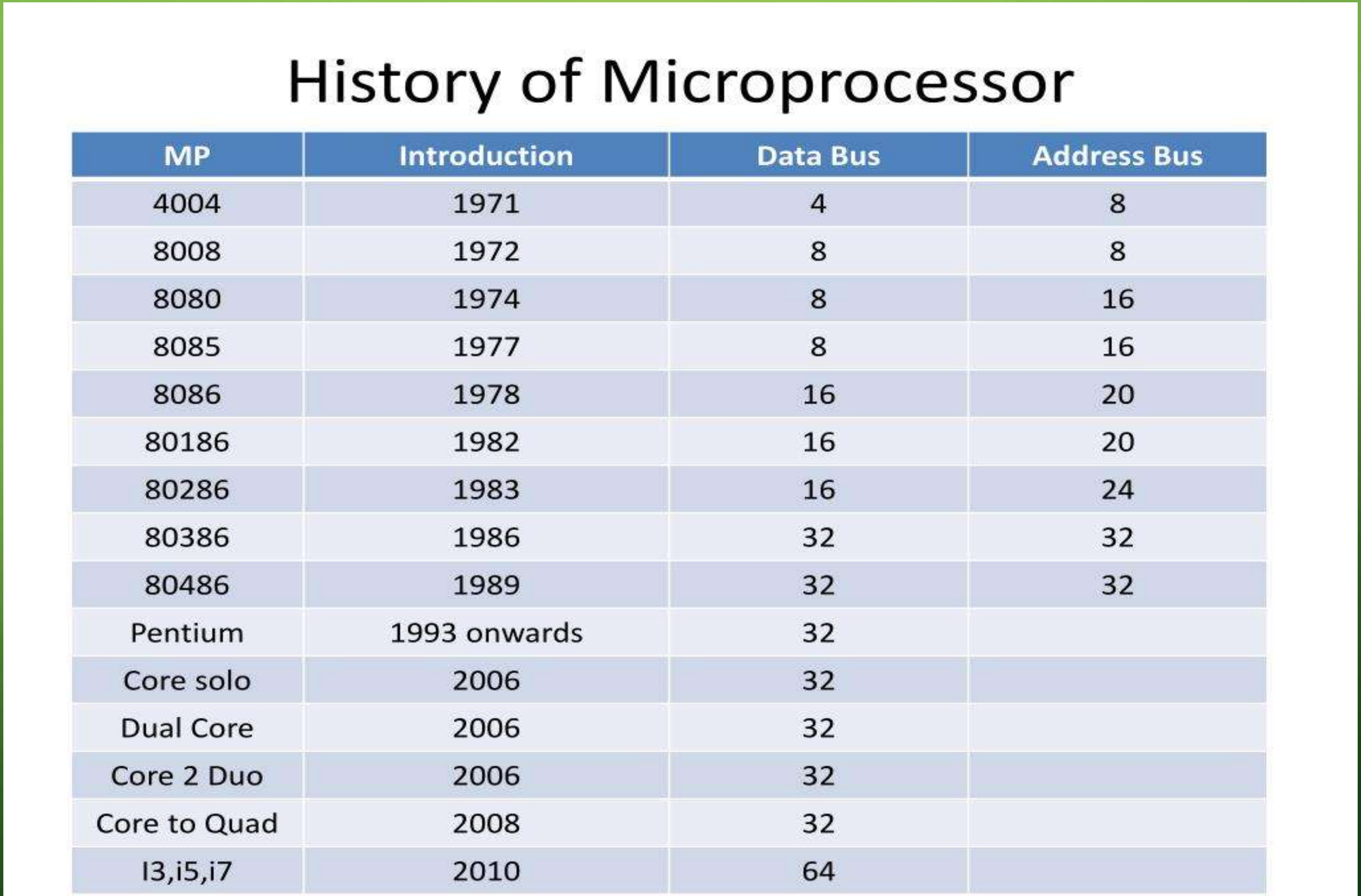
INTRODUCTION TO MICROPROCESSOR

UNIT-1

M. A. HIMAYATH SHAMSHI (ASSOC.PROF)

DEPARTMENT OF ECE

VAAGDEVI COLLEGE OF ENGINEERING



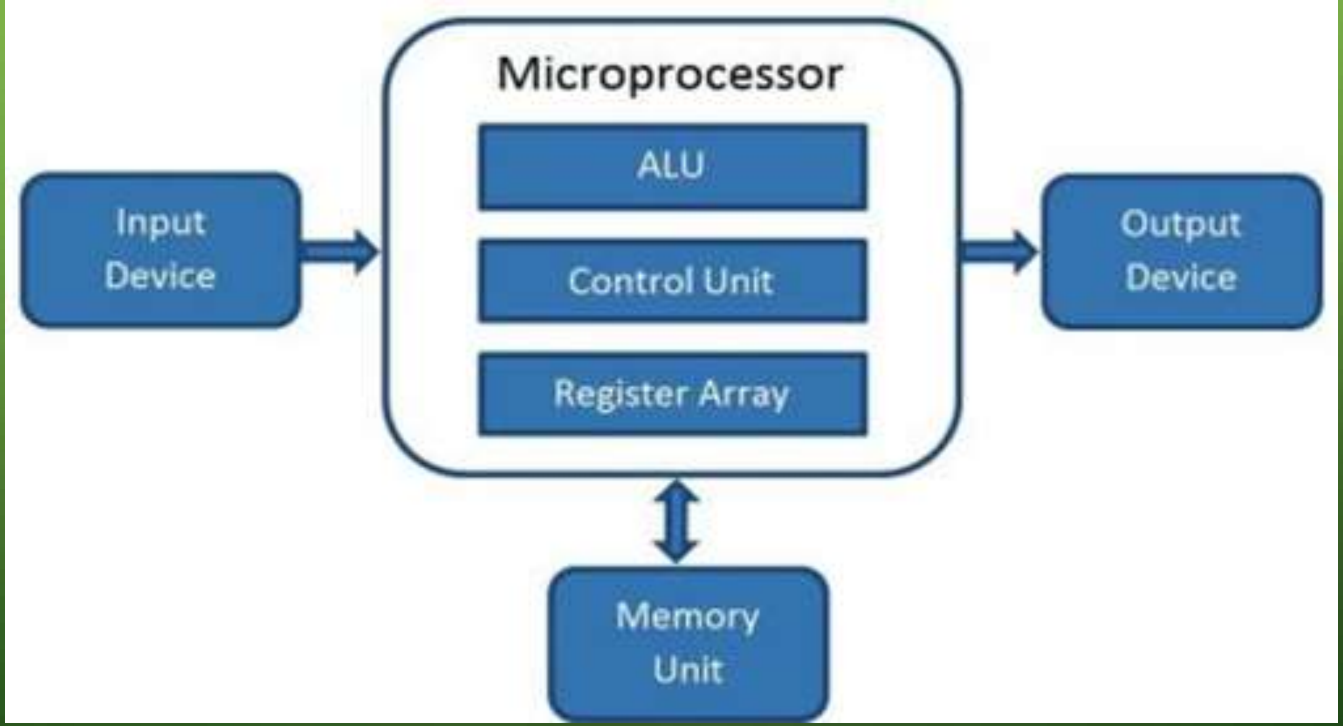
History of Microprocessor

MP	Introduction	Data Bus	Address Bus
4004	1971	4	8
8008	1972	8	8
8080	1974	8	16
8085	1977	8	16
8086	1978	16	20
80186	1982	16	20
80286	1983	16	24
80386	1986	32	32
80486	1989	32	32
Pentium	1993 onwards	32	
Core solo	2006	32	
Dual Core	2006	32	
Core 2 Duo	2006	32	
Core to Quad	2008	32	
I3,i5,i7	2010	64	

Microprocessor - Overveiw

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.

Block Diagram Of Basic Micro Computer



How does a Microprocessor Work?

- The microprocessor follows a sequence: Fetch, Decode, and then Execute.
- Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

List of Terms Used in a Microprocessor

Instruction Set − It is the set of instructions that the microprocessor can understand.

Bandwidth − It is the number of bits processed in a single instruction.

Clock Speed − It determines the number of operations per second the processor can perform. It is expressed in megahertz (MHz) or gigahertz (GHz).It is also known as Clock Rate.

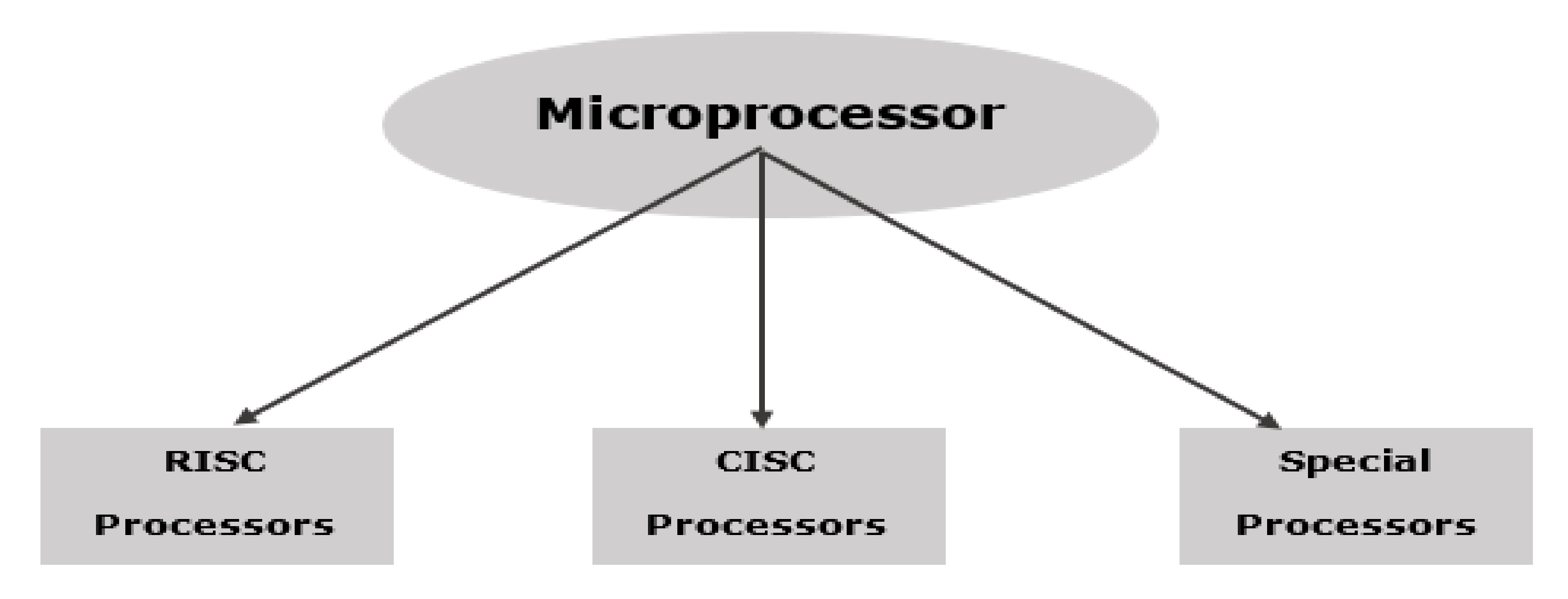
Word Length − It depends upon the width of internal data bus, registers, ALU, etc. An 8-bit microprocessor can process 8-bit data at a time. The word length ranges from 4 bits to 64 bits depending upon the type of the microcomputer.

