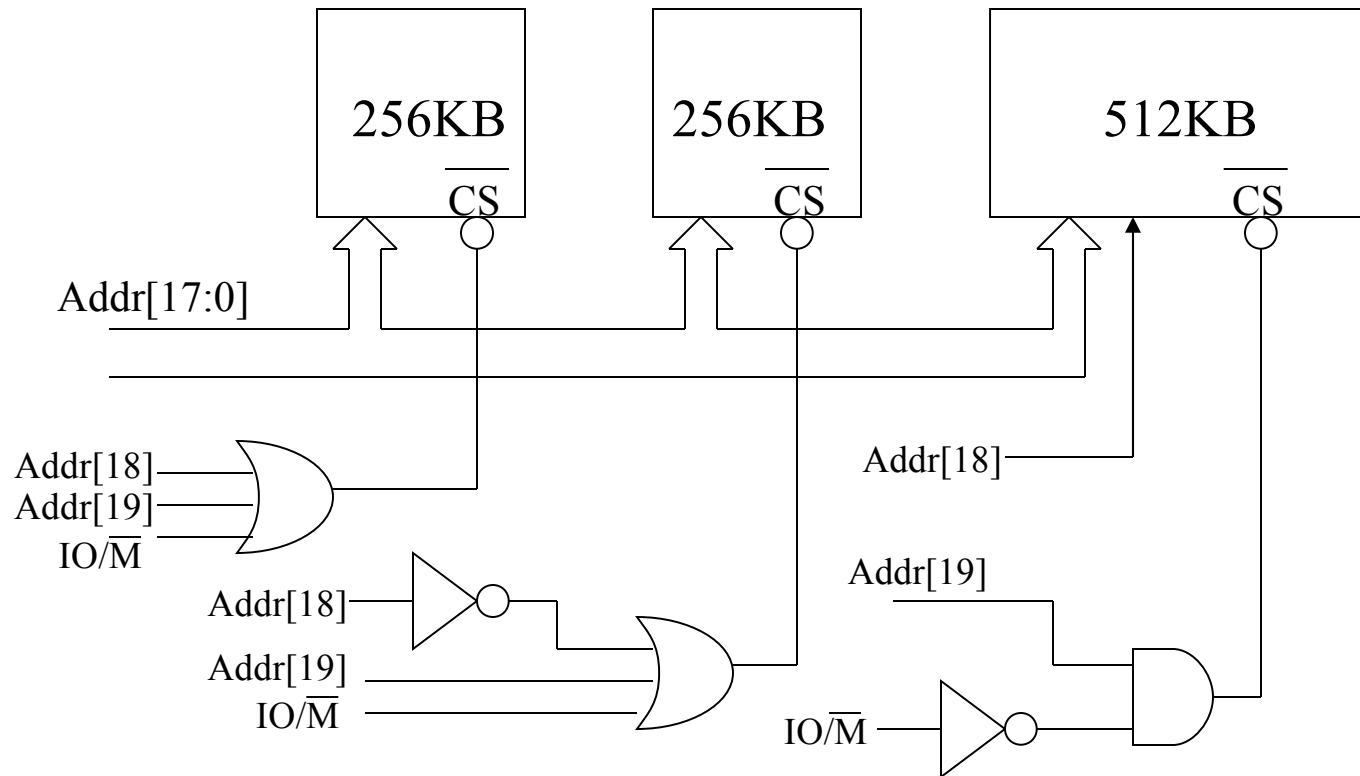
— *Solution 3:*



Memory Address Decoding

- Design a 1MB memory system consisting of multiple memory chips

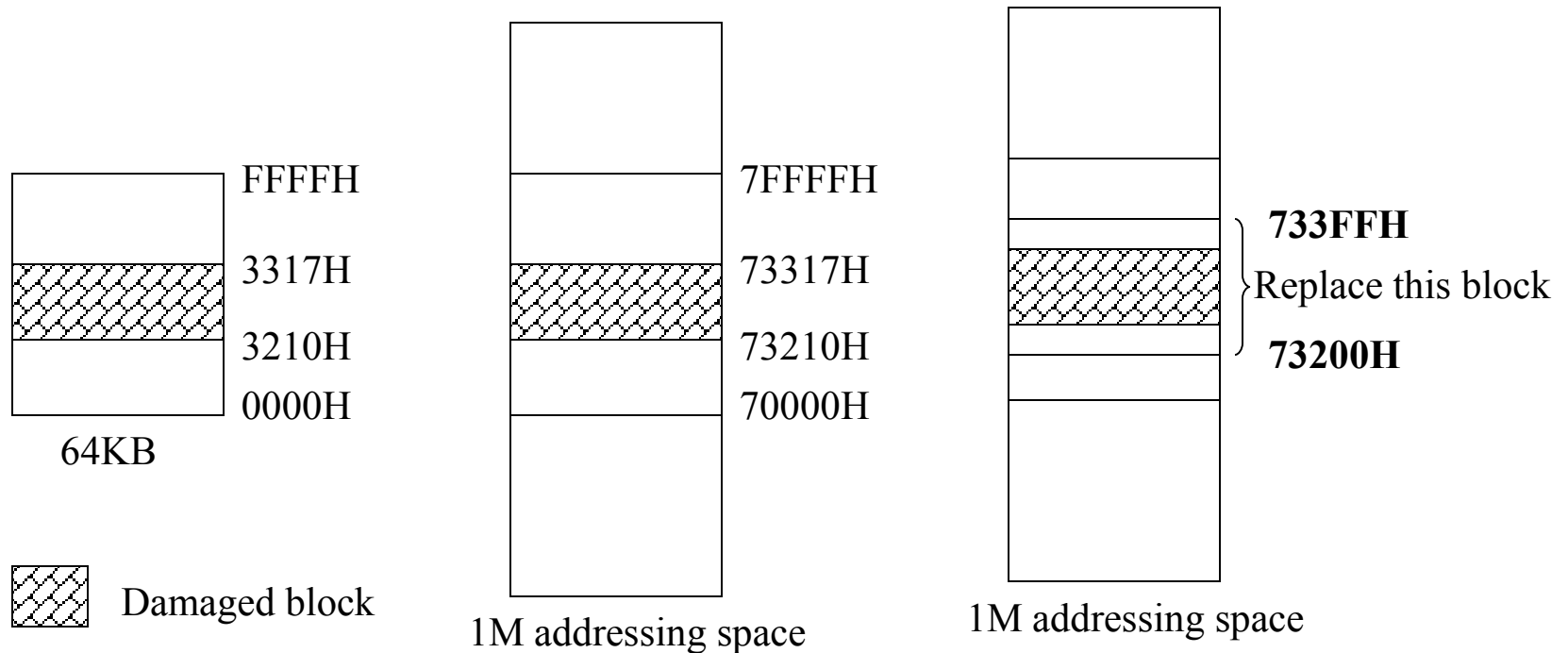
— *Solution 4:*



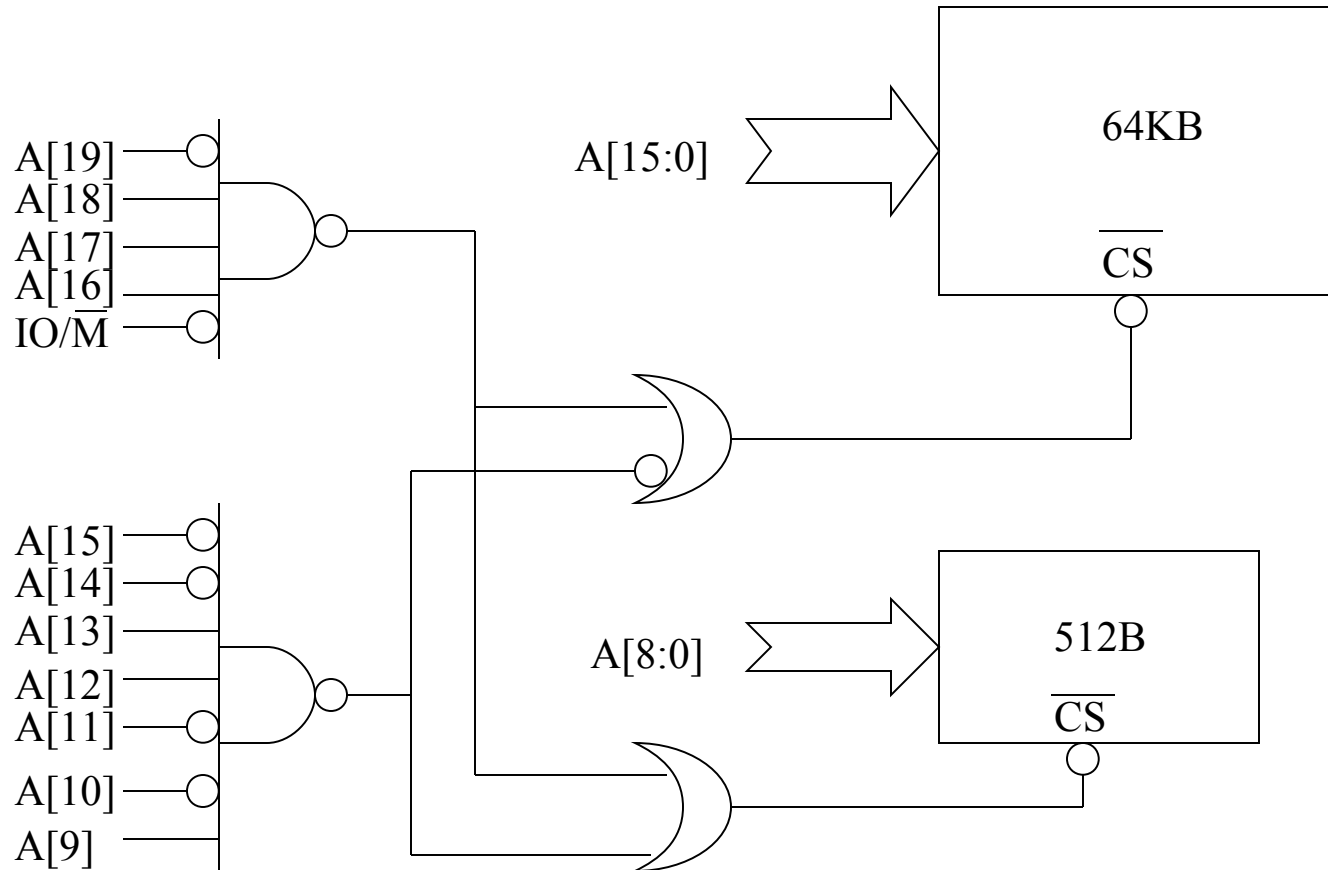
Memory Address Decoding

❑ Exercise Problem:

- A 64KB memory chip is used to build a memory system with the starting address of 7000H. A block of memory locations in the memory chip are damaged.



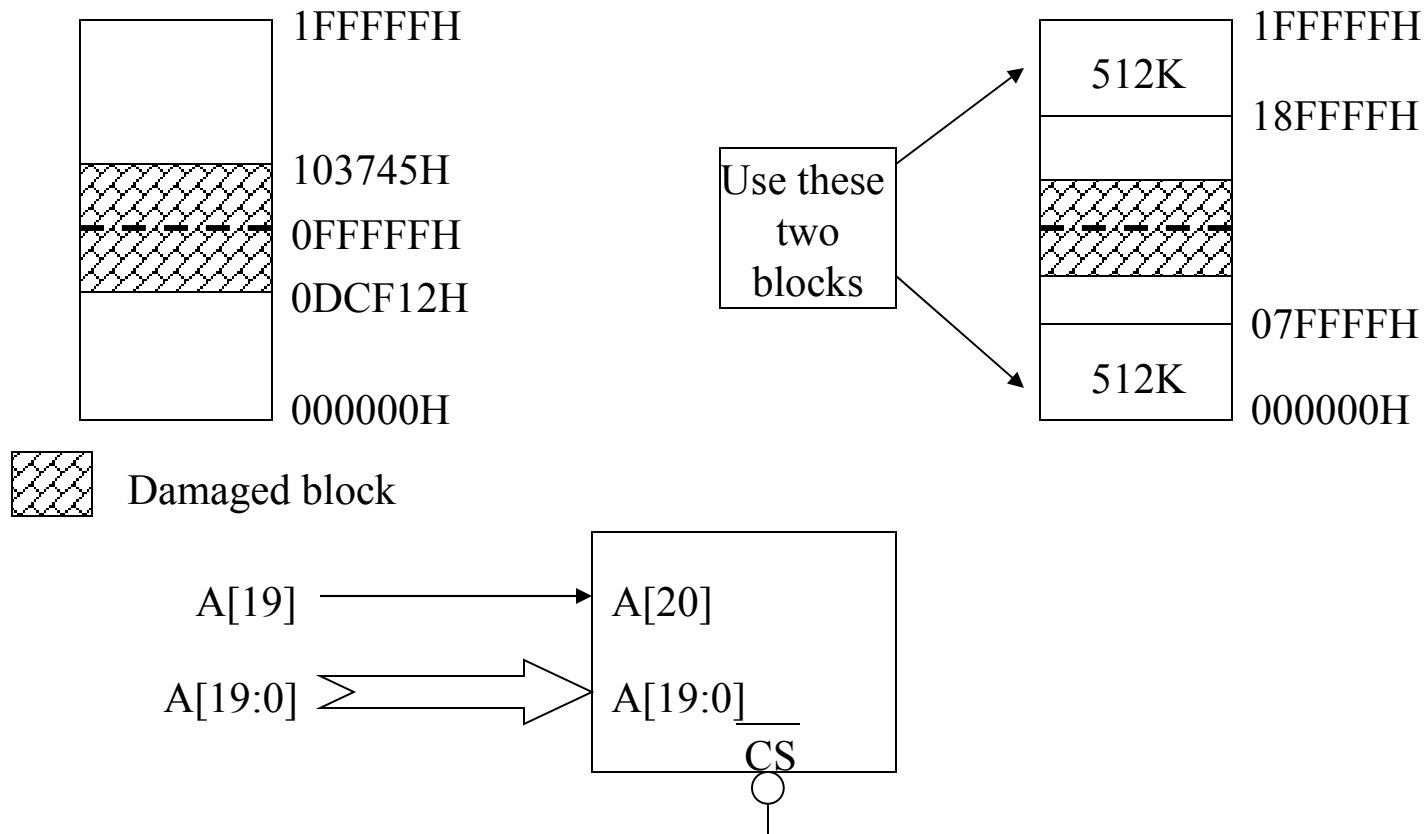
Memory Address Decoding



Memory Address Decoding

❑ Exercise Problem:

- A 2MB memory chip with a damaged block (from 0DCF12H to 103745H) is used to build a 1MB memory system for an 8088-based computer



Memory Address Decoding

❑ Partial decoding

— Example:

- build a 32KB memory system by using four 8KB memory chips
- The starting address of the 32KB memory system is 30000H

		0 0 1 1 0	1 1	1 1 1 1 1 1 1 1 1 1	high addr. of chip #4
		0 0 1 1 0	1 1	0 0 0 0 0 0 0 0 0 0	Low addr. of chip #4
Chip #4	36000H	0 0 1 1 0	1 0	1 1 1 1 1 1 1 1 1 1	high addr. of chip #3
Chip #3	34000H	0 0 1 1 0	1 0	0 0 0 0 0 0 0 0 0 0	Low addr. of chip #3
Chip #2	32000H	0 0 1 1 0	0 1	1 1 1 1 1 1 1 1 1 1	high addr. of chip #2
Chip #1	30000H	0 0 1 1 0	0 1	0 0 0 0 0 0 0 0 0 0	Low addr. of chip #2
		0 0 1 1 0	0 0	1 1 1 1 1 1 1 1 1 1	high addr. of chip #1
		0 0 1 1 0	0 0	0 0 0 0 0 0 0 0 0 0	Low addr. of chip #1

Memory Map - Partial address decoding

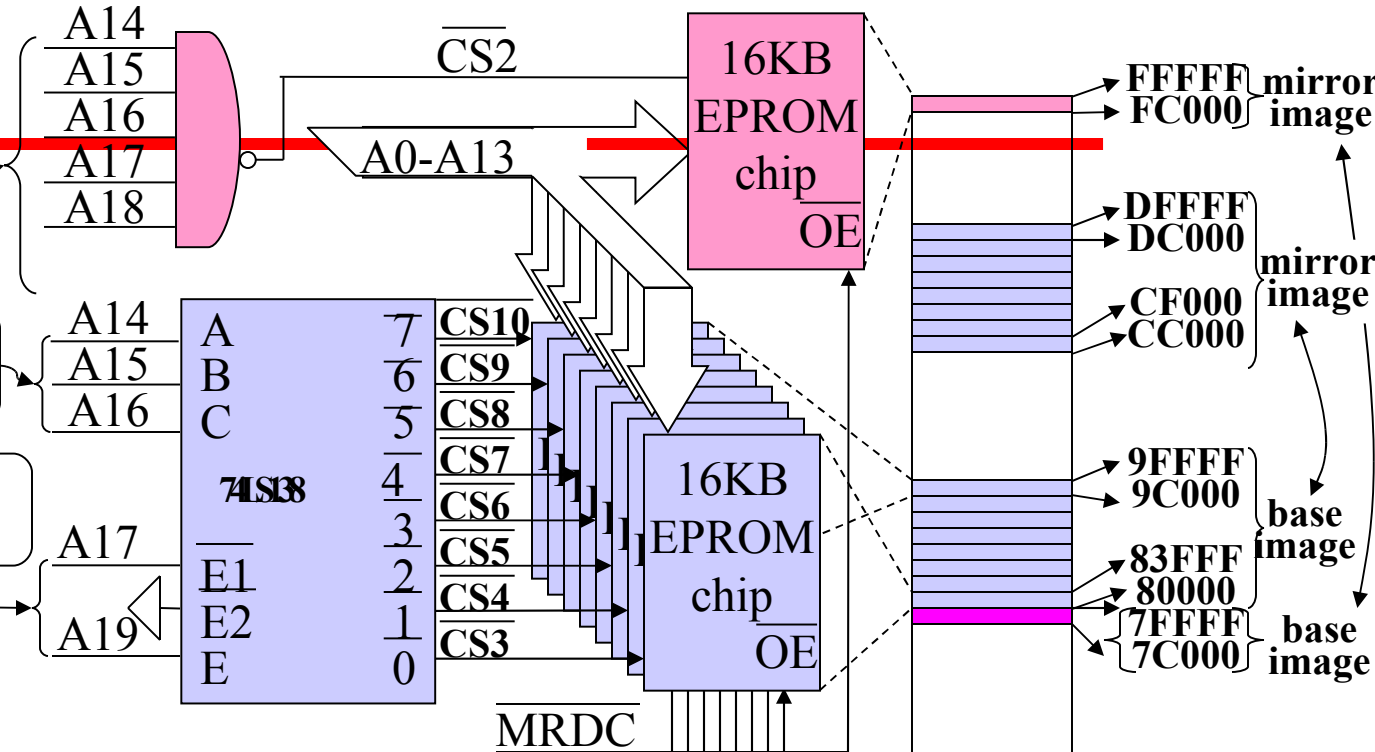
Some address lines **not** used by the decoder or memory chip
=> mirror images = **partial address decoding**

decoding

A18=...=A14=1
select the EPROM

A16, A15, A14 select
one EPROM chip

A19 = 1, A17 = 0
activate the decoder



The same Memory-map assignment

A 64Kbyte = 2^{16} RAM chip has
16 address lines, A0 - A15

A19, A18 assigned to 00 =>
CS active for every address
from 00000 to 3FFFF

$\text{IO}/\overline{\text{M}} = 0 \Rightarrow$ Memory map

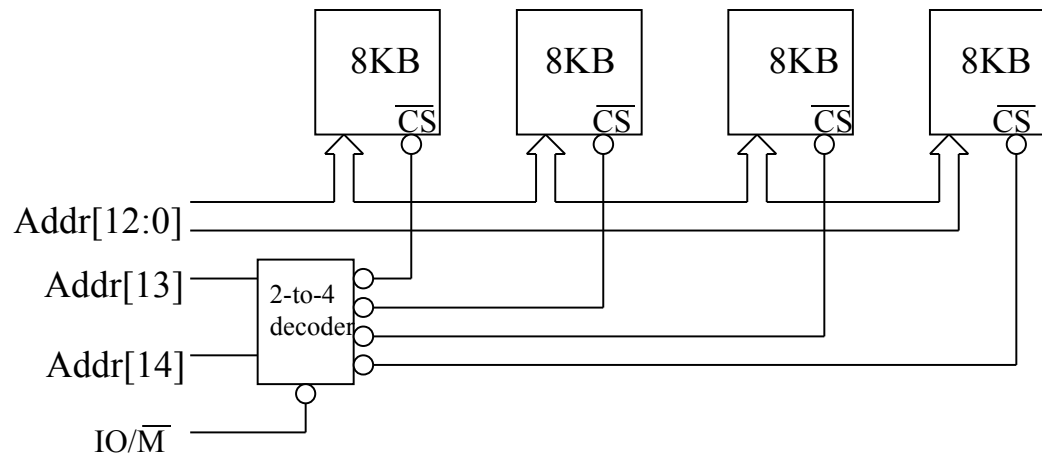
$\begin{cases} \text{A18} = 0 \\ \text{A19} = 0 \\ \text{IO}/\overline{\text{M}} = 0 \end{cases}$

A16, A17 not used => four images for the same chip

8088 Memory Map

Memory Address Decoding

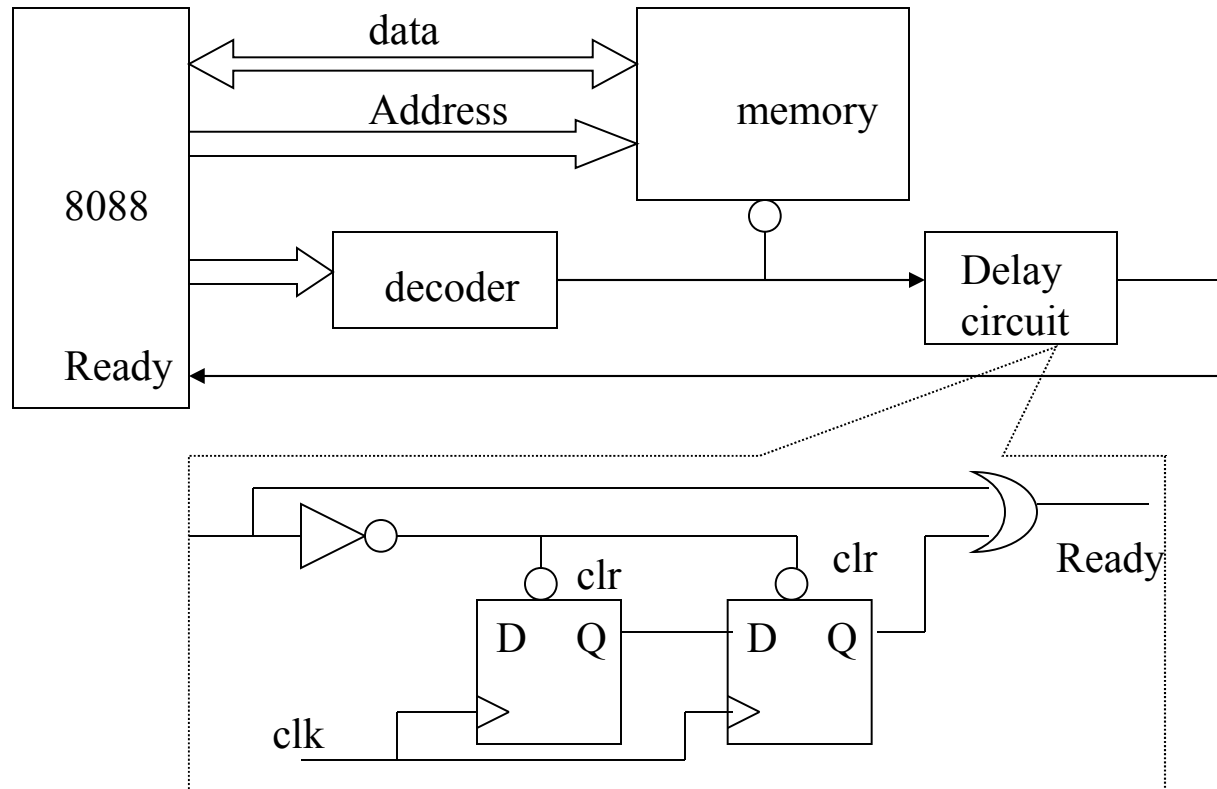
— Implementation of partial decoding



- ❑ With the above decoding scheme, what happens if the processor accesses location 02117H, 32117H, and 9A117H?
- ❑ If two 16KB memory chips are used to implement the 32KB memory system, what is the partial decoding circuit?
- ❑ What are the advantage and disadvantage of partial decoding circuits?

Generating Wait States

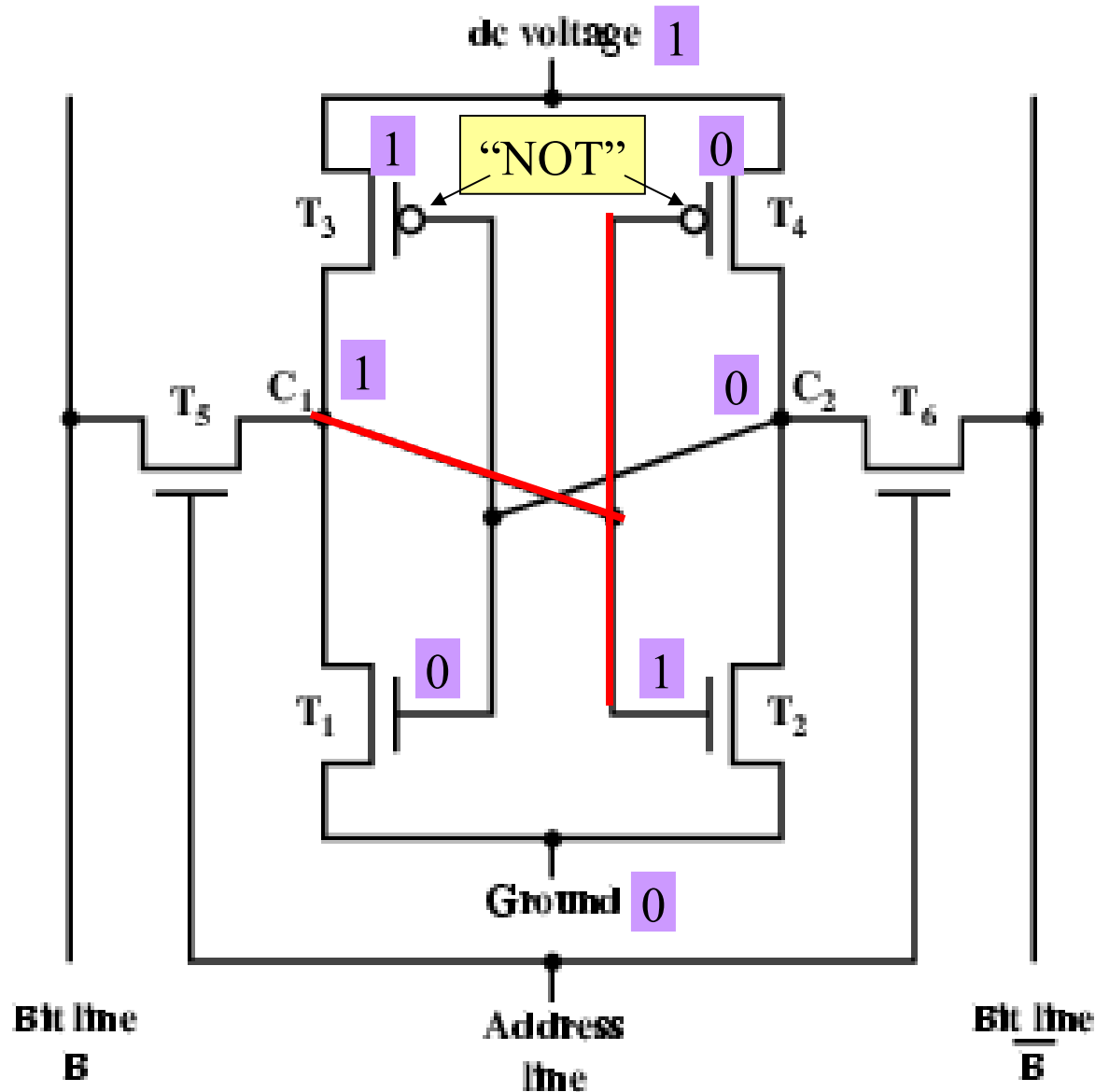
- ❑ Wait states are inserted into memory read or write cycles if slow memories are used in computer systems
- ❑ Ready signal is used to indicate if wait states are needed