Data Types − The microprocessor has multiple data type formats like binary, BCD, ASCII, signed and unsigned numbers.

Features of a Microprocessor

- **Cost-effective** − The microprocessor chips are available at low prices and results its low cost.
- **Size** − The microprocessor is of small size chip, hence is portable.
- **Low Power Consumption** − Microprocessors are manufactured by using metal oxide semiconductor technology, which has low power consumption.
- **Versatility** − The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
- **Reliability** − The failure rate of an IC in microprocessors is very low, hence it is reliable.

Microprocessor - Classification



Characteristics of RISC

The major characteristics of a RISC processor are as follows −

- It consists of simple instructions.
- It supports various data-type formats.
- It utilizes simple addressing modes and fixed length instructions for pipelining.
- It supports register to use in any context.
- One cycle execution time.
- “LOAD” and “STORE” instructions are used to access the memory location.
- It consists of larger number of registers.
- It consists of less number of transistors.

Characteristics of CISC

- Variety of addressing modes.
- Larger number of instructions.
- Variable length of instruction formats.
- Several cycles may be required to execute one instruction.
- Instruction-decoding logic is complex.
- One instruction is required to support multiple addressing modes.

Special Processors

These are the processors which are designed for some special purposes. Few of the special processors are briefly discussed −

Coprocessor

A coprocessor is a specially designed microprocessor, which can handle its particular function many times faster than the ordinary microprocessor.

For example – Math Coprocessor.

Some Intel math-coprocessors are –

- 8087-used with 8086
- 80287-used with 80286
- 80387-used with 80386

Input/Output Processor

It is a specially designed microprocessor having a local memory of its own, which is used to control I/O devices with minimum CPU involvement.

For example –

- DMA (direct Memory Access) controller
- Keyboard/mouse controller
- Graphic display controller
- SCSI port controller

Transputer (Transistor Computer)

A transputer is a specially designed microprocessor with its own local memory and having links to connect one transputer to another transputer for inter-processor communications. It was first designed in 1980 by In mos and is targeted to the utilization of VLSI technology.

A transputer can be used as a single processor system or can be connected to external links, which reduces the construction cost and increases the performance

For example – 16-bit T212, 32-bit T425, the floating point (T800, T805 & T9000) processors.

DSP (Digital Signal Processor)

This processor is specially designed to process the analog signals into a digital form. This is done by sampling the voltage level at regular time intervals and converting the voltage at that instant into a digital form. This process is performed by a circuit called an analogue to digital converter, A to D converter or ADC.

A DSP contains the following components –

- **Program Memory** – It stores the programs that DSP will use to process data.
- **Data Memory** – It stores the information to be processed.
- **Compute Engine** – It performs the mathematical processing, accessing the program from the program memory and the data from the data memory.

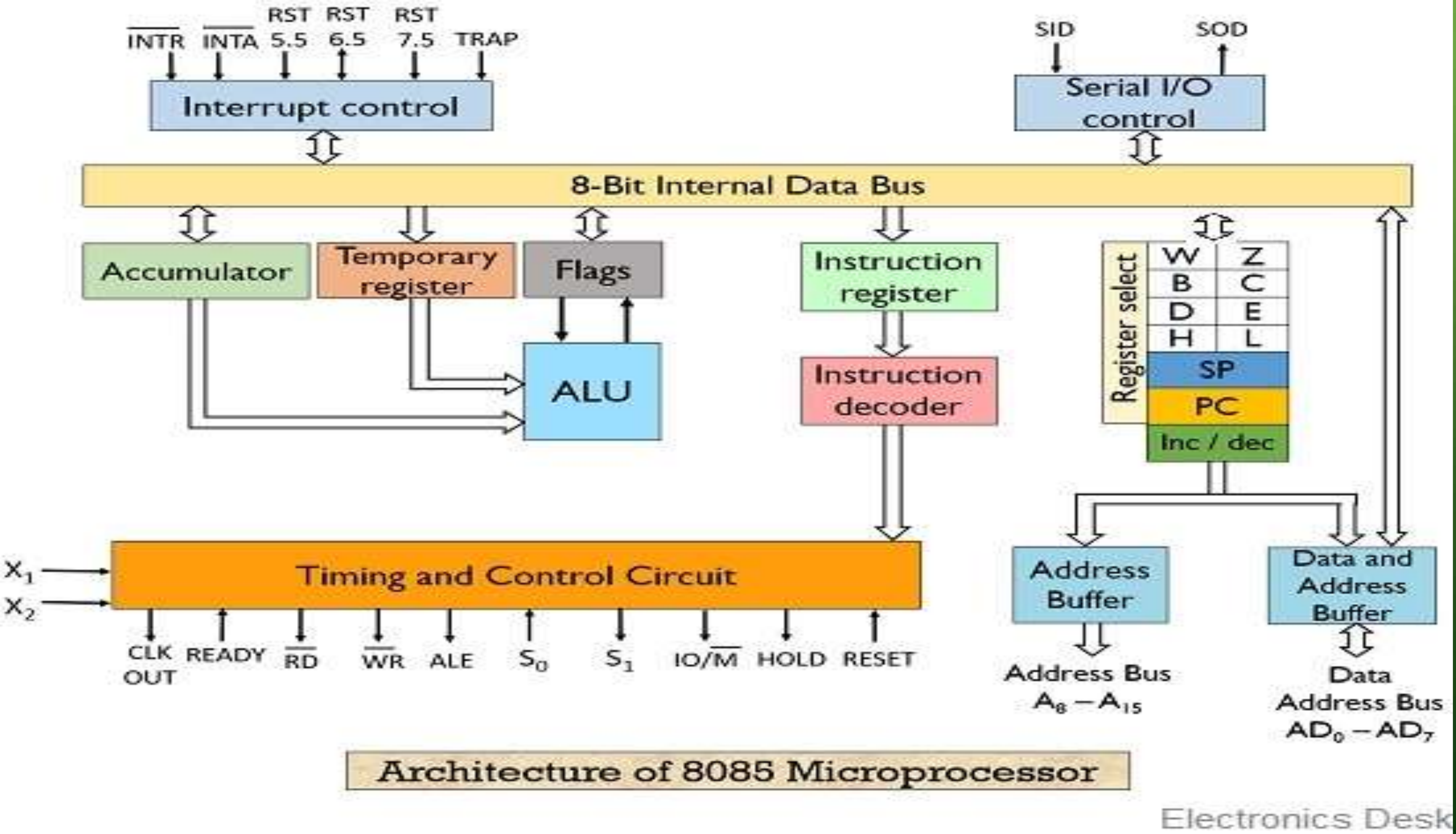
Input/Output – It connects to the outside world.

Its applications are –

- Sound and music synthesis,
- Audio and video compression
- Video signal processing
- 2D and 3d graphics acceleration.

For example – Texas Instrument’s TMS 320 series, e.g., TMS 320C40, TMS320C50.

Overview Of Microprocessor - 8085 Architecture

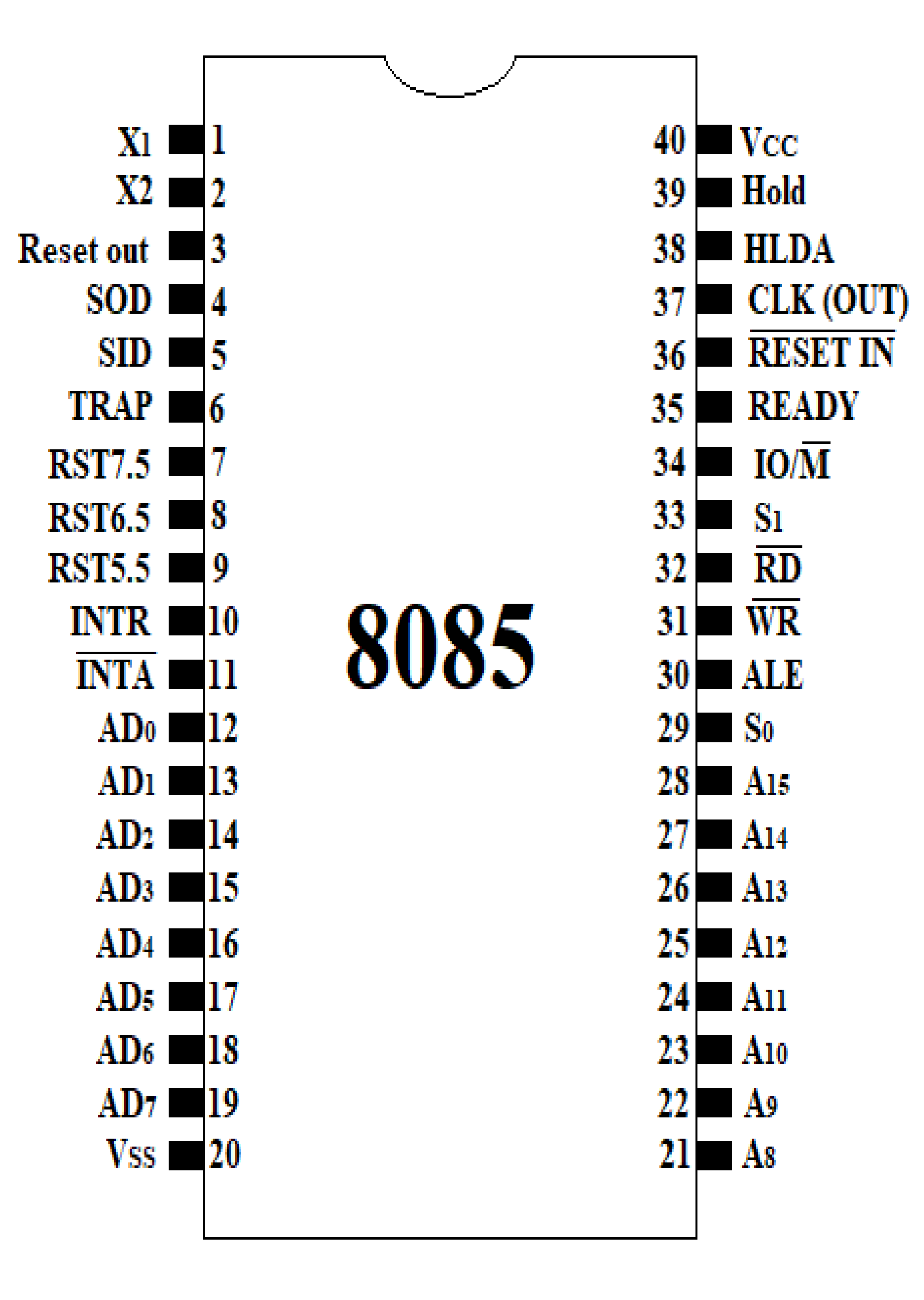


Architecture of 8085 Microprocessor

8085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8-bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration −

- 8-bit data bus
- 16-bit address bus, which can address up to 64KB
- A 16-bit program counter
- A 16-bit stack pointer
- Six 8-bit registers arranged in pairs: BC, DE, HL
- Requires +5V supply to operate at 3.2 MHZ single phase clock
- It is used in washing machines, microwave ovens mobile phones, etc.



Microprocessor - 8086 Overview

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

•It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.

•It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.

•It is available in 3 versions based on the frequency of operation −

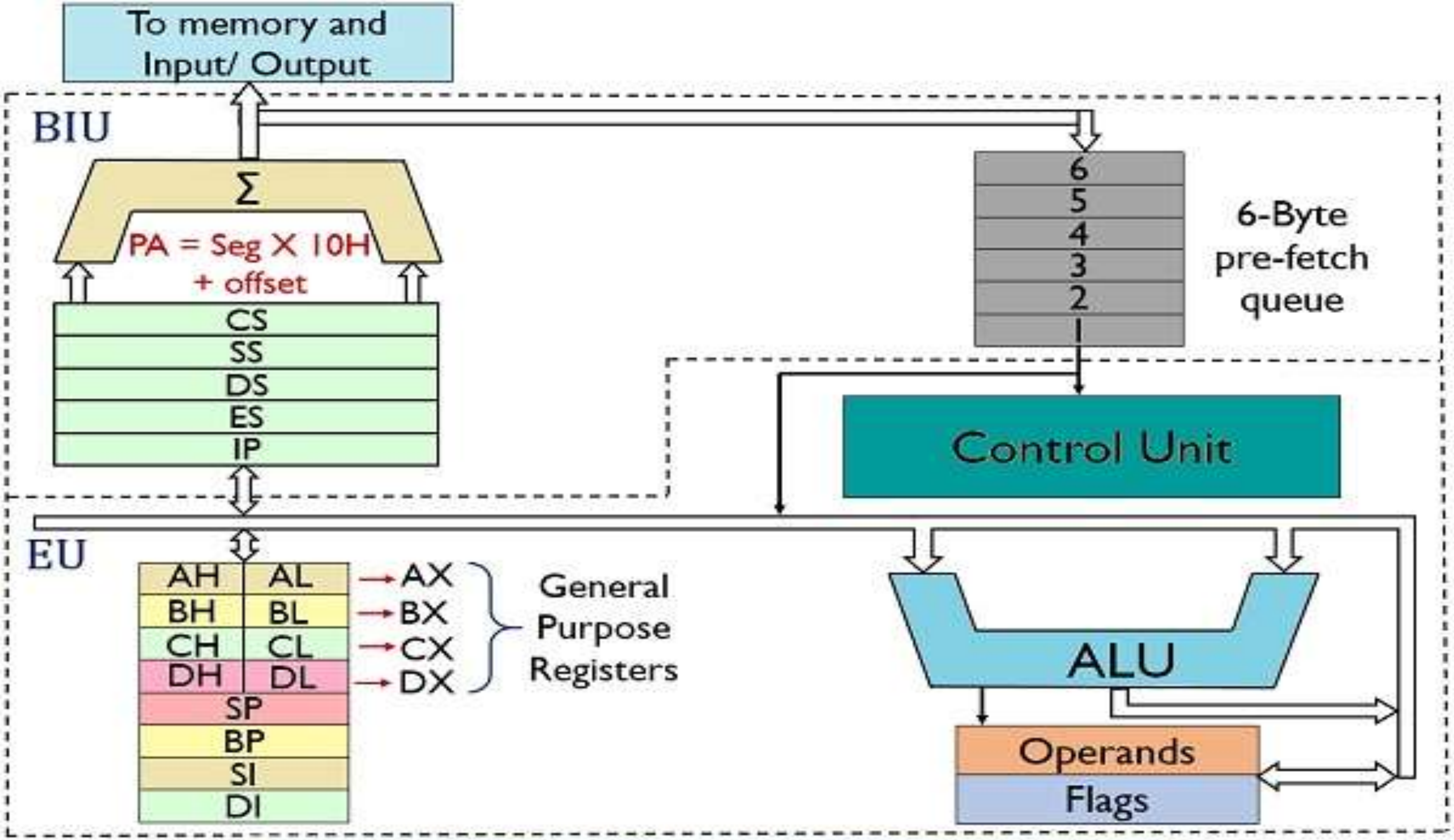
- 8086 → 5MHz
- 8086-2 → 8MHz
- (c)8086-1 → 10 MHz

•It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.

•Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.

•Execute stage executes these instructions , It has 256 vectored interrupts.

•It consists of 29,000 transistors.



Block Diagram of 8086 Microprocessor

Electronics Desk

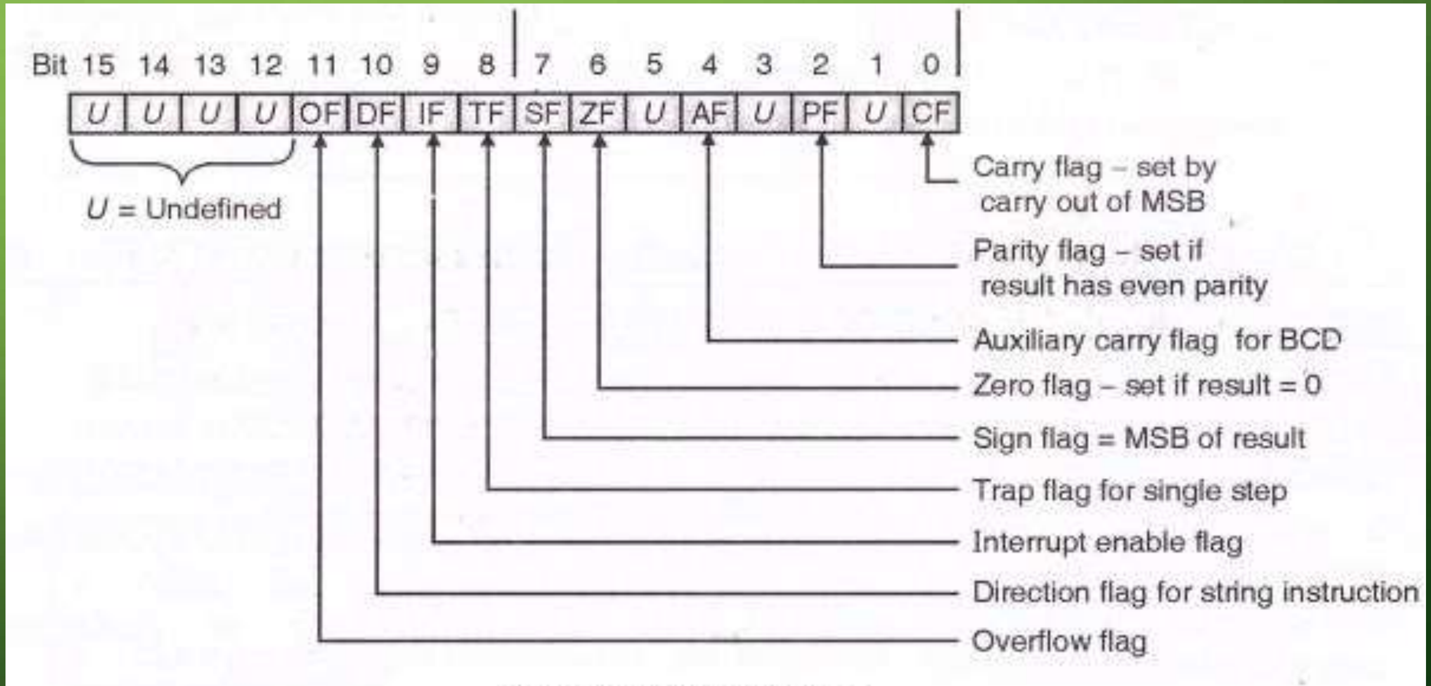
- **EU (Execution Unit)**

Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

- **BIU (Bus Interface Unit)**

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory. EU has no direction connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus.

- **<u>Flag Register of 8086</u>**



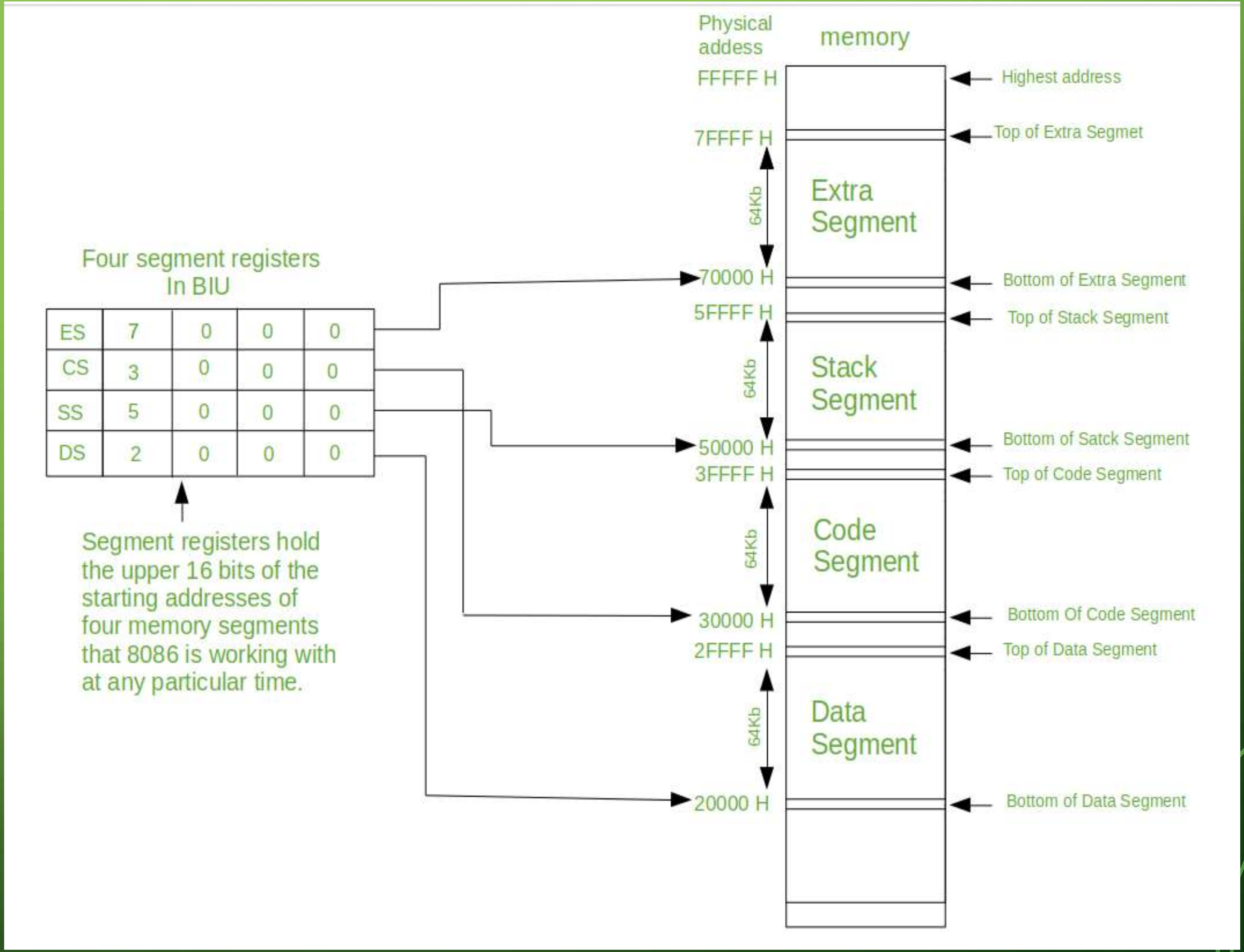
Bit 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