1. Static RAM (SRAM)

- Essentially uses flip-flops to store charge (transistor circuit)
- As long as power is present, transistors do not lose charge (no refresh)
- Very fast (no sense circuitry to drive nor charge depletion)
- Complex construction
- Large bit circuit
- Expensive
- Used for Cache RAM because of speed and no need for large volume

Static RAM Structure



six transistors
per bit
(flip flop)

0/1 = example

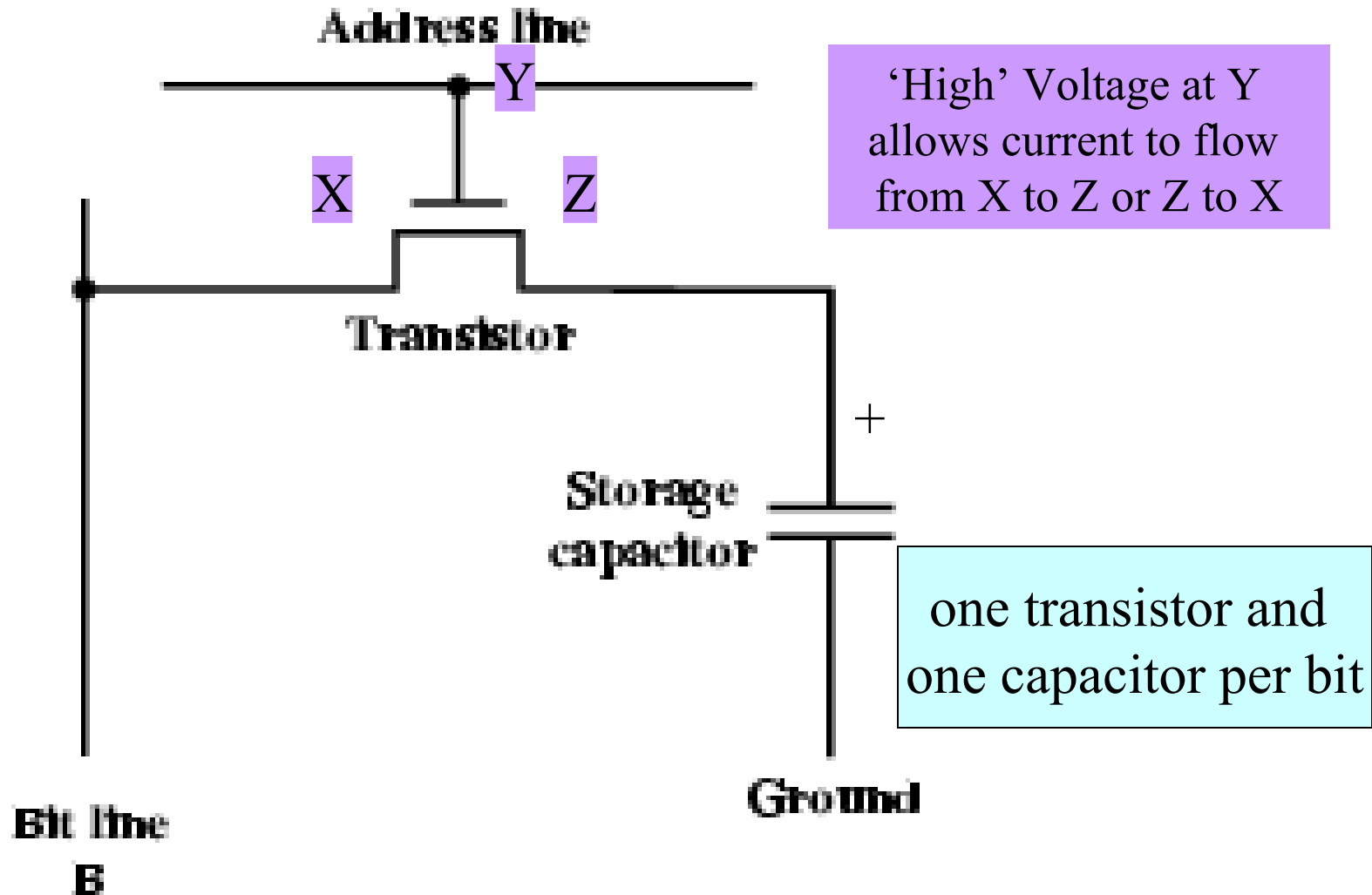
Static RAM Operation

- Transistor arrangement (flip flop) has 2 stable logic states
- Write
 - 1.signal bit line: High \rightarrow 1 Low \rightarrow 0
 - 2.address line active \rightarrow “switch” flip flop to stable state matching bit line
- Read
 - 1.address line active
 - 2.drive bit line to same state as flip flop

2. Dynamic RAM (DRAM)

- Bits stored as charge in capacitors
- Simpler construction
- Smaller per bit
- Less expensive
- Slower than SRAM
- Typical application is main memory
- Essentially analogue -- level of charge determines value

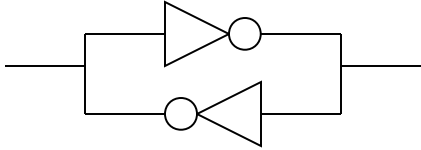
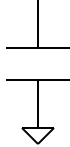
Dynamic RAM Structure



DRAM Operation

- Address line **active**
 - transistor switch closed and current flows
- **Write**
 1. data signal to bit line: High → 1 Low → 0
 2. address line active → transfers charge from bit line to capacitor
- **Read**
 1. address line active
 2. transfer charge from capacitor to bit line (then to amplifier)
 3. capacitor **charge must be restored !**

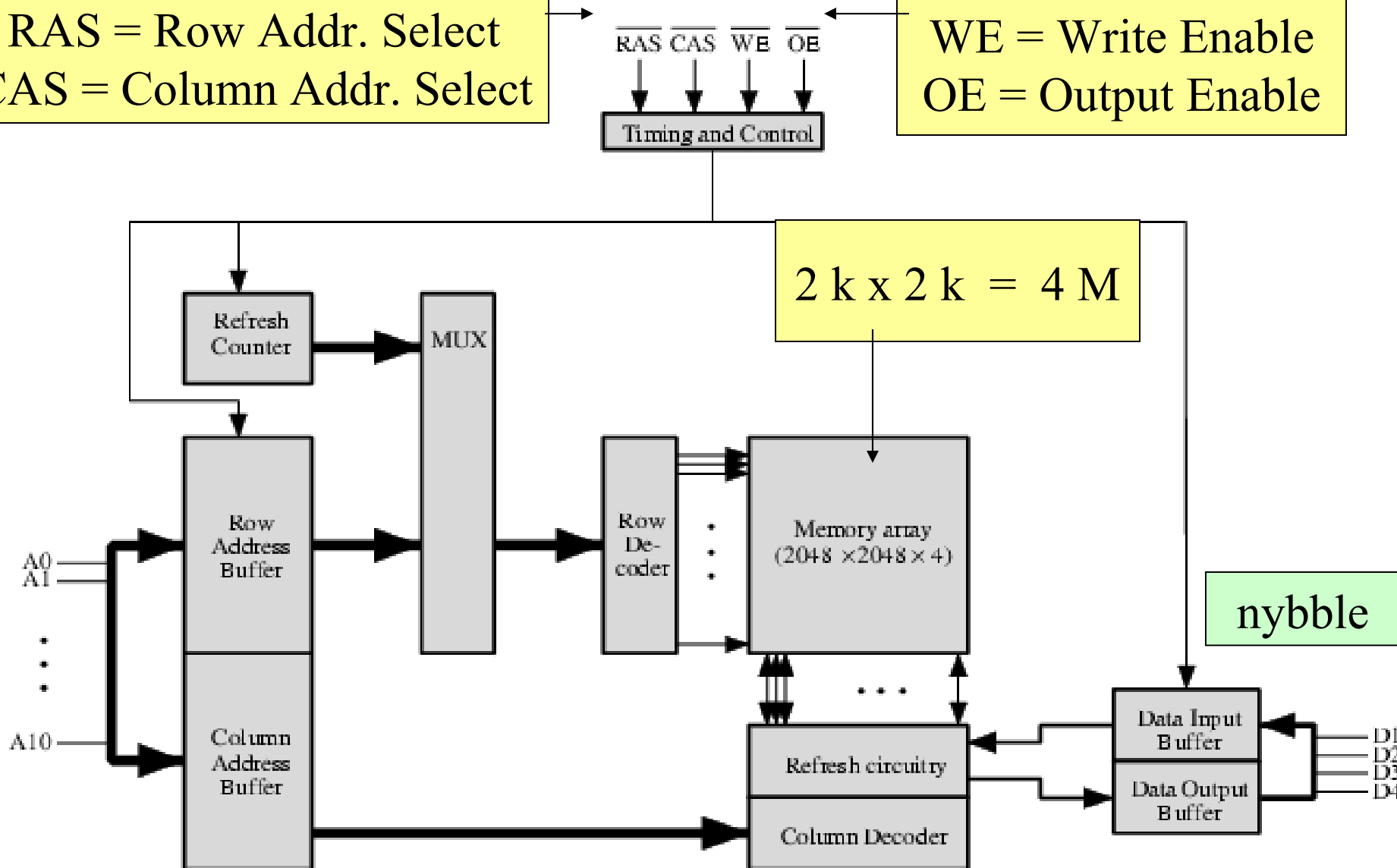
SRAM v.s. DRAM

	Static Random Access Memory (SRAM)	Dynamic Random Access Memory (DRAM)
Storage element		
Advantages	<ol style="list-style-type: none">1. Fast2. No refreshing operations	<ol style="list-style-type: none">1. High density and less expensive
Disadvantages	<ol style="list-style-type: none">1. Large silicon area2. expensive	<ol style="list-style-type: none">1. Slow2. Require refreshing operations
Applications	High speed memory applications, Such as cache	Main memories in computer systems

Typical 16 Mb DRAM (4M x 4)

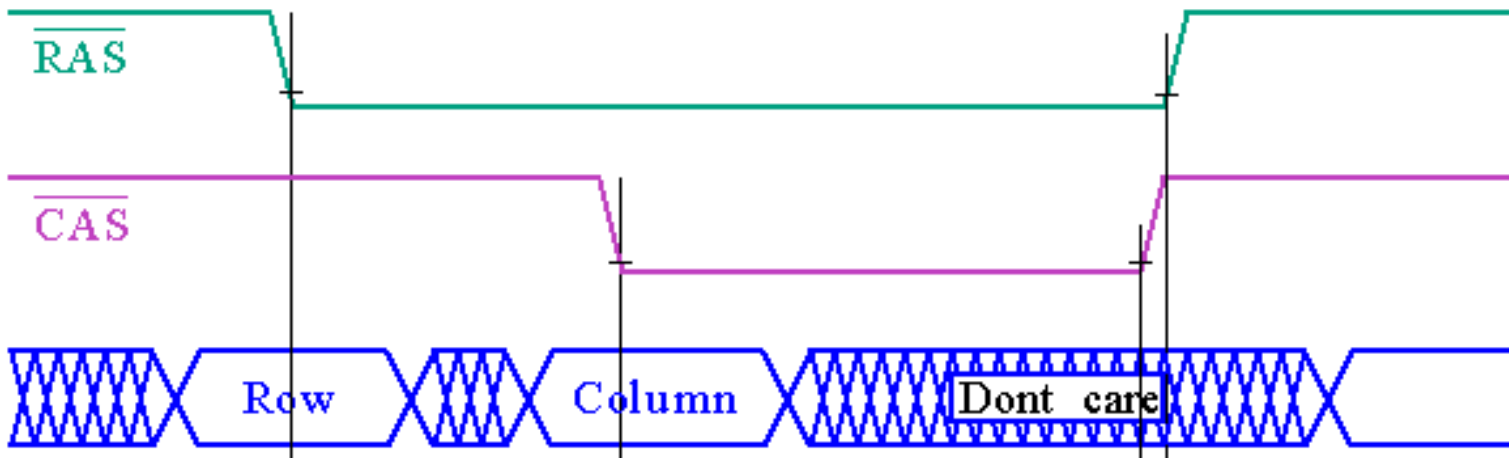
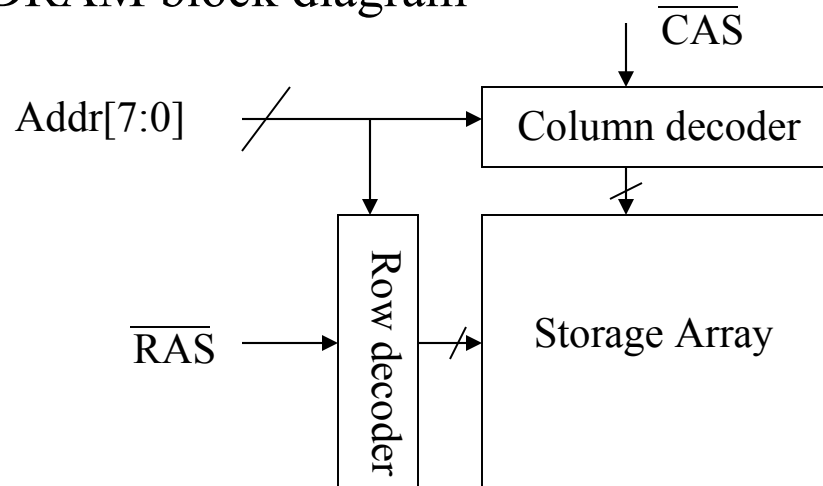
RAS = Row Addr. Select
CAS = Column Addr. Select