U = Undefined
Carry flag – set by carry out of MSB
Parity flag – set if result has even parity
Auxiliary carry flag for BCD
Zero flag – set if result = 0
Sign flag = MSB of result
Trap flag for single step
Interrupt enable flag
Direction flag for string instruction
Overflow flag

8086 flag register format

Microprocessor – 8086Memory Organization

- Operating frequency of 8086 is 5MHZ
- Memory Capacity is 1MB
- Address Bus Capacity is 20 bits
- Data Bus Capacity is 16 bits
- Operating Voltage is 5V to 12V



Four segment registers In BIU

ES	7	0	0	0
CS	3	0	0	0
SS	5	0	0	0
DS	2	0	0	0

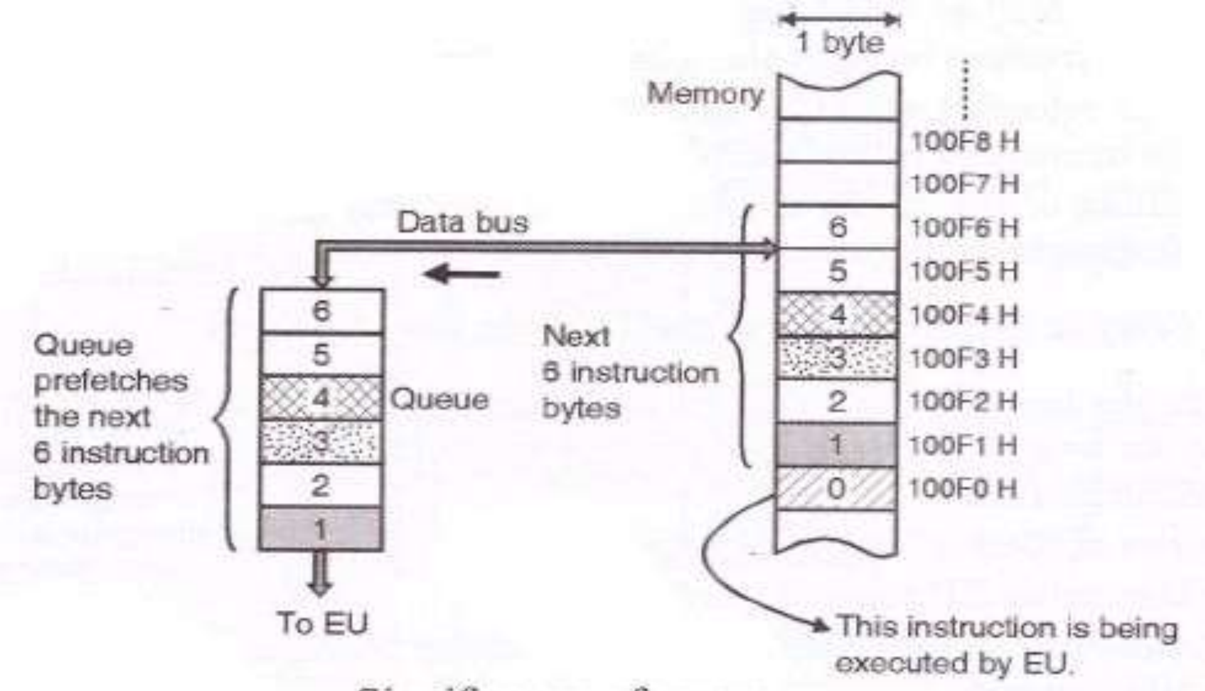
Segment registers hold the upper 16 bits of the starting addresses of four memory segments that 8086 is working with at any particular time.

Advantages of the Segmentation The main advantages of segmentation are as follows:

- It provides a powerful memory management mechanism.
- Data related or stack related operations can be performed in different segments.
- Code related operation can be done in separate code segments.
- It allows to processes to easily share data.
- It allows to extend the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 Megabytes. Without segmentation, it would require 20 bit registers.
- It is possible to enhance the memory size of code data or stack segments beyond 64 KB by allotting more than one segment for each area.

Instruction QUEUE

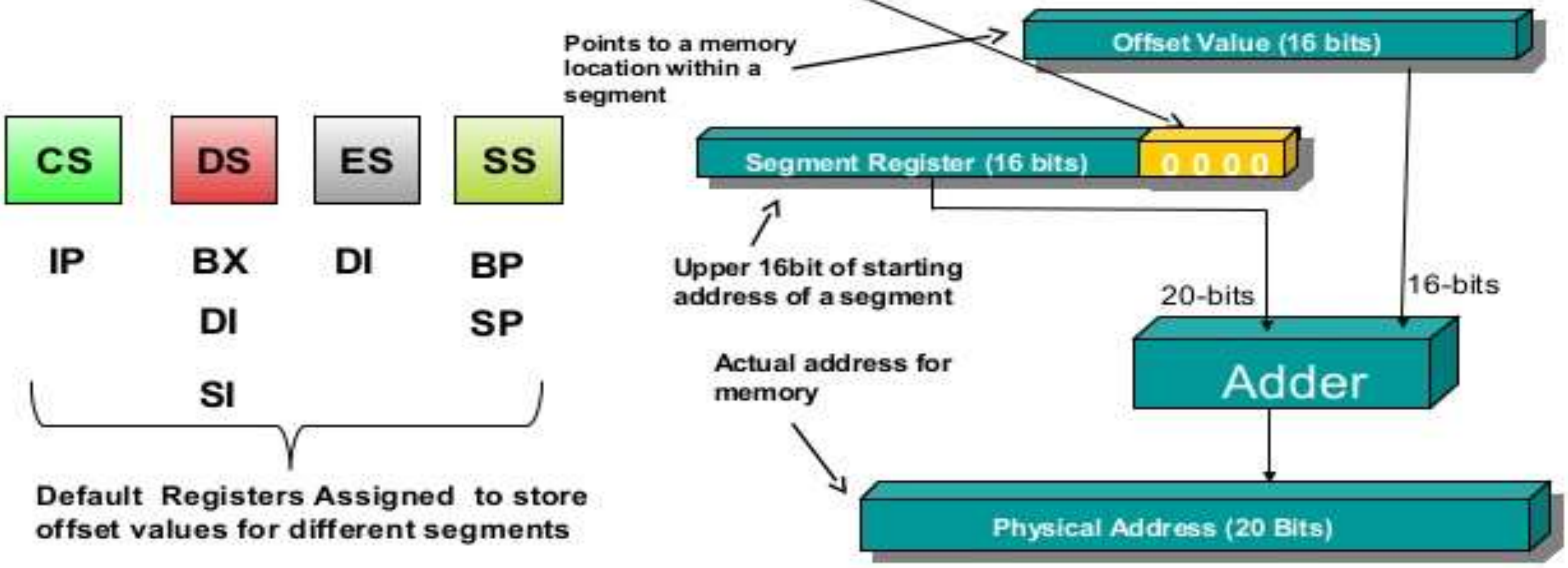
BIU contains the instruction queue. BIU gets upto 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.