WE = Write Enable
OE = Output Enable



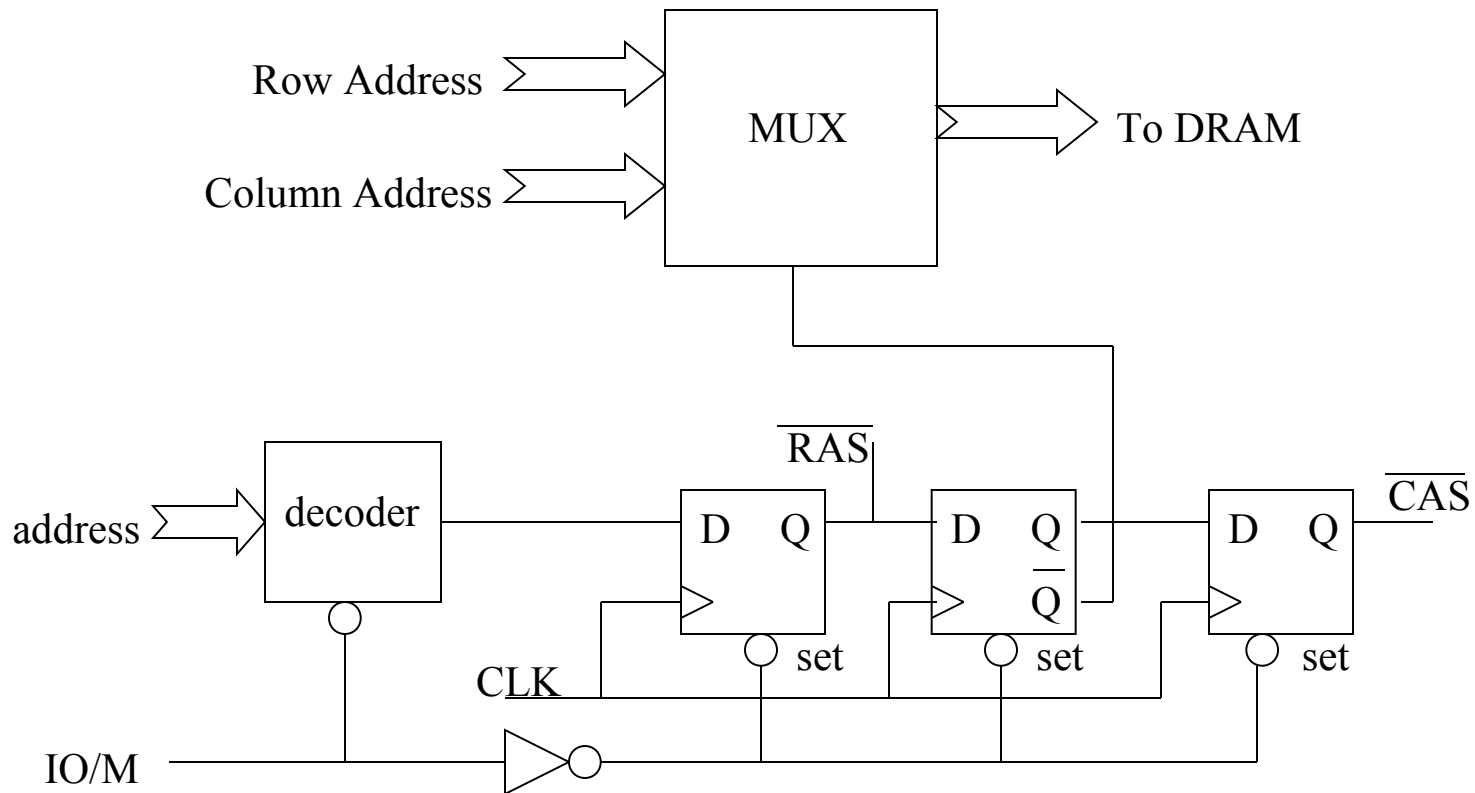
Accessing DRAMs

□ DRAM block diagram



Accessing DRAMs

❑ Address bus selection circuit



Accessing DRAMs

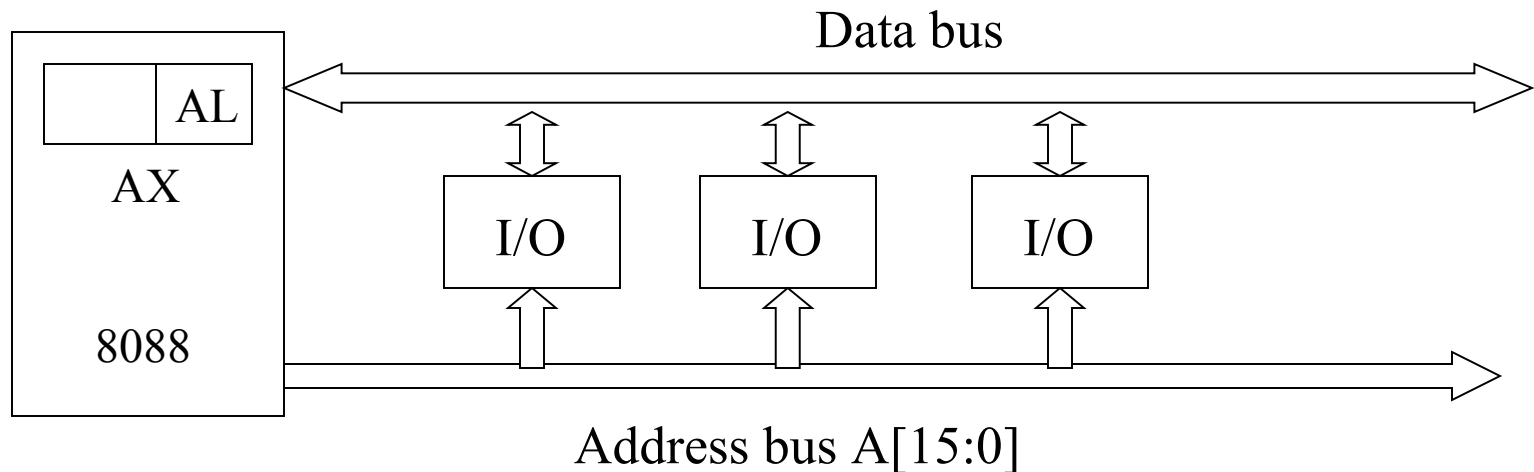
❑ Refreshing operations

- Because leakage current will destroy information stored on DRAM capacitors periodic refreshing operations are required for DRAM circuits
- During refreshing operation, DRAM circuit are not able to response processor's request to perform read or write operations
- How to suspend memory operations?
- DRAM controllers are developed to take care DRAM refreshing operations

I/O System Design

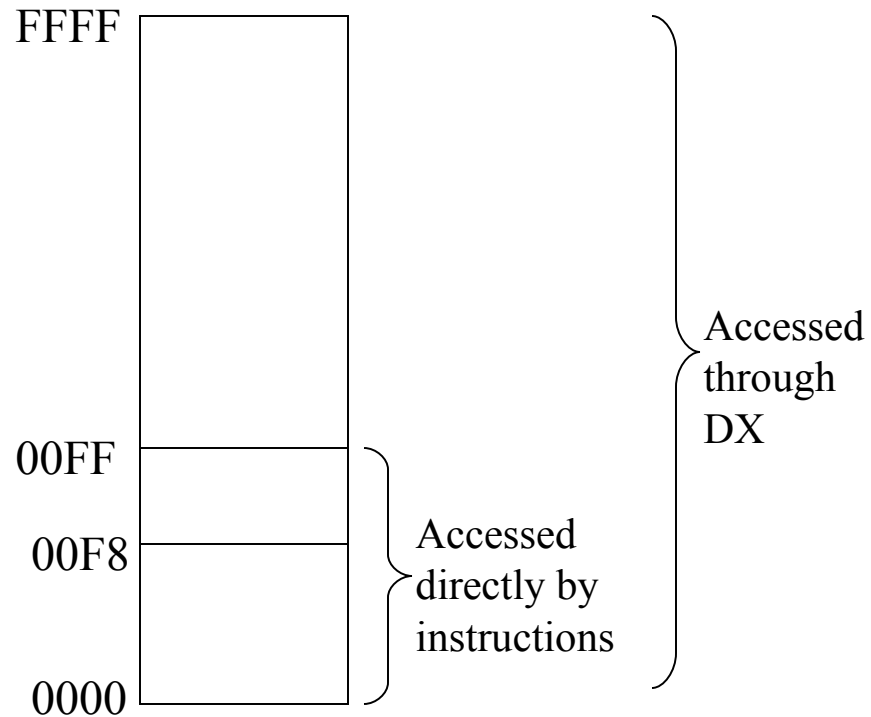
Overview of 8088 I/O System

- ❑ 65,536 possible I/O ports
- ❑ Data transfer between ports and the processor is over data bus
- ❑ 8088 uses address bus A[15:0] to locate an I/O port
- ❑ AL (or AX) is the processor register that takes input data (or provide output data)



8088 Port Addressing Space

□ Addressing Space



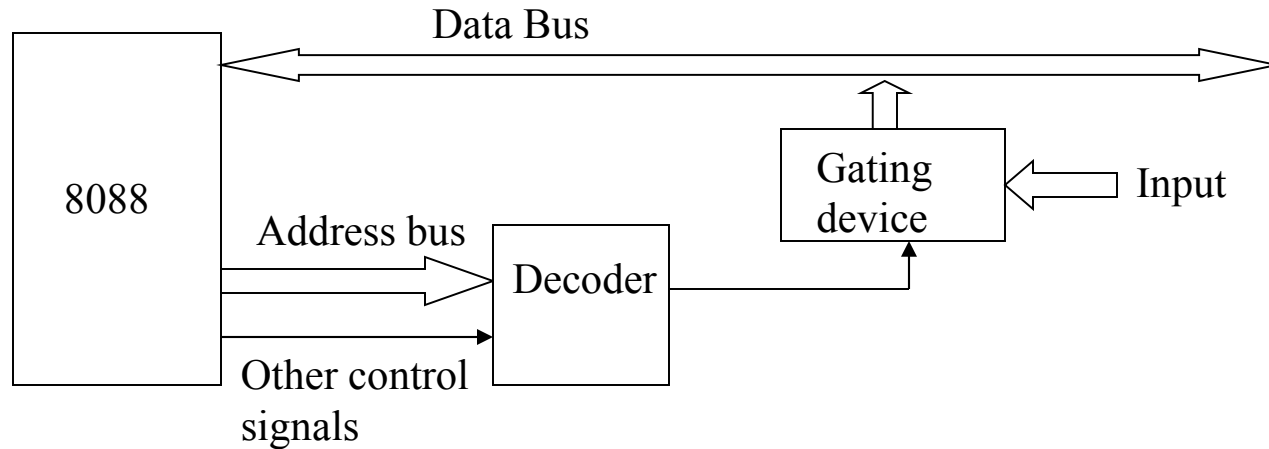
□ Accessing directly by instructions

```
IN    AL,    80H
IN    AX,    6H
OUT   3CH,   AL
OUT   0A0H,  AX
```

□ Accessing through DX

```
IN    AL,    DX
IN    AX,    DX
OUT   DX,    AL
OUT   DX,    AX
```

Input Port Implementation

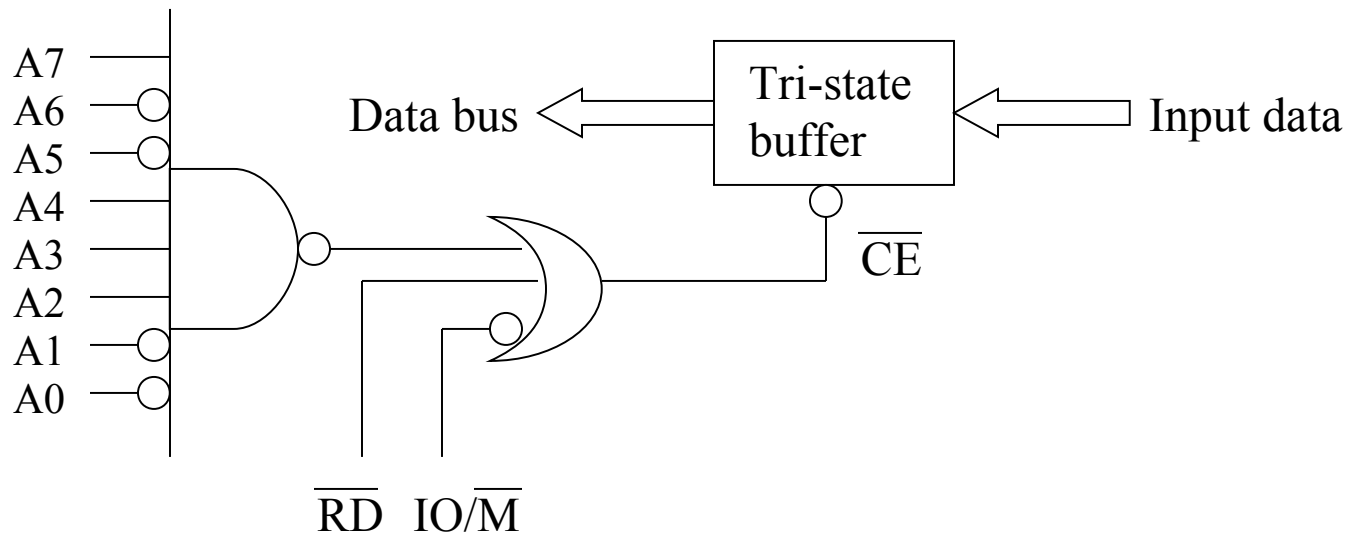


- The outputs of the gating device are high impedance when the processor is not accessing the input port
- When the processor is accessing the input port, the gating device transfers input data to CPU data bus
- The decoding circuit controls when the gating device has high impedance output and when it transfers input data to data bus