Significance of queue

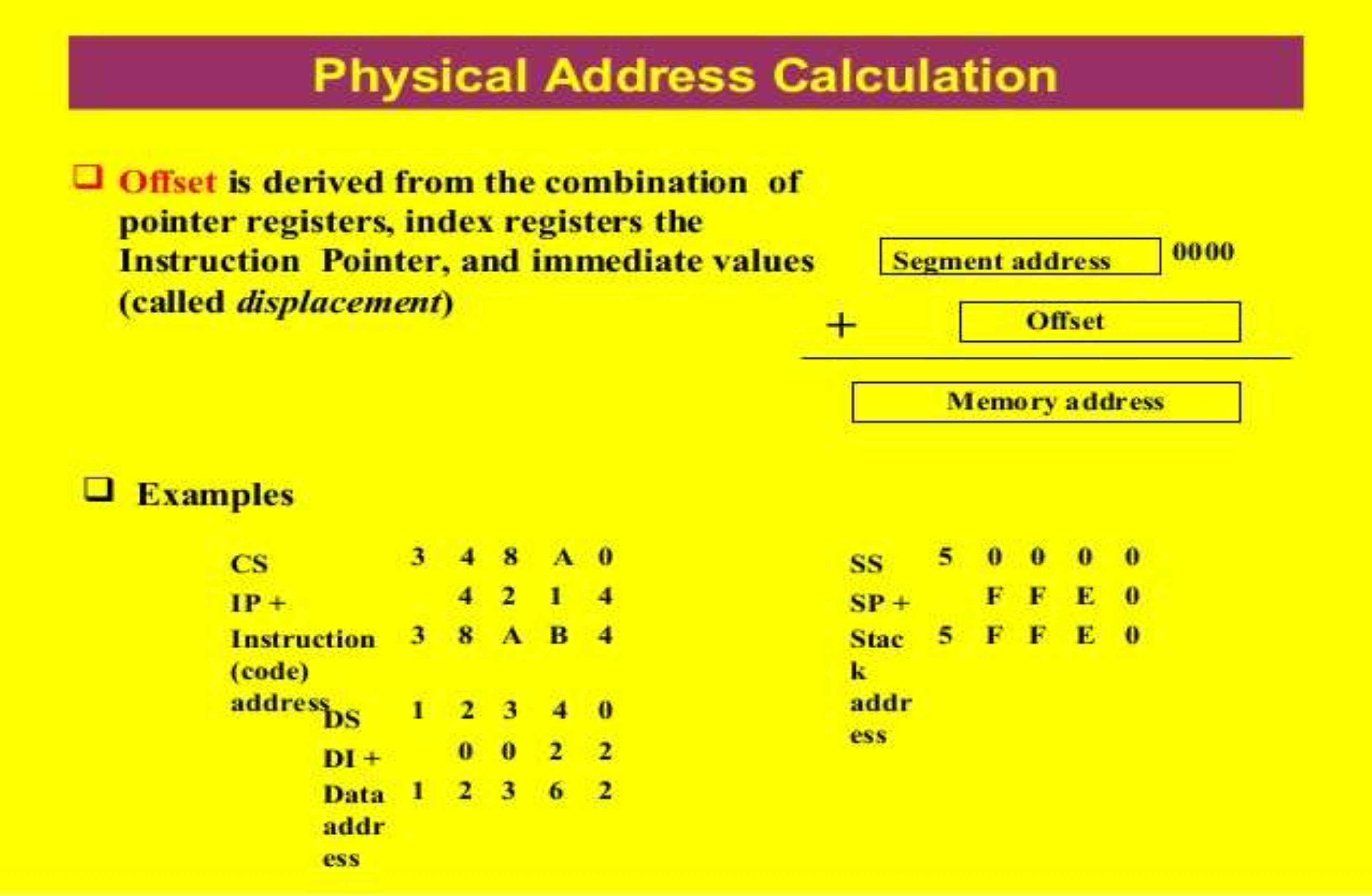
Physical Address Generation in 8086

- The 20-bit physical address is generated by adding 16-bit contents of a **segment register** with an 16-bit **offset value** (also called **Effective Address**) which is stored in a corresponding **default register** (either in IP, BX, SI, DI, BP or SP. Different segments have different default register for offset, for example IP is default offset register for Code Segment)
- BIU always appends **4 zeros** automatically to the 16-bit address of a segment register (to make it 20-bit) because it knows the starting address of a segment always ends with 4 zeros



Points to a memory location within a segment
Offset Value (16 bits)
CS
DS
ES
SS
IP
BX
DI
SI
DI
BP
SP
Segment Register (16 bits)
0 0 0 0
Upper 16bit of starting address of a segment
20-bits
16-bits
Adder
Actual address for memory
Physical Address (20 Bits)

Default Registers Assigned to store offset values for different segments



Physical Address Calculation

- Offset is derived from the combination of pointer registers, index registers the Instruction Pointer, and immediate values (called *displacement*)

Segment address 0000

+ Offset

Memory address

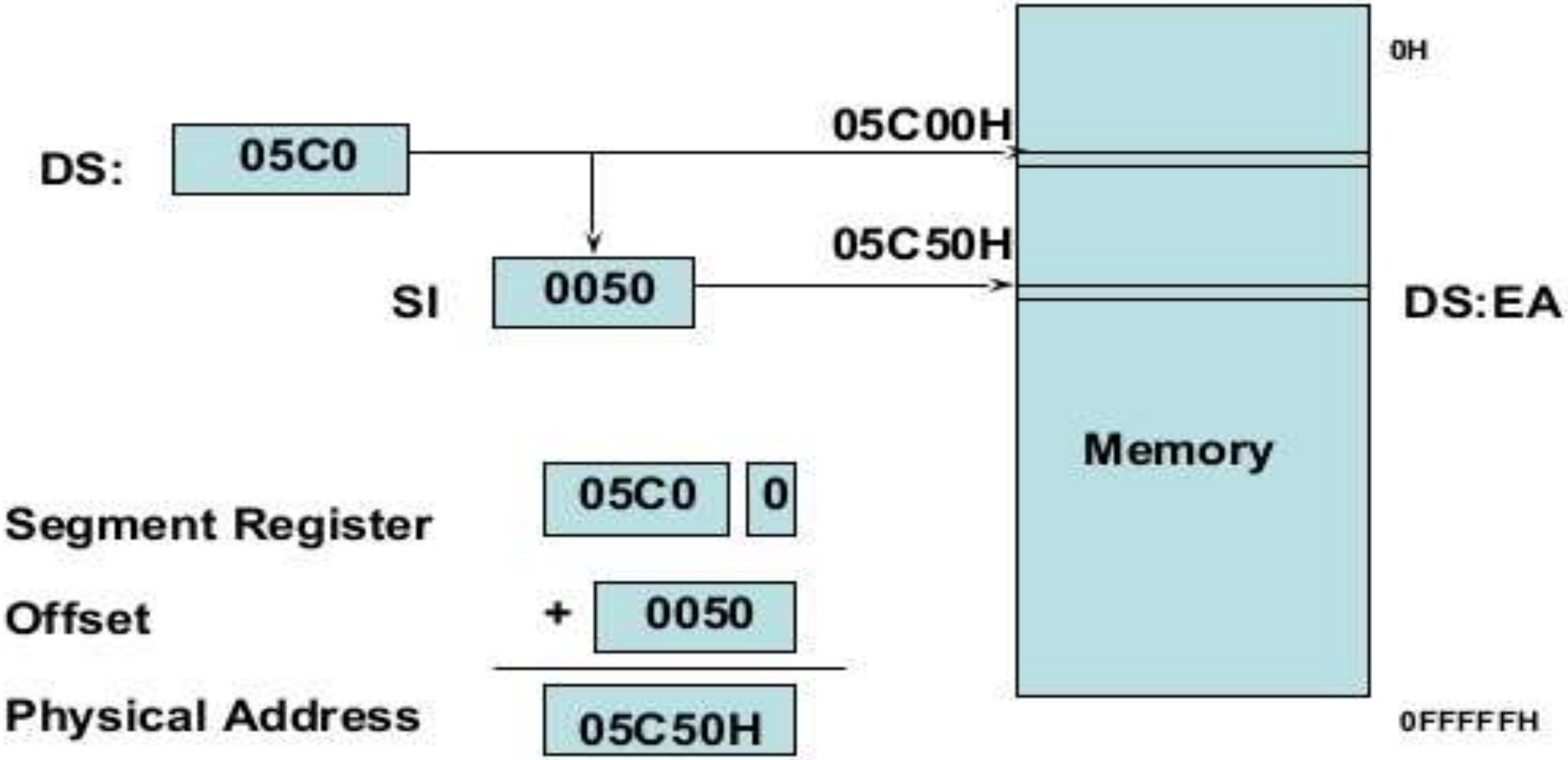
- Examples

CS	3	4	8	A	0
IP +		4	2	1	4
Instruction (code) address	3	8	A	B	4

DS	1	2	3	4	0
DI +		0	0	2	2
Data address	1	2	3	6	2

SS	5	0	0	0	0
SP +		F	F	E	0
Stack address	5	F	F	E	0

Example of Physical Address Generation for Data Segment

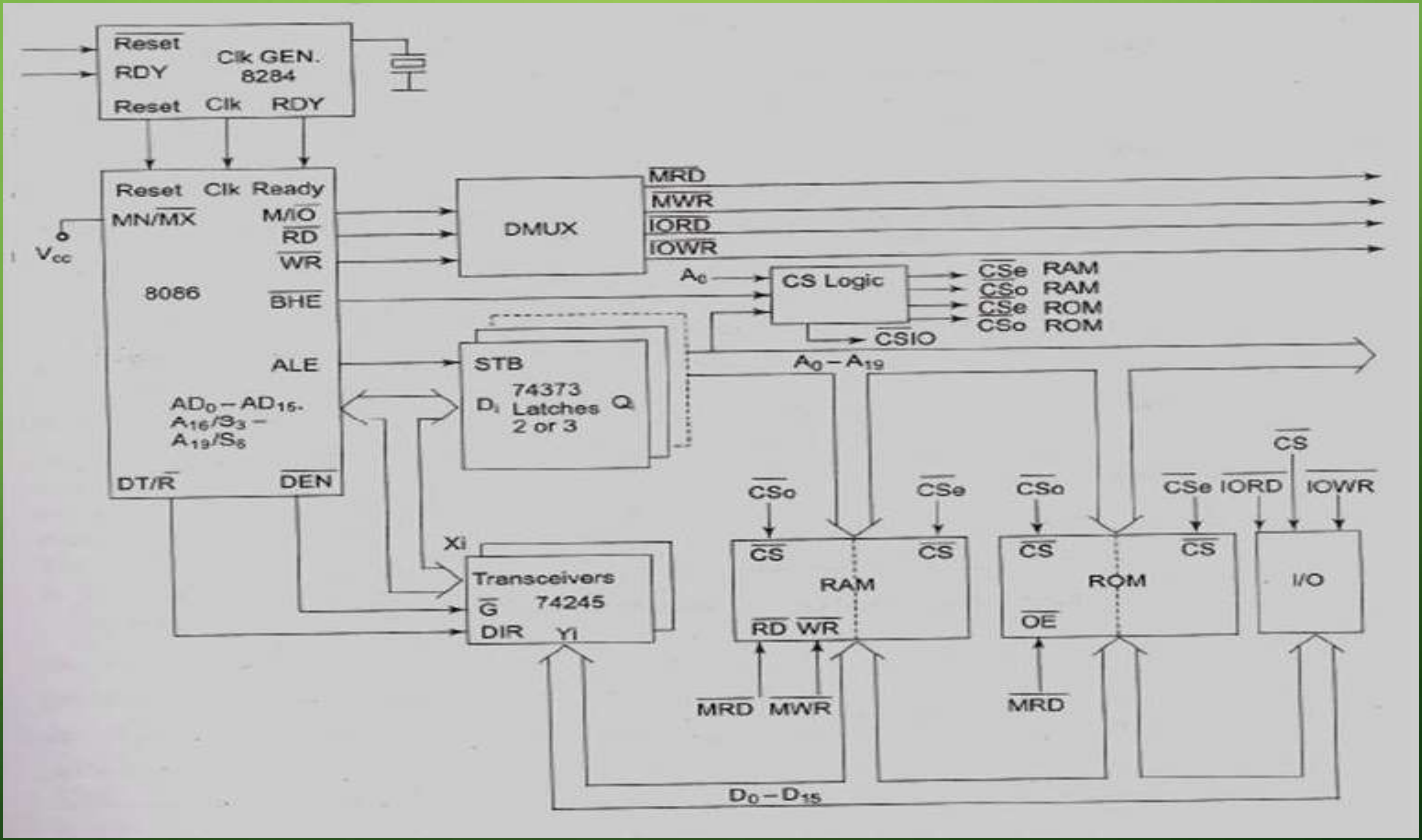


Data is fetched with respect to the DS register which contains starting or base address

The effective address (EA) or offset is in SI (default register for DS)

The EA depends on the addressing mode

Minimum Mode of 8086 Microprocessor



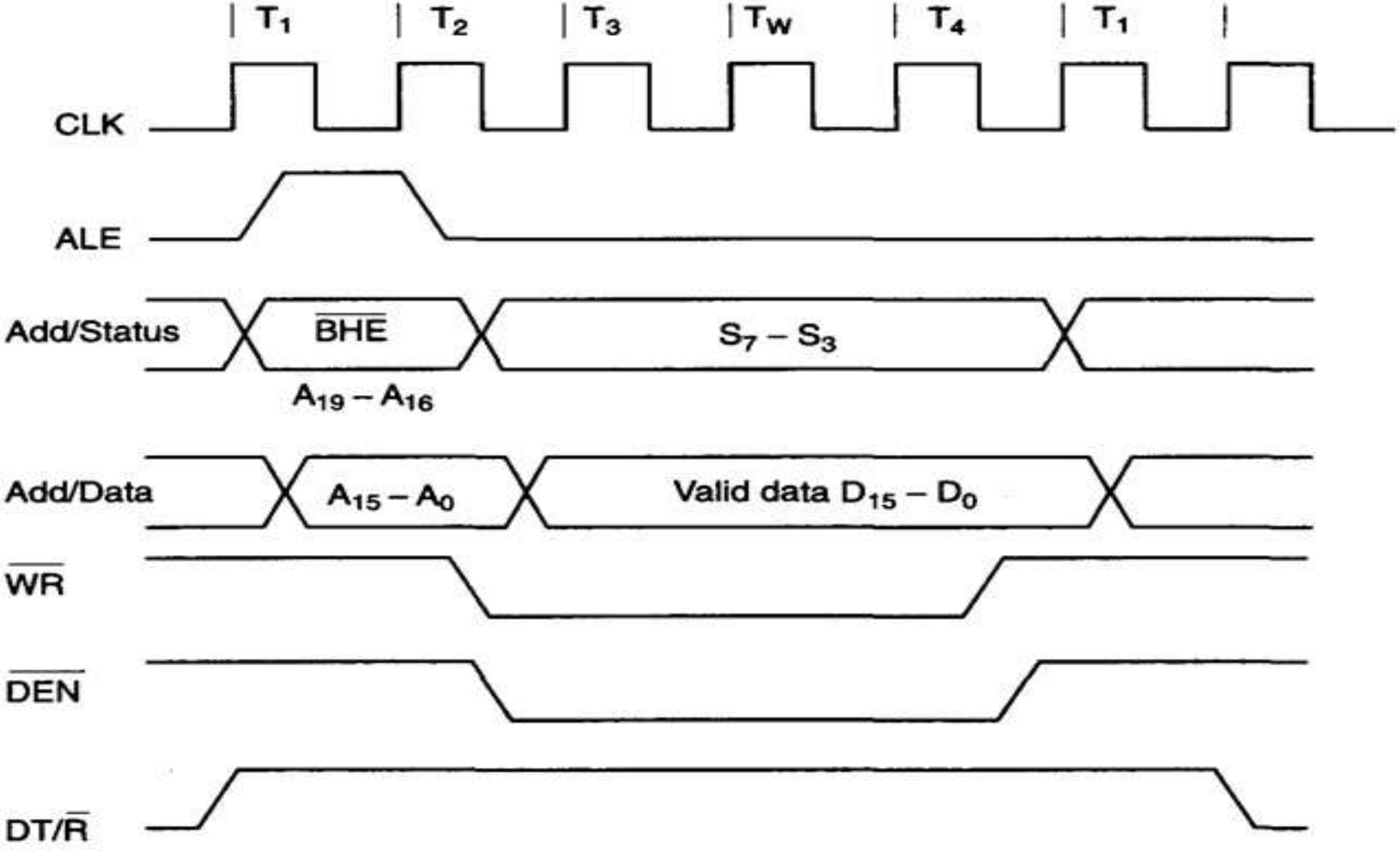
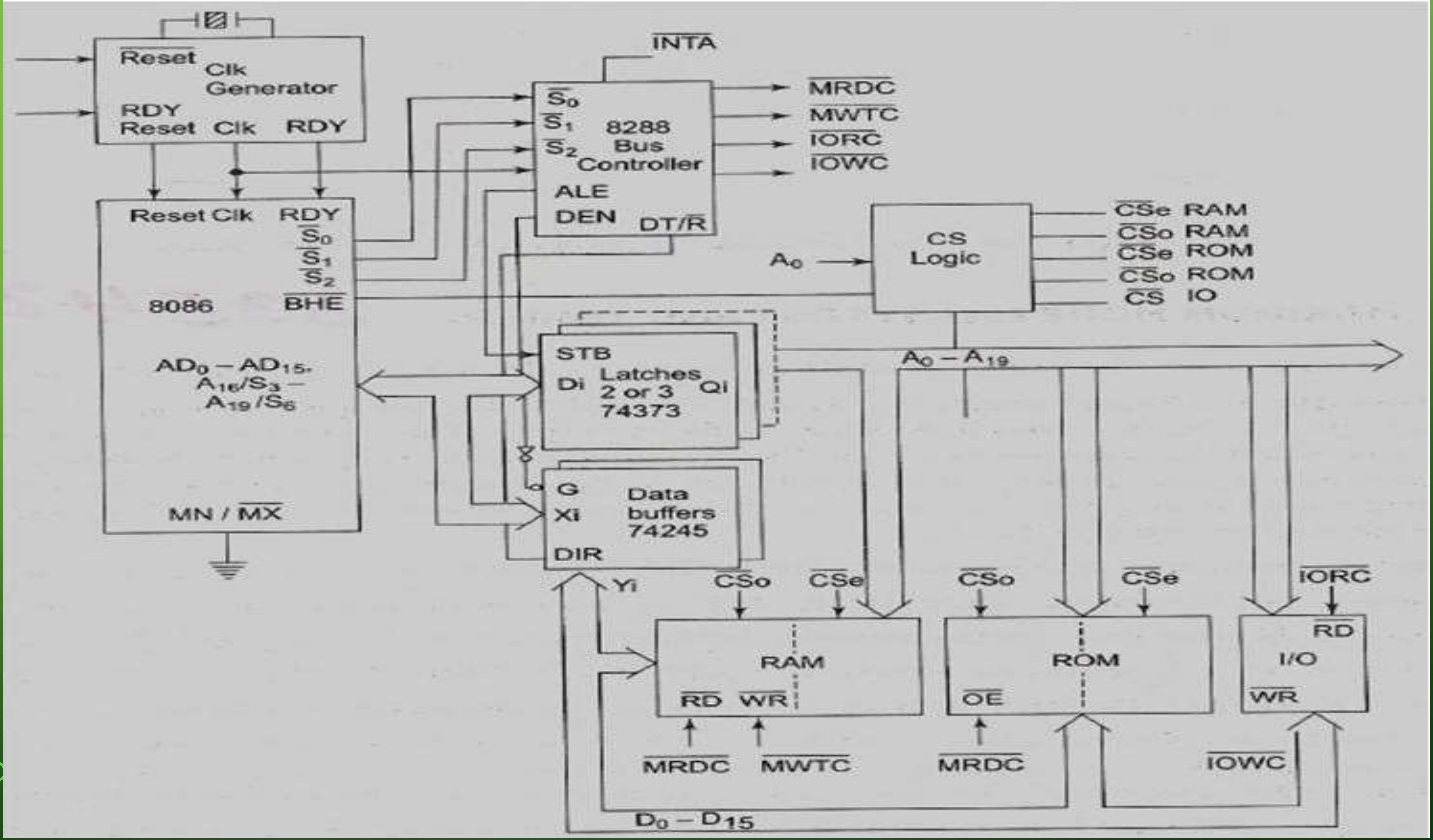
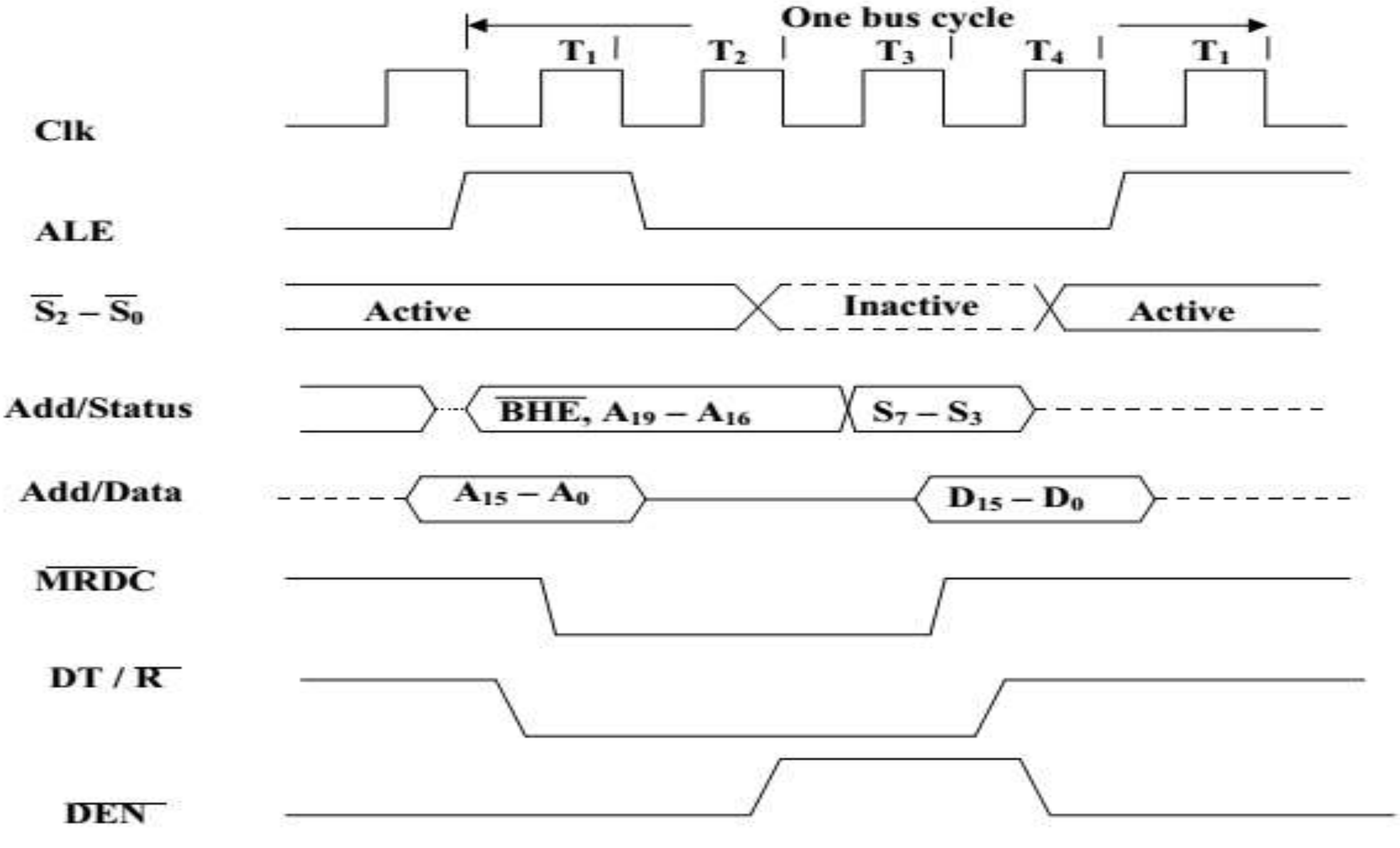