Input Port Implementation

□ Circuit Implementation

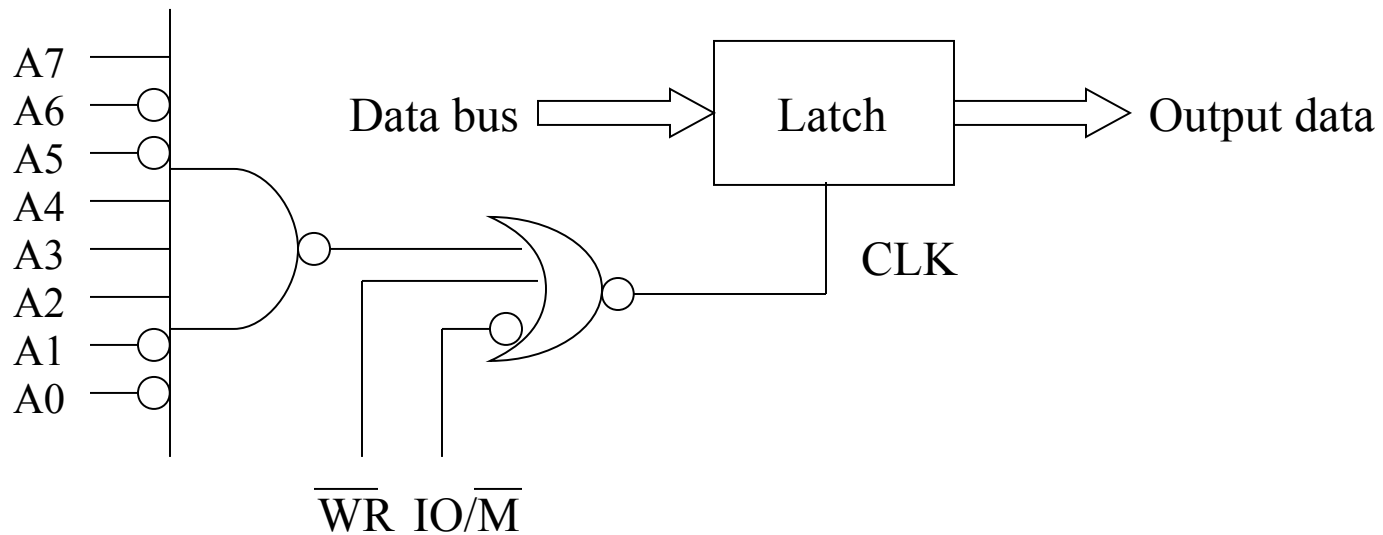
— Assume that the address of the input port is 9CH



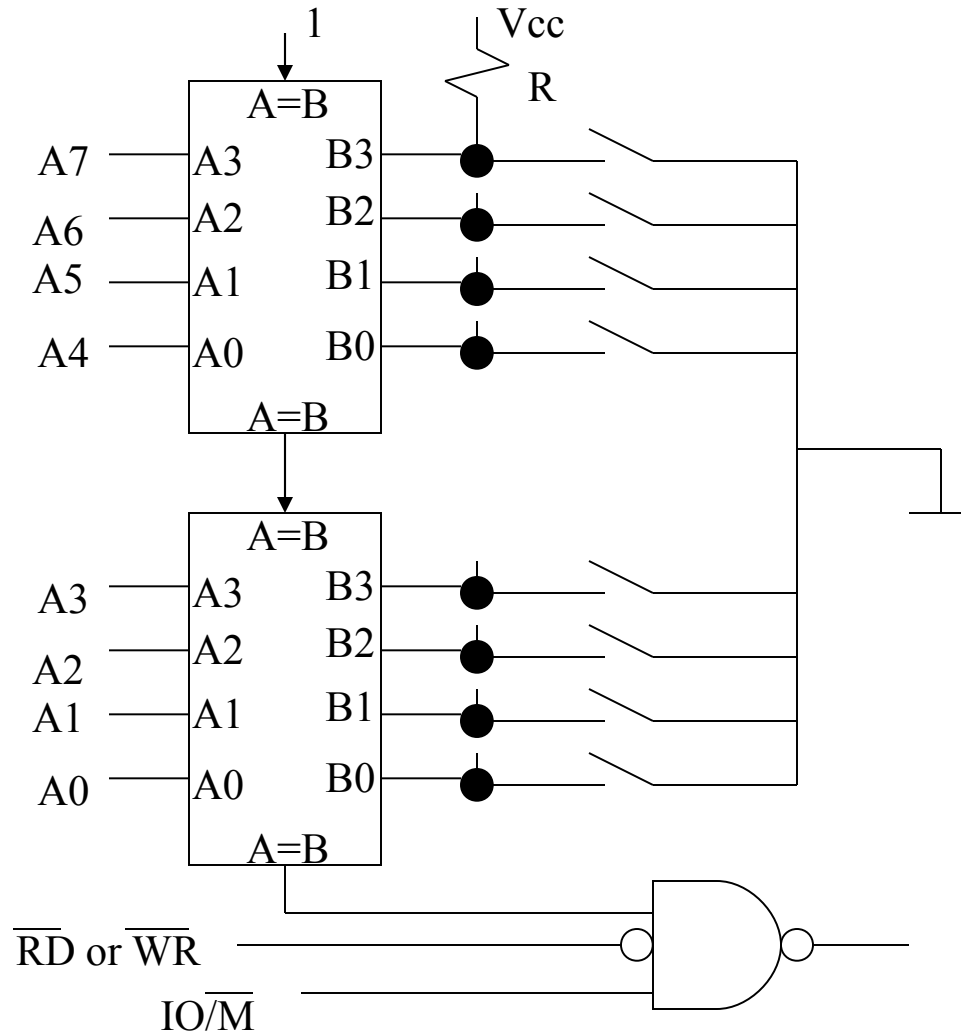
Output Port Implementation

□ Circuit Implementation

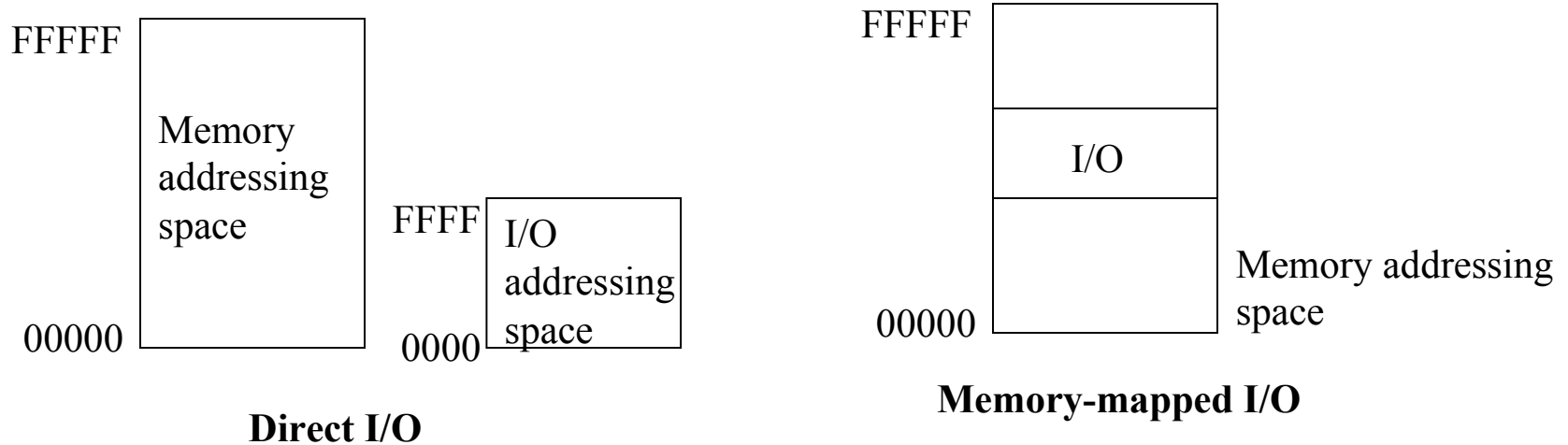
— Assume that the address of the output port is 9CH



A Reconfigurable Port Decoder



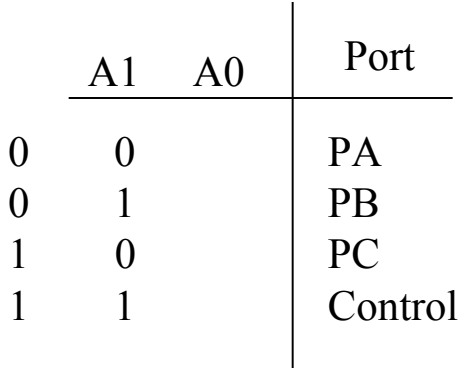
Direct I/O v.s. Memory-Mapped I/O



- ❑ Direct I/O: I/O addresses are separated from memory address
 - Advantage: Do not take memory addressing space
 - Disadvantage: Use only AL or AX transferring data
- ❑ Memory-mapped I/O: I/O ports are treated as memory locations
 - Advantage: Accessing I/O ports is like accessing memory locations
 - Can use other instructions to access I/O ports
 - Disadvantage: Take memory addressing space

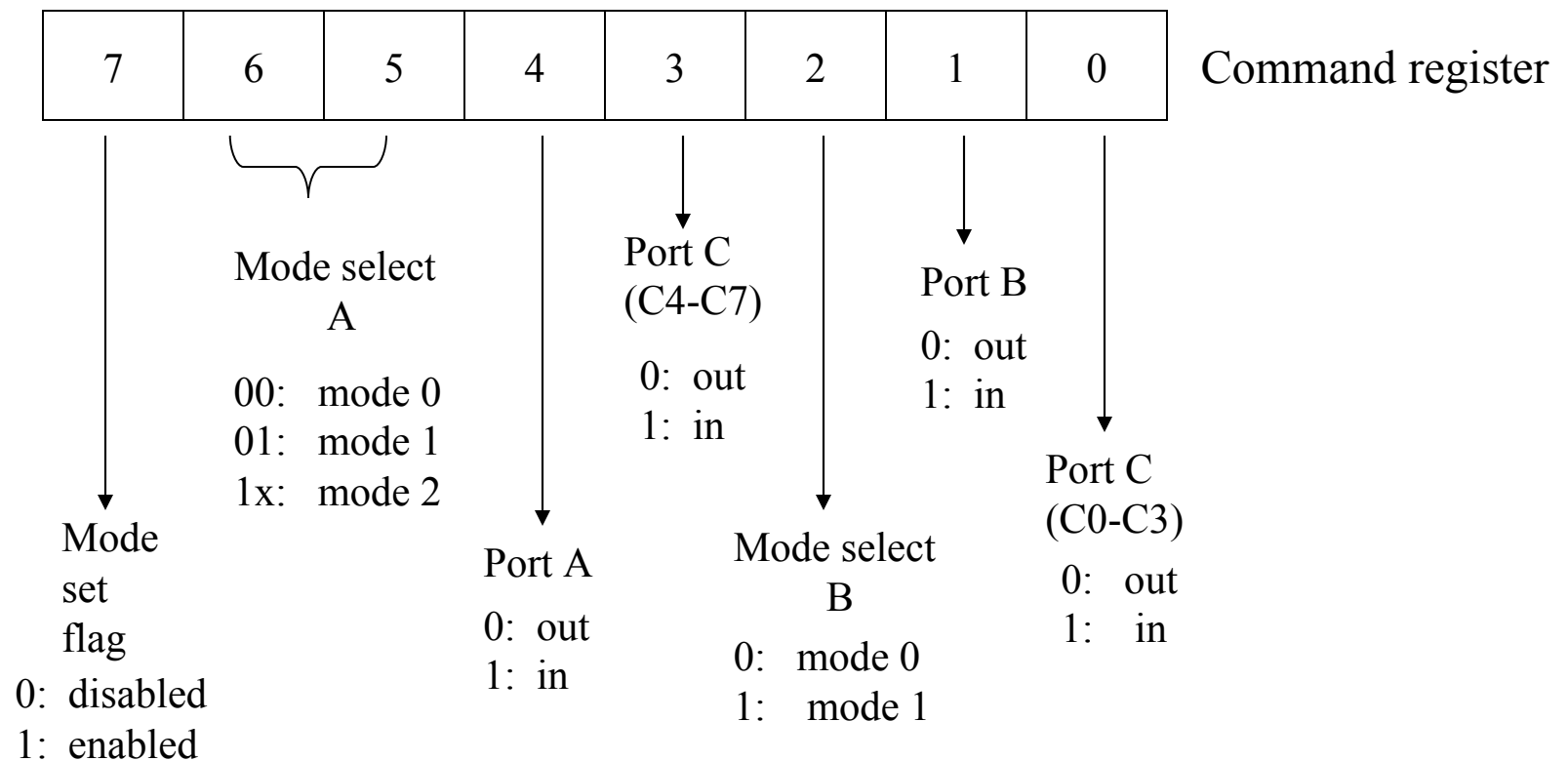
Handshaking

- I/O devices are typically slower than the microprocessor.
- Handshaking is used to synchronize I/O with the microprocessor.
 - A device indicates that it is ready for a command or data (through some I/O pin or port).
 - The processor issues a command to the device, and the device indicates it is busy (not ready).
 - The I/O device finishes its task and indicates a ready condition, and the cycle continues.
- There are two basic mechanisms for the processor to service a device.
 - Polling: Processor initiated. Device indicates it is ready by setting some status bit and the processor periodically checks it.
 - Interrupts: Device initiated. The act of setting a status bit causes an interrupt, and the processor calls an ISR to service the device.