Fig. 1.9(b) *Write Cycle Timing Diagram for Minimum Operation*

Maximum Mode of 8086 Microprocessor





Memory Read Timing in Maximum Mode

Pin Diagram of 8086 Microprocessor

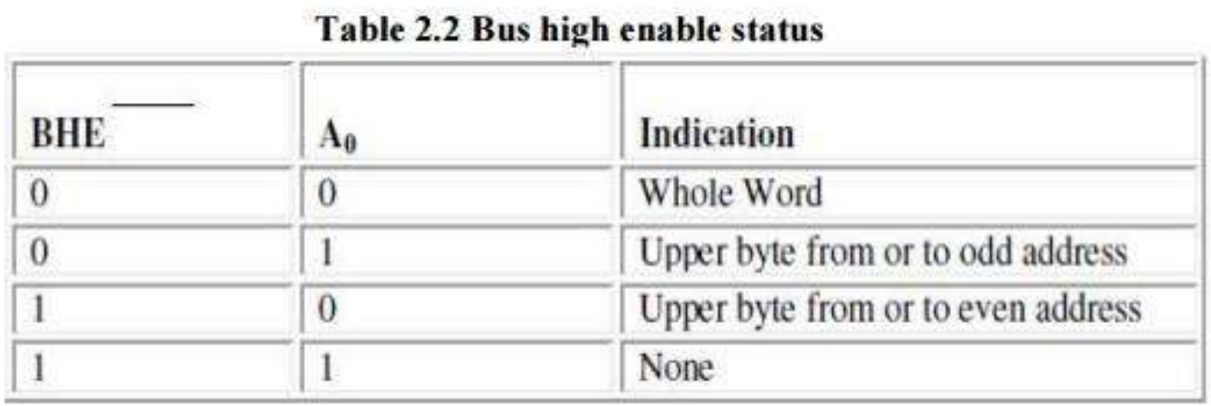
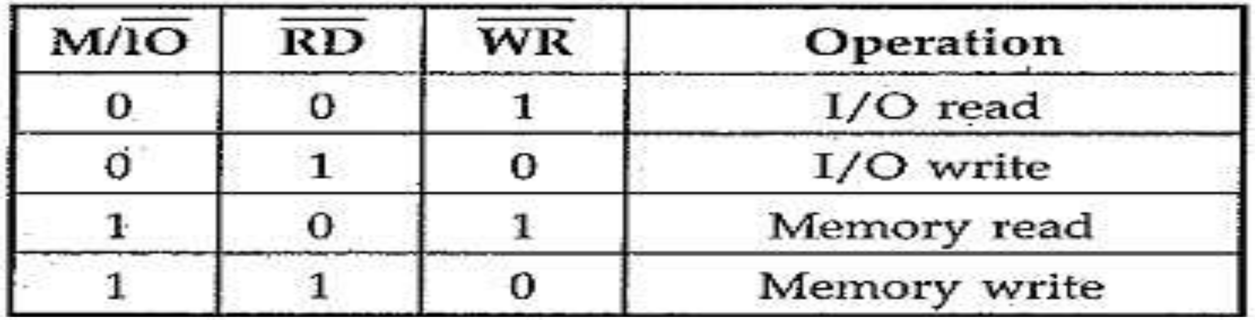
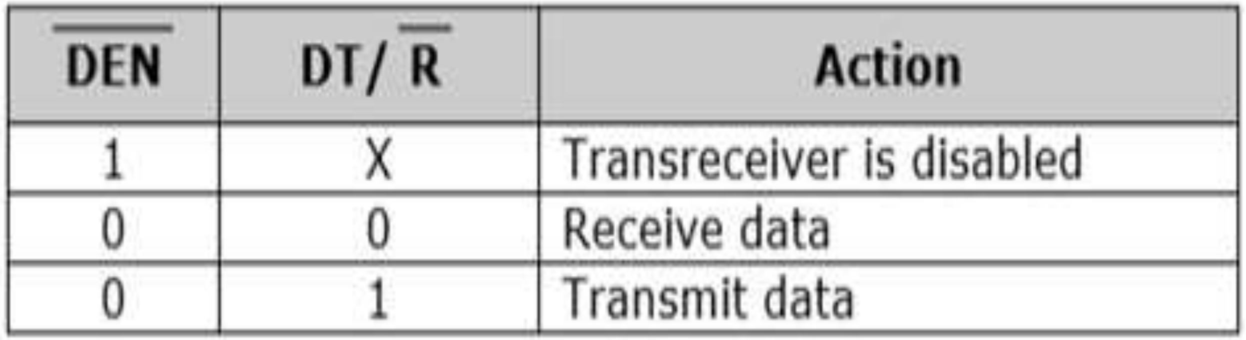


Table 2.2 Bus high enable status

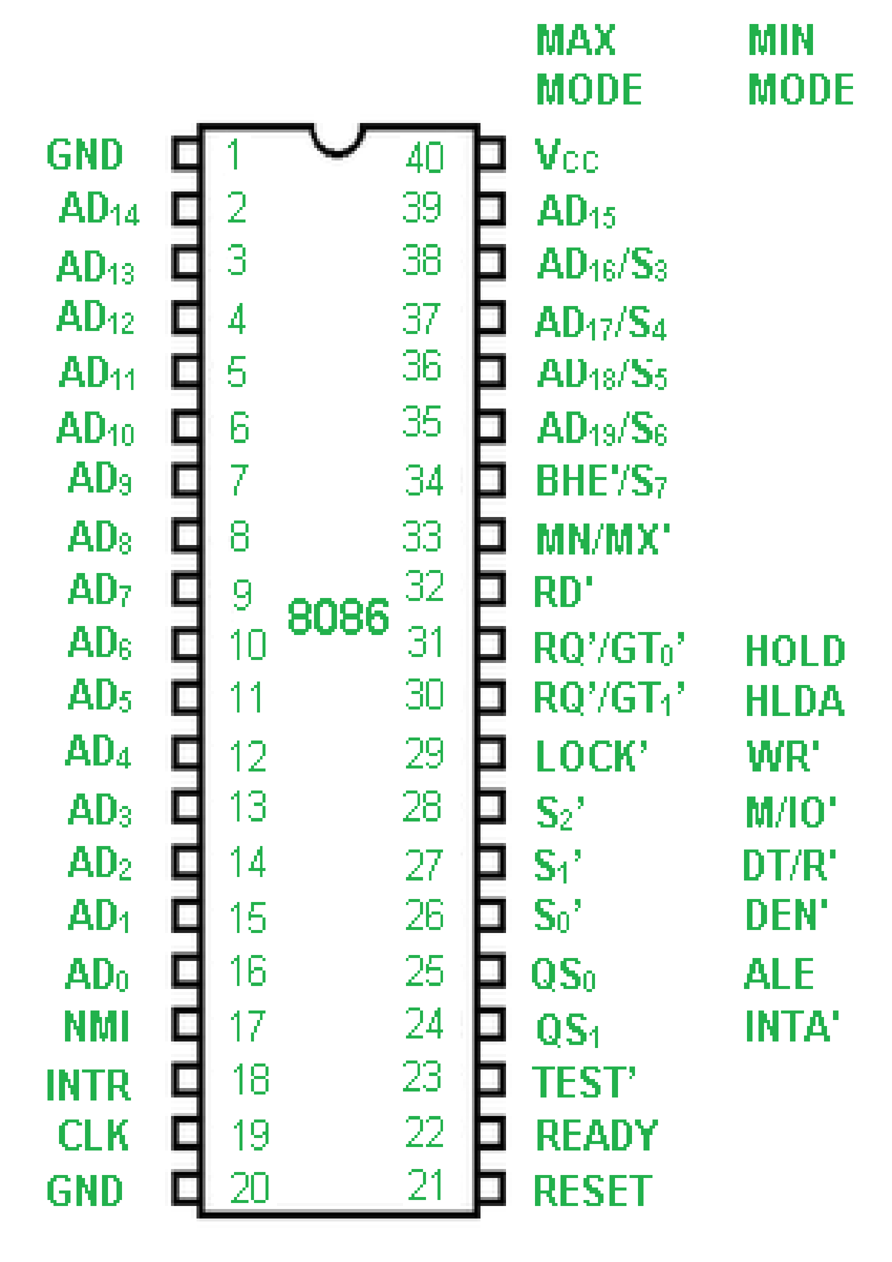
$\overline{BHE}$	A_0	Indication
0	0	Whole Word
0	1	Upper byte from or to odd address
1	0	Upper byte from or to even address
1	1	None



$M/\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write



$\overline{DEN}$	$DT/\overline{R}$	Action
1	X	Transreceiver is disabled
0	0	Receive data
0	1	Transmit data