Programming 8255

- ❑ 8255 has three operation modes: *mode 0*, *mode 1*, and *mode 2*



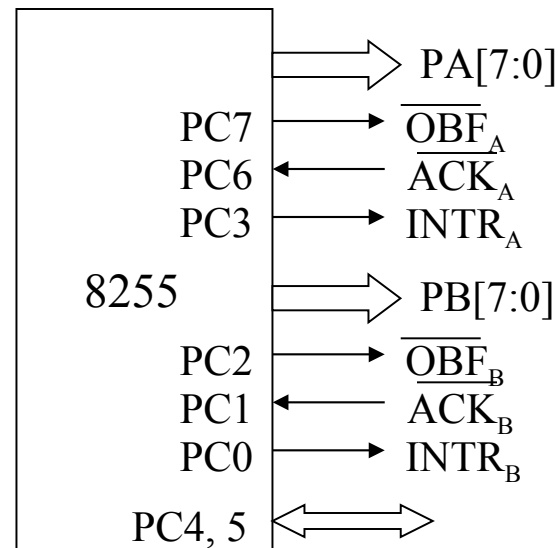
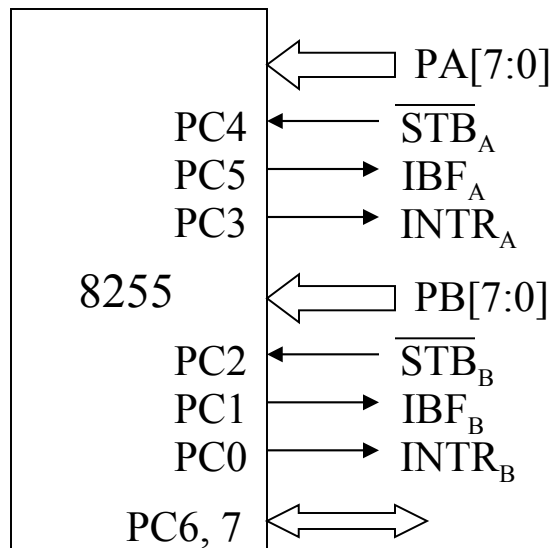
Programming 8255

□ Mode 0:

- Ports A, B, and C can be individually programmed as input or output ports
- Port C is divided into two 4-bit ports which are independent from each other

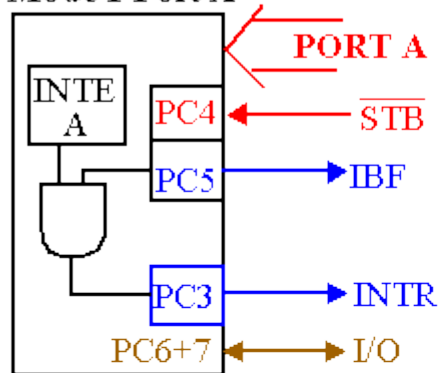
□ Mode 1:

- Ports A and B are programmed as input or output ports
- Port C is used for handshaking

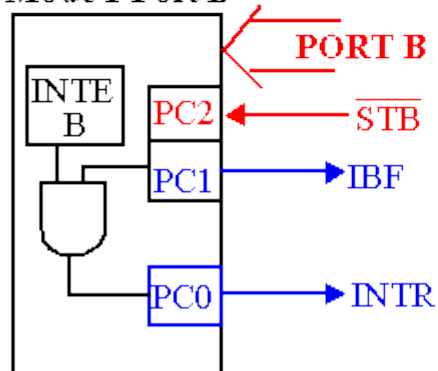


- STB** The strobe input loads data into the port latch on a 0-to-1 transition
- IBF** **Input buffer full** is an output indicating that the input latch contain information
- INTR** **Interrupt request** is an output that requests an interrupt
- INTE** The **interrupt enable signal** is neither an input nor an output; it is an internal bit programmed via the PC4(port A) or PC2(port B) bits.
- PC7,PC6** The port C pins 7 and 6 are general-purpose I/O pins that are available for any purpose.

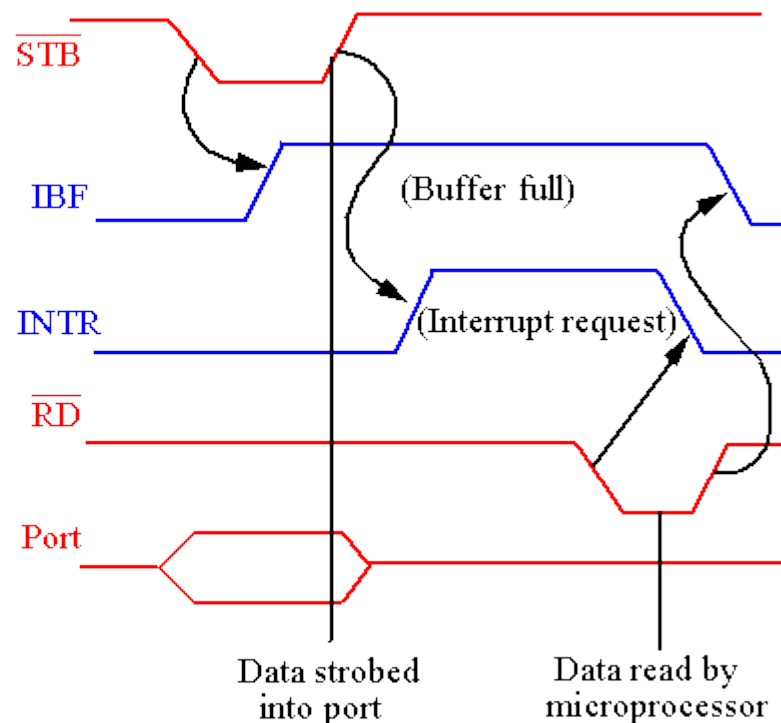
Mode 1 Port A



Mode 1 Port B



Timing Diagram



$\overline{\text{OBF}}$ **Output buffer full** is an output that goes low when data is latched in either port A or port B. Goes low on $\overline{\text{ACK}}$.

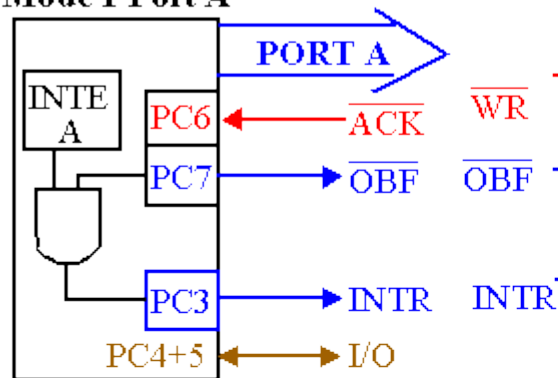
$\overline{\text{ACK}}$ The **acknowledge** signal causes the $\overline{\text{OBF}}$ pin to return to 0. This is a response from an external device.

INTR **Interrupt request** is an output that requests an interrupt

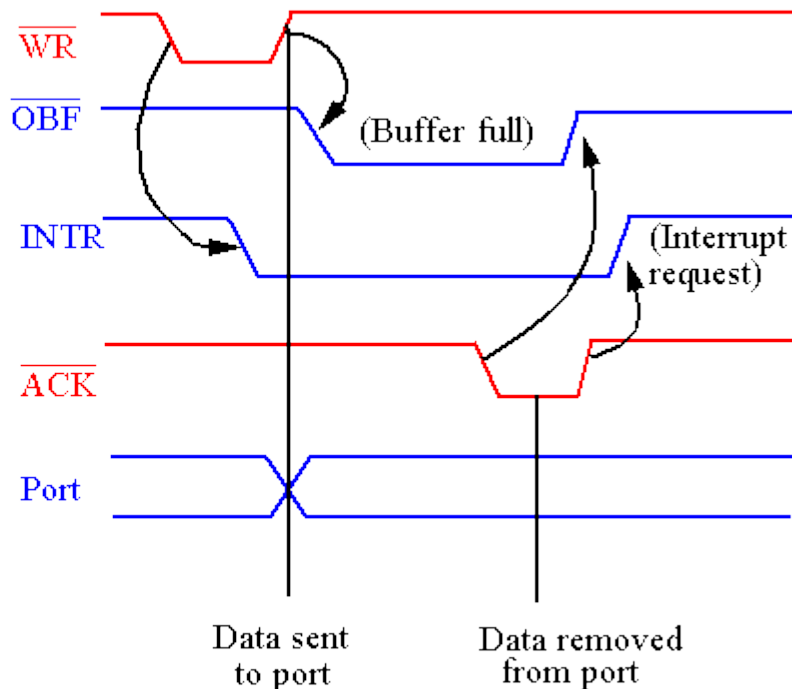
INTE The **interrupt enable signal** is neither an input nor an output; it is an internal bit programmed via the PC6(port A) or PC2(port B) bits.

PC5,PC4 The port C pins 5 and 4 are general-purpose I/O pins that are available for any purpose.

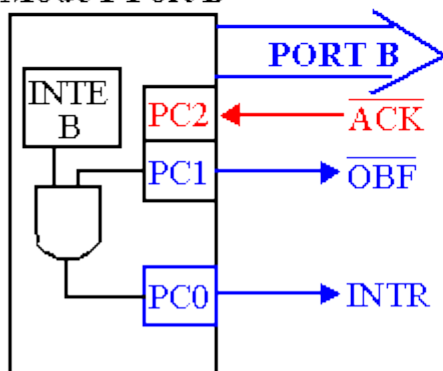
Mode 1 Port A



Timing Diagram



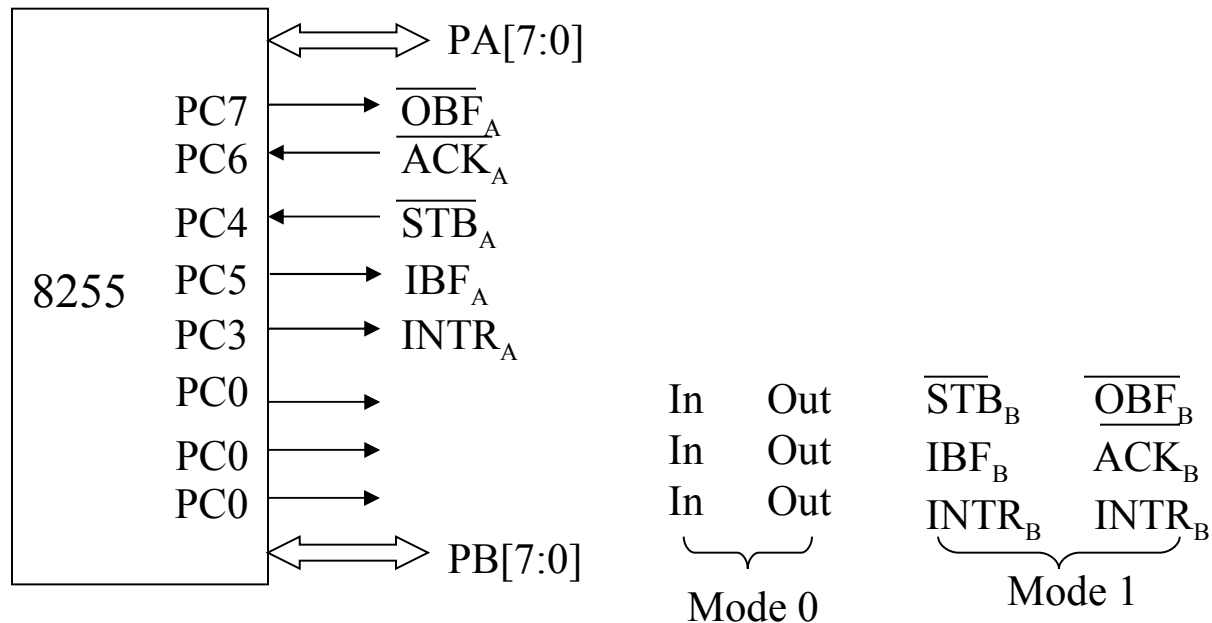
Mode 1 Port B



Programming 8255

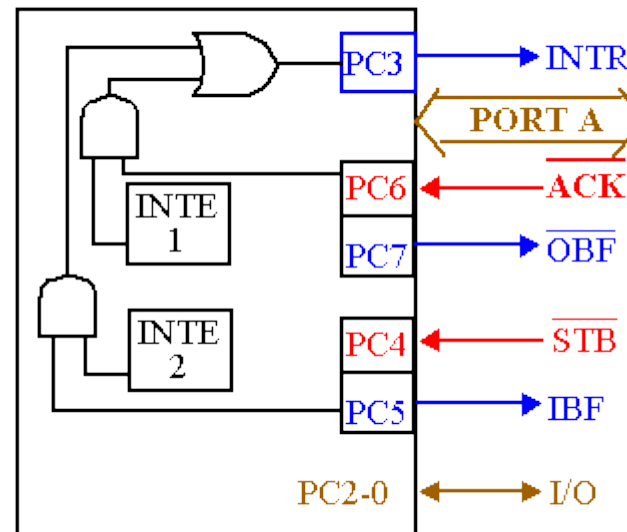
□ Mode 2:

- Port A is programmed to be bi-directional
- Port C is for handshaking
- Port B can be either input or output in mode 0 or mode 1



1. Can you design a decoder for an 8255 chip such that its base address is 40H?
2. Write the instructions that set 8255 into mode 0, port A as input, port B as output, PC0-PC3 as input, PC4-PC7 as output ?

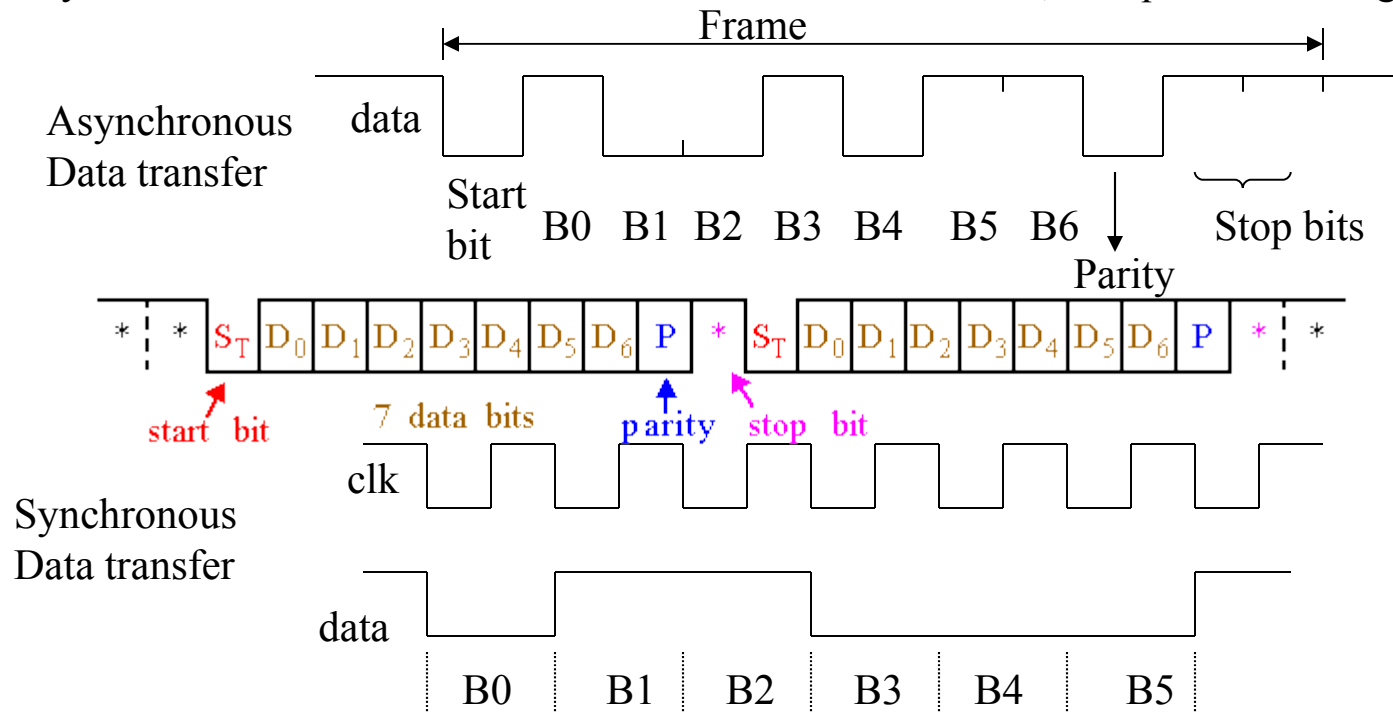
INTR	Interrupt request is an output that requests an interrupt
$\overline{\text{OBF}}$	Output buffer full is an output indicating that the output buffer contains data for the bi-directional bus
$\overline{\text{ACK}}$	Acknowledge is an input that enables tri-state buffers which are otherwise in their high-impedance state
$\overline{\text{STB}}$	The strobe input loads data into the port A latch
IFB	Input buffer full is an output indicating that the input latch contains information for the external bi-directional bus
INTE	Interrupt enable are internal bits that enable the INTR pin. Bit PC6(INTE1) and PC4(INTE2)
PC2,PC1 and PC0	Theses port C pins are general-purpose I/O pins that are available for any purpose.



Serial Data Transfer

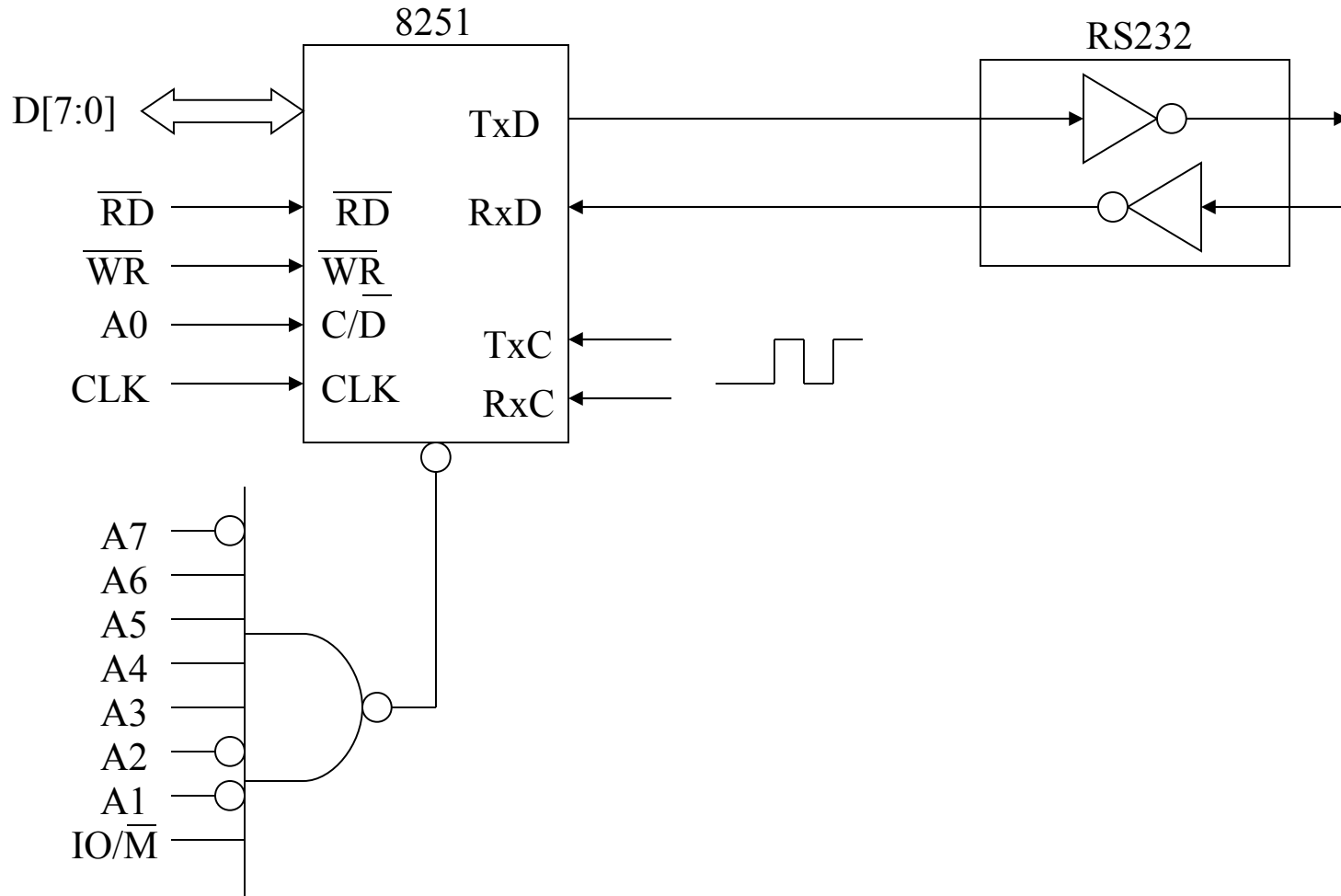
□ Asynchronous v.s. Synchronous

- Asynchronous transfer does not require clock signal. However, it transfers extra bits (start bits and stop bits) during data communication
- Synchronous transfer does not transfer extra bits. However, it requires clock signal



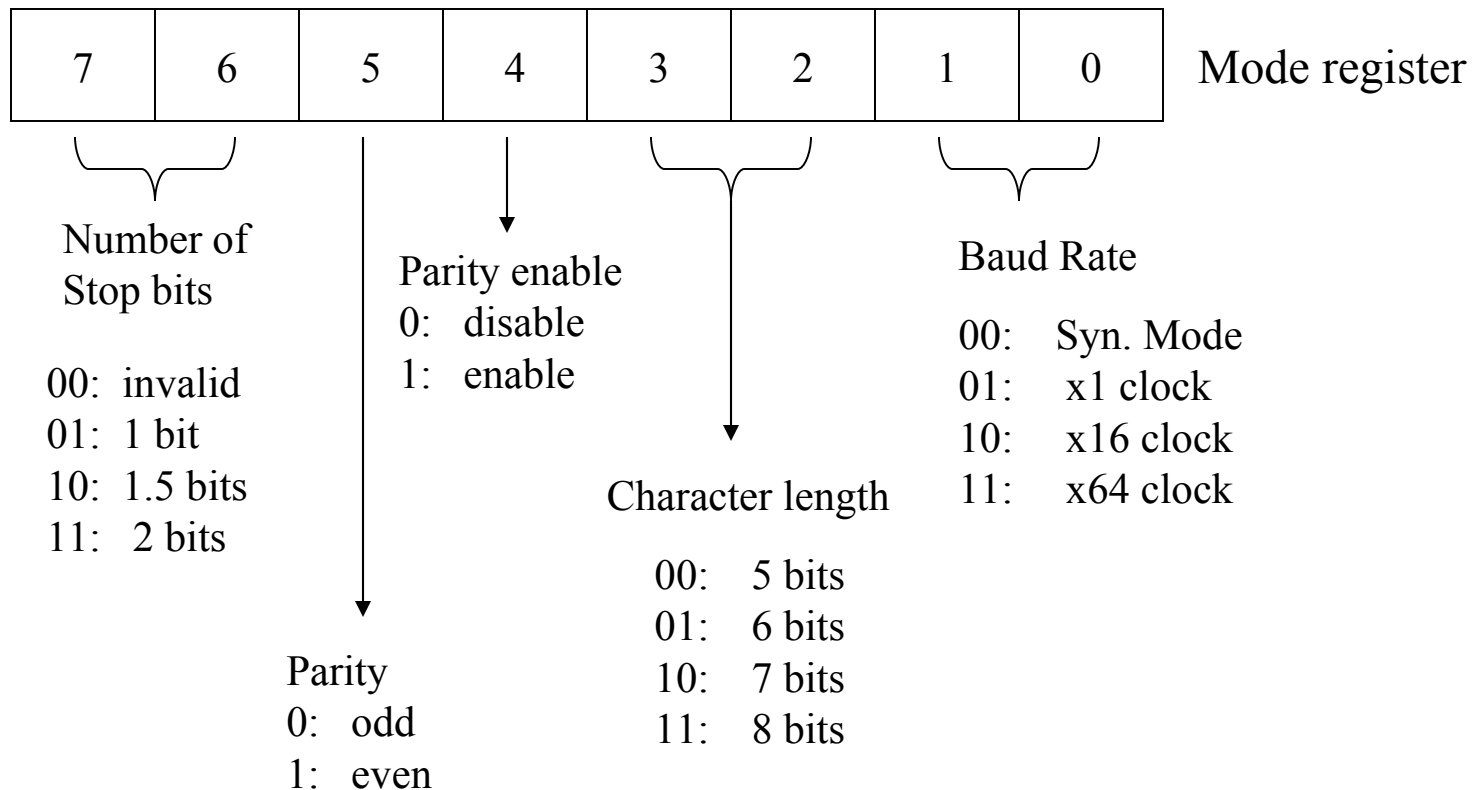
Baud (Baud is # of bits transmitted/sec, including start, stop, data and parity).

8251 USART Interface



Programming 8251

❑ 8251 mode register



Programming 8251

❑ 8251 command register

EH	IR	RTS	ER	SBRK	RxE	DTR	TxE	command register
----	----	-----	----	------	-----	-----	-----	------------------

TxE: transmit enable
DTR: data terminal ready
RxE: receiver enable
SBPRK: send break character
ER: error reset
RTS: request to send
IR: internal reset
EH: enter hunt mode

Programming 8251

❑ 8251 status register

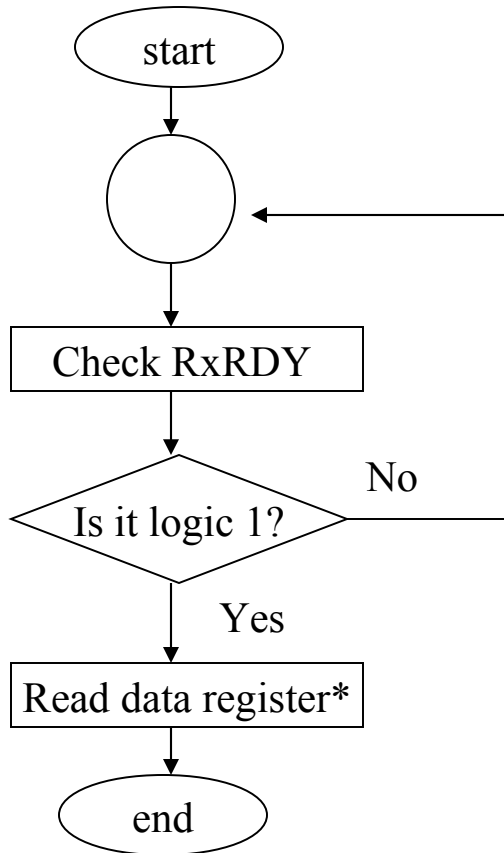
DSR	SYNDET	FE	OE	PE	TxEMPTY	RxRDY	TxRDY
-----	--------	----	----	----	---------	-------	-------

status register

TxRDY:	transmit ready
RxRDY:	receiver ready
TxEMPTY:	transmitter empty
PE:	parity error
OE:	overflow error
FE:	framing error
SYNDET:	sync. character detected
DSR:	data set ready

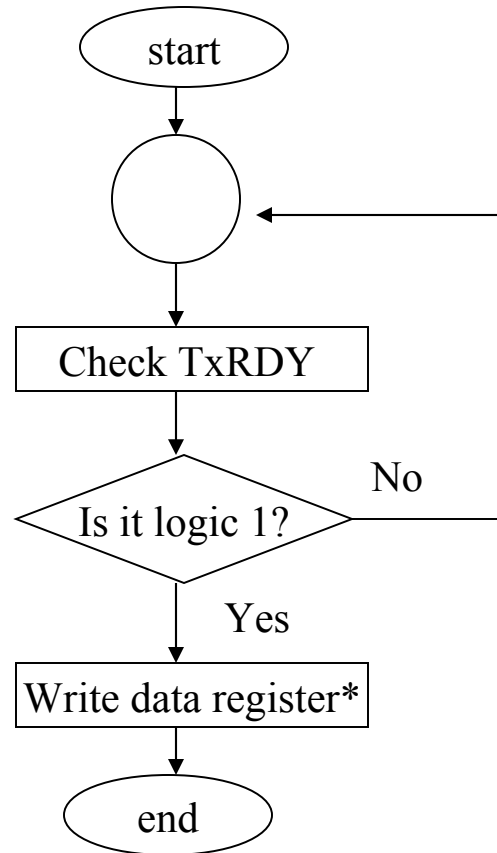
Simple Serial I/O Procedures

❑ Read



* This clears RxRDY

❑ Write

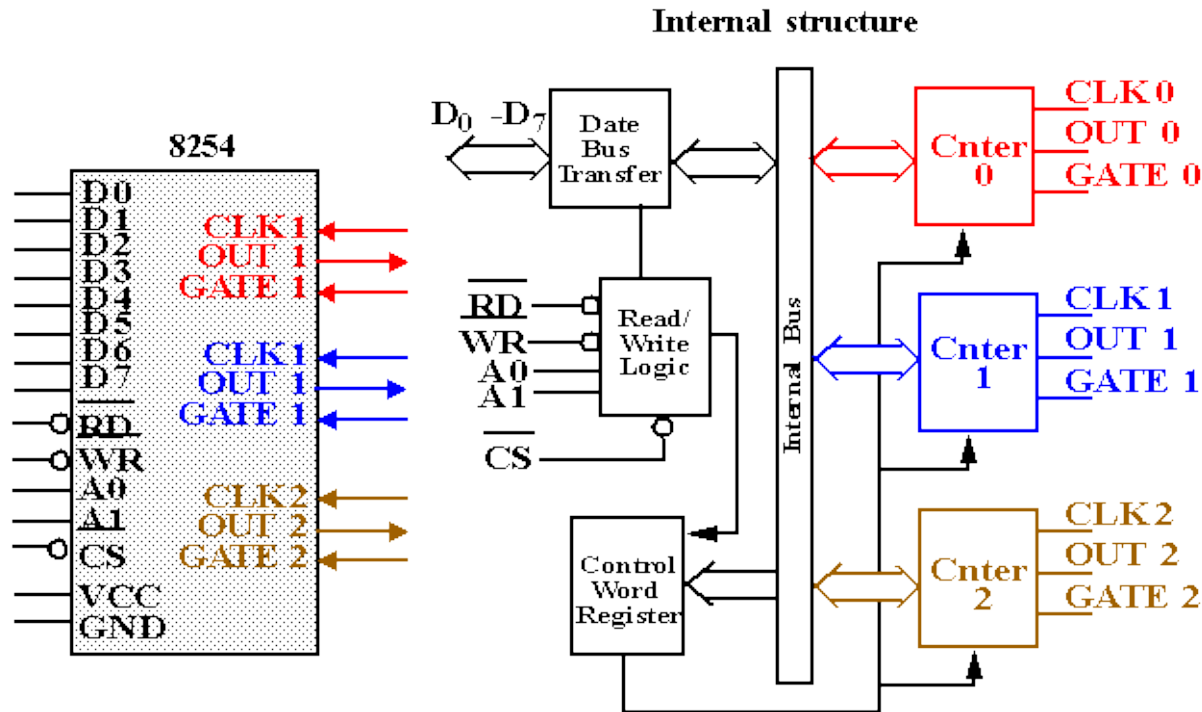


* This clears TxRDY

Errors

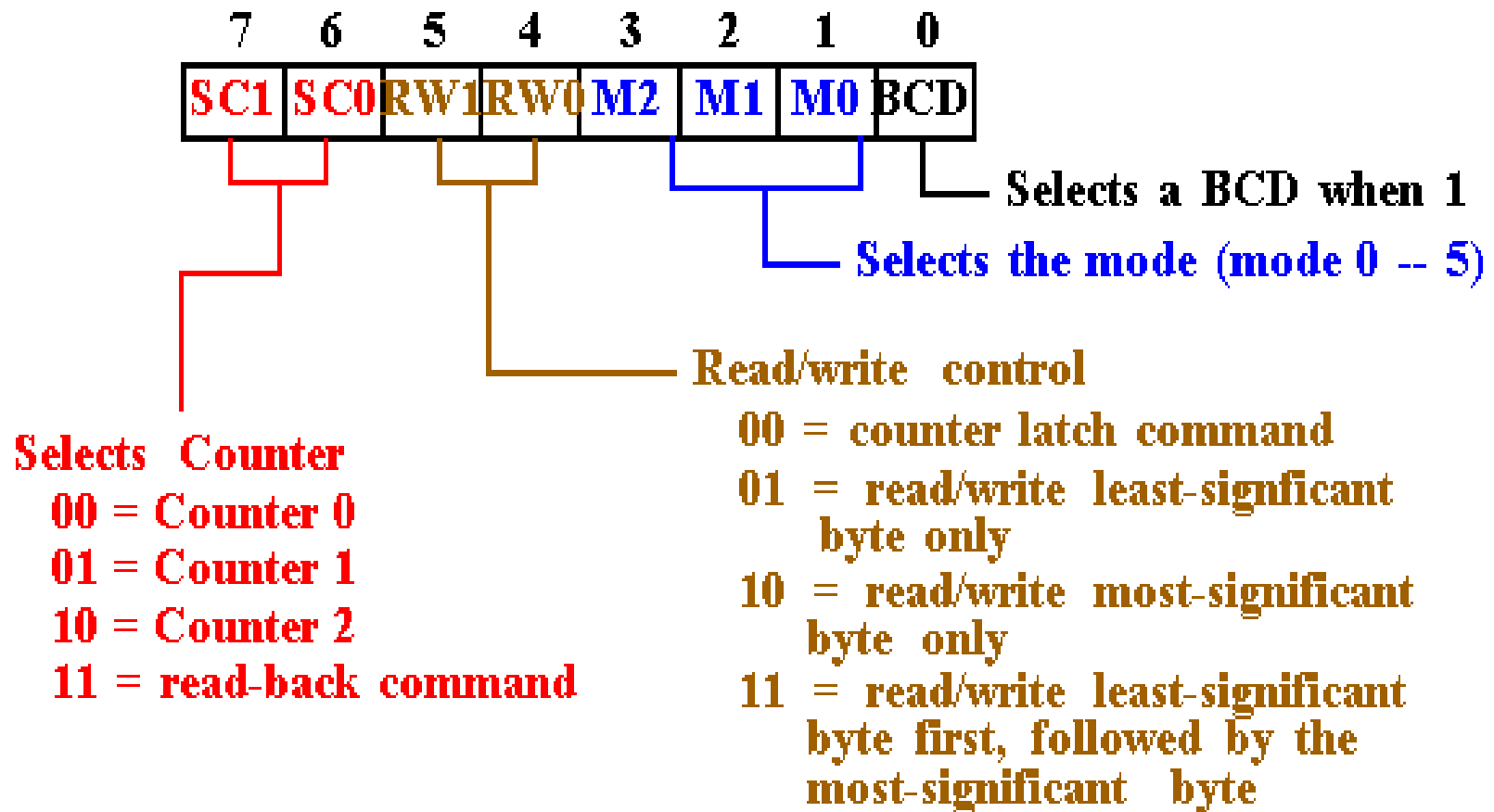
- Parity error: Received data has wrong error -- transmission bit flip due to noise.
- Framing error: Start and stop bits not in their proper places.
 - This usually results if the receiver is receiving data at the incorrect baud rate.
- Overrun error: Data has overrun the internal receiver FIFO buffer.
 - Software is failing to read the data from the FIFO.

Programmable Timer 8254



A ₁	A ₀	Function
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word

8254 Programming

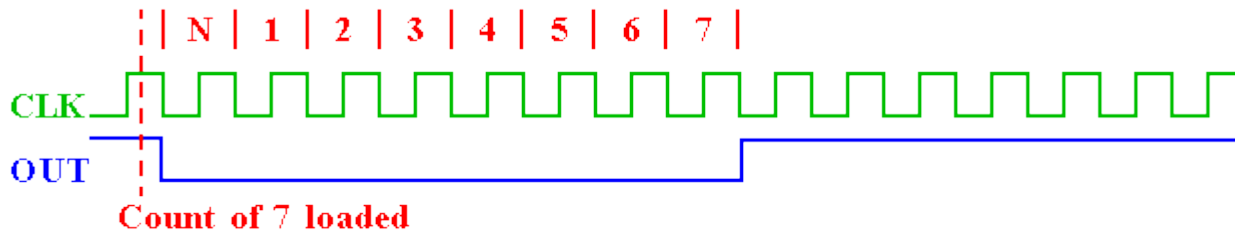


8254 Programming

- Each counter may be programmed with a count of 1 to FFFFH.
 - Minimum count is 1 all modes except 2 and 3 with minimum count of 2.
- Each counter has a program control word used to select the way the counter operates.
 - If two bytes are programmed, then the first byte (LSB) stops the count, and the second byte (MSB) starts the counter with the new count.

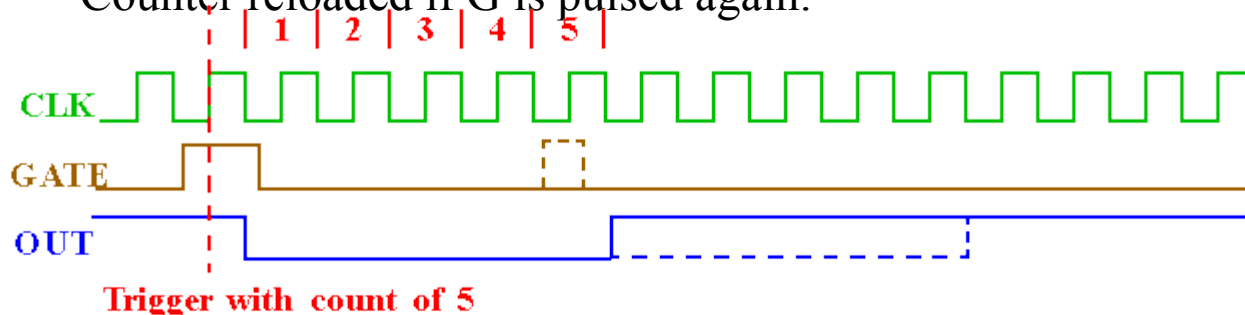
8254 Modes

- Mode 0: An events counter enabled with G.
 - The output becomes a logic 0 when the control word is written and remains there until N plus the number of programmed counts.



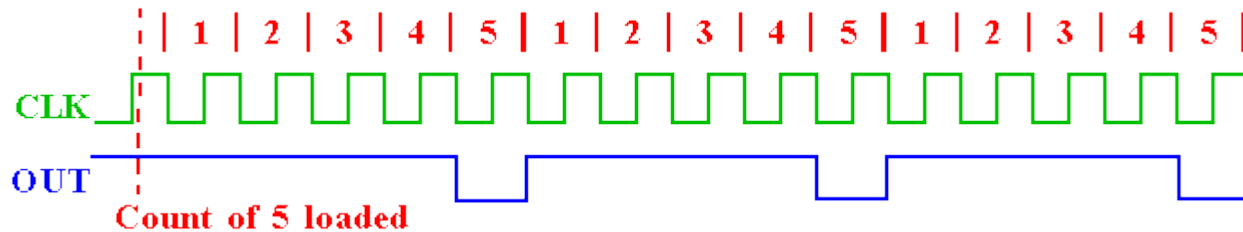
Mode 1: One-shot mode.

- The G input triggers the counter to output a 0 pulse for 'count' clocks.
- Counter reloaded if G is pulsed again.

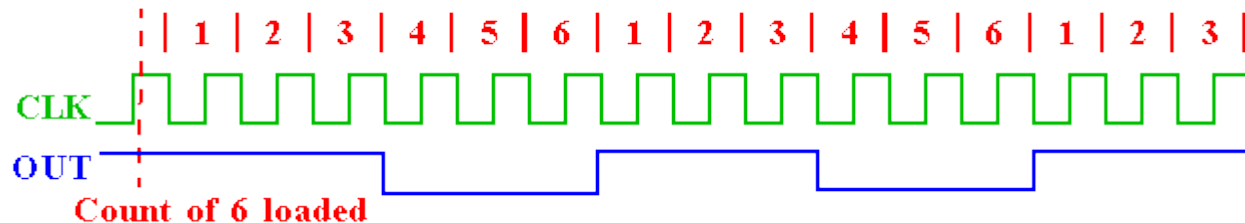


8254 Modes

- Mode 2: Counter generates a series of pulses 1 clock pulse wide.
 - The separation between pulses is determined by the count.
 - The cycle is repeated until reprogrammed or G pin set to 0.



- Mode 3: Generates a continuous square-wave with G set to 1.
 - If count is even, 50% duty cycle otherwise OUT is high 1 cycle longer.



8254 Modes