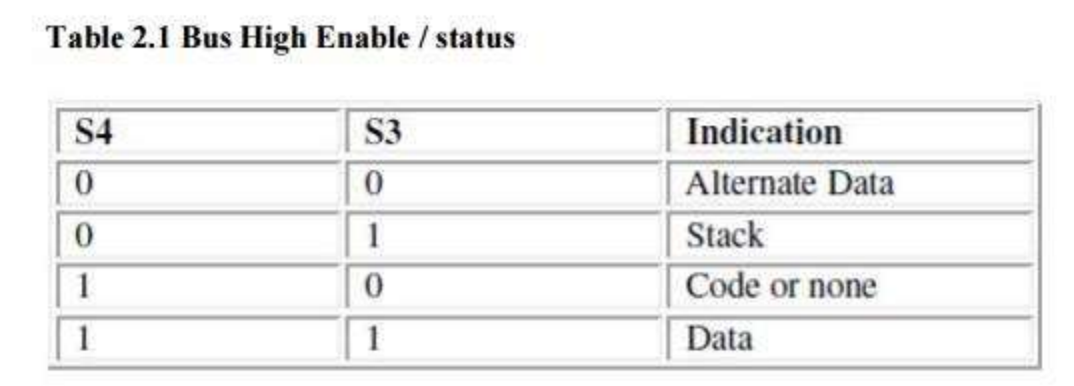
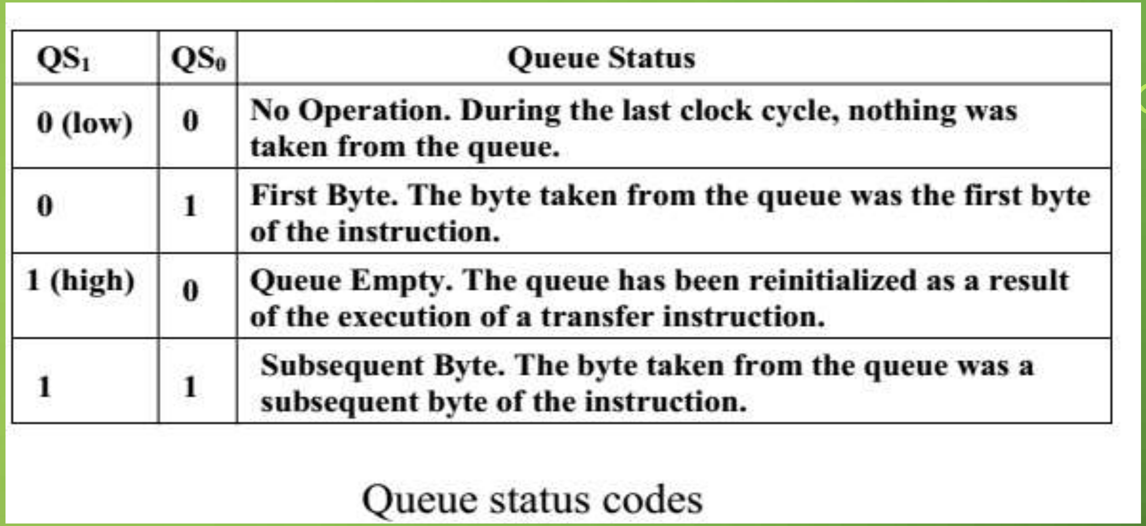


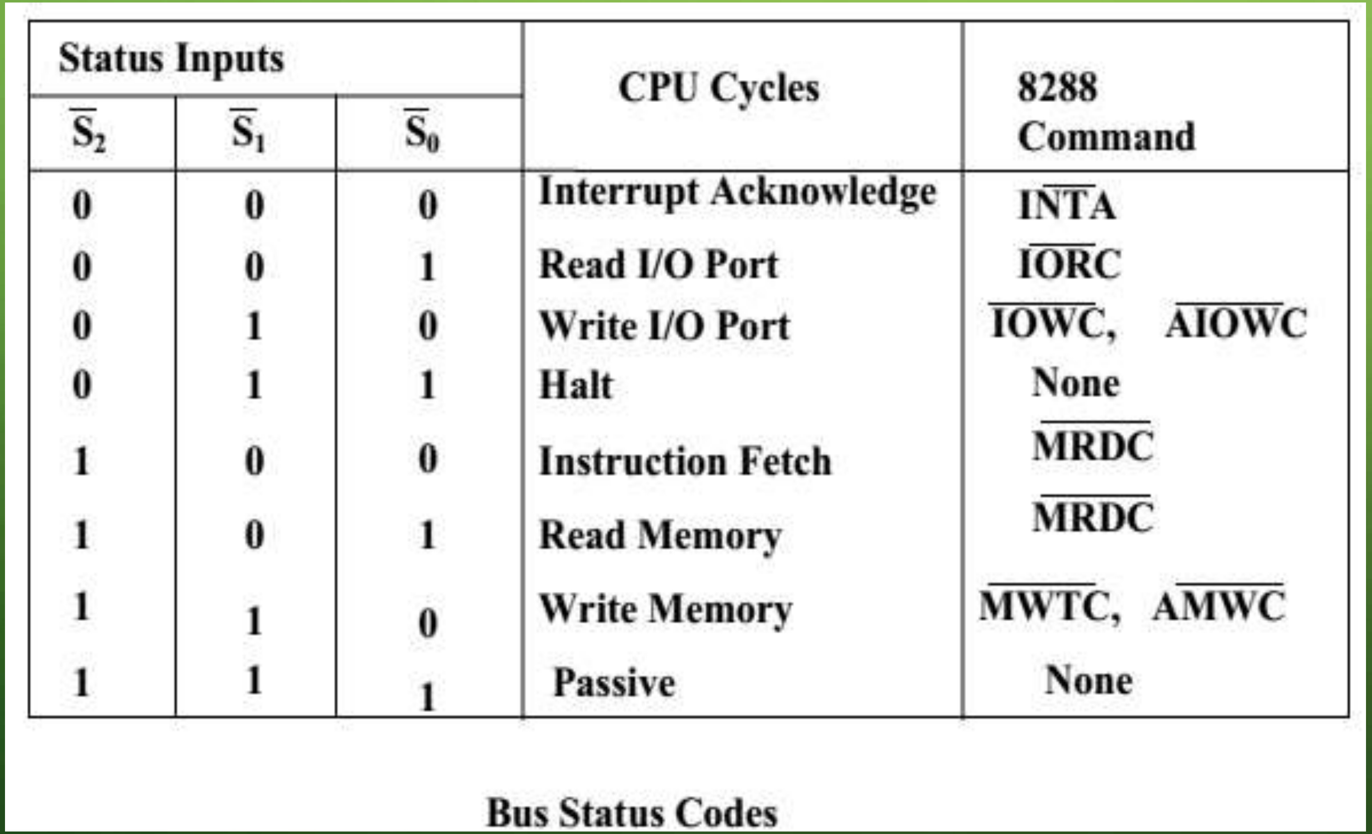
Table 2.1 Bus High Enable / status

S4	S3	Indication
0	0	Alternate Data
0	1	Stack
1	0	Code or none
1	1	Data



QS_1	QS_0	Queue Status
0 (low)	0	No Operation. During the last clock cycle, nothing was taken from the queue.
0	1	First Byte. The byte taken from the queue was the first byte of the instruction.
1 (high)	0	Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction.
1	1	Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction.

Queue status codes



Status Inputs			CPU Cycles	8288 Command
$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

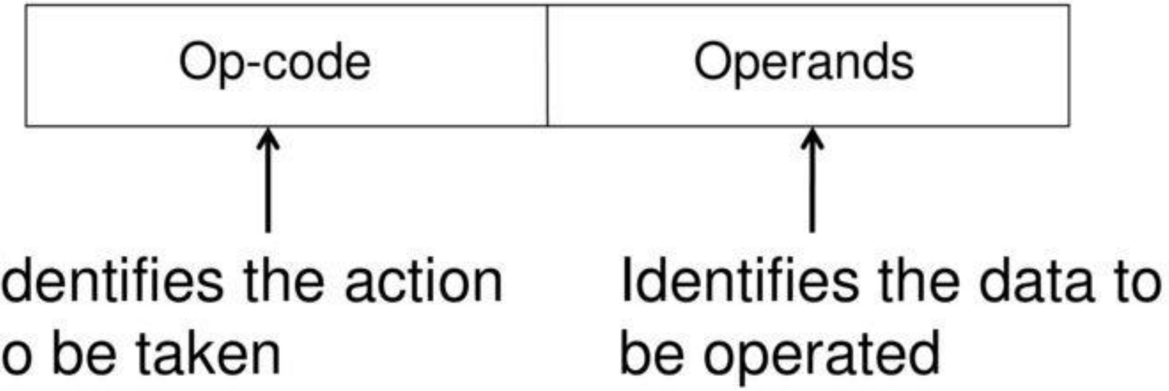
Bus Status Codes

Unit II

Assembly language of 8086

Instruction Format

Format of a typical microprocessor instruction



Op-code | Operands

Identifies the action to be taken

Identifies the data to be operated

E.g. MOV AX,BX

ADD AX,BX

•Op-code are usually written in the form called mnemonic.

E.g. MOVE → MOV

ADDITION → ADD

INCREASE → INC

Addressing modes of 8086

1) Immediate addressing mode
2) Register addressing mode
3) Direct memory addressing mode
4) Register based indirect addressing mode
5) Register relative addressing mode
6) Base indexed addressing mode
7) Relative based indexed addressing mode
8) Implied addressing mode

- **Immediate addressing mode**: In this mode, the operand is specified in the instruction itself. Instructions are longer but the operands are easily identified.
  Example: MOV CL, 12H ,
  MOV AX,1025H
- **Register addressing mode**: In this mode, operands are specified using registers. This addressing mode is normally preferred because the instructions are compact and fastest executing of all instruction forms.
- Registers may be used as source operands, destination operands or both. • Example: MOV AX, BX ,This instruction copies the contents of BX register into AX register. AX ← BX

- **Direct memory addressing mode** : In this mode, address of the operand is directly specified in the instruction. Here only the offset address is specified.

Example: MOV CL, [4321H] ,This instruction moves data from location 4321H in the data segment into CL.

The physical address is calculated as DS * 10H + 4321,Assume DS = 5000H

PA = 50000 + 4321 = 54321H : CL ← [54321H]

Example : MOV BX,[5689H]

- **Register based indirect addressing mode** : In this mode, the effective address of the memory may be taken directly from one of the base register or index register specified by instruction.

If register is SI, DI and BX then DS is by default segment register. • If BP is used, then SS is by default segment register.

Example: MOV CX, [BX] This instruction moves a word from the address pointed by BX and BX + 1 in data segment into CL and CH respectively.

CL ← DS: [BX] and CH ← DS: [BX + 1]

Physical address can be calculated as DS * 10H + BX.

- **Register relative addressing mode**: In this mode, the operand address is calculated using one of the base registers and an 8 bit or a 16 bit displacement. • Example: MOV CL, [BX + 04H]
- This instruction moves a byte from the address pointed by BX + 4 in data segment to CL. CL ← DS: [BX + 04H] • Physical address can be calculated as DS * 10H + BX + 4H.

- **Base indexed addressing mode:** Here, operand address is calculated as base register plus an index register.

Example: • MOV CL, [BX + SI] ,This instruction moves a byte from the address pointed by BX + SI in data segment to CL.

CL ← DS: [BX + SI] • Physical address can be calculated as DS * 10H + BX + SI.

Example : MOV AX, [SI+BX]

- **Relative based indexed addressing mode** :In this mode, the address of the operand is calculated as the sum of base register, index register and 8 bit or 16 bit displacement.

Example: MOV CL, [BX + DI + 20] ,This instruction moves a byte from the address pointed by BX + DI + 20H in data segment to CL.

CL ← DS: [BX + DI + 20H]

Physical address can be calculated as DS * 10H + BX + DI + 20H.

MOV AX,[BX+SI+3000H]

- **Implied addressing mode** : In this mode, the operands are implied and are hence not specified in the instruction.

Example: • STC • This sets the carry flag.

Intra segment Direct: The effective branch address is sum of 8 or 16 bit displacement and the current contents of IP(Instruction Pointer).It can be used with either conditional or unconditional branching.

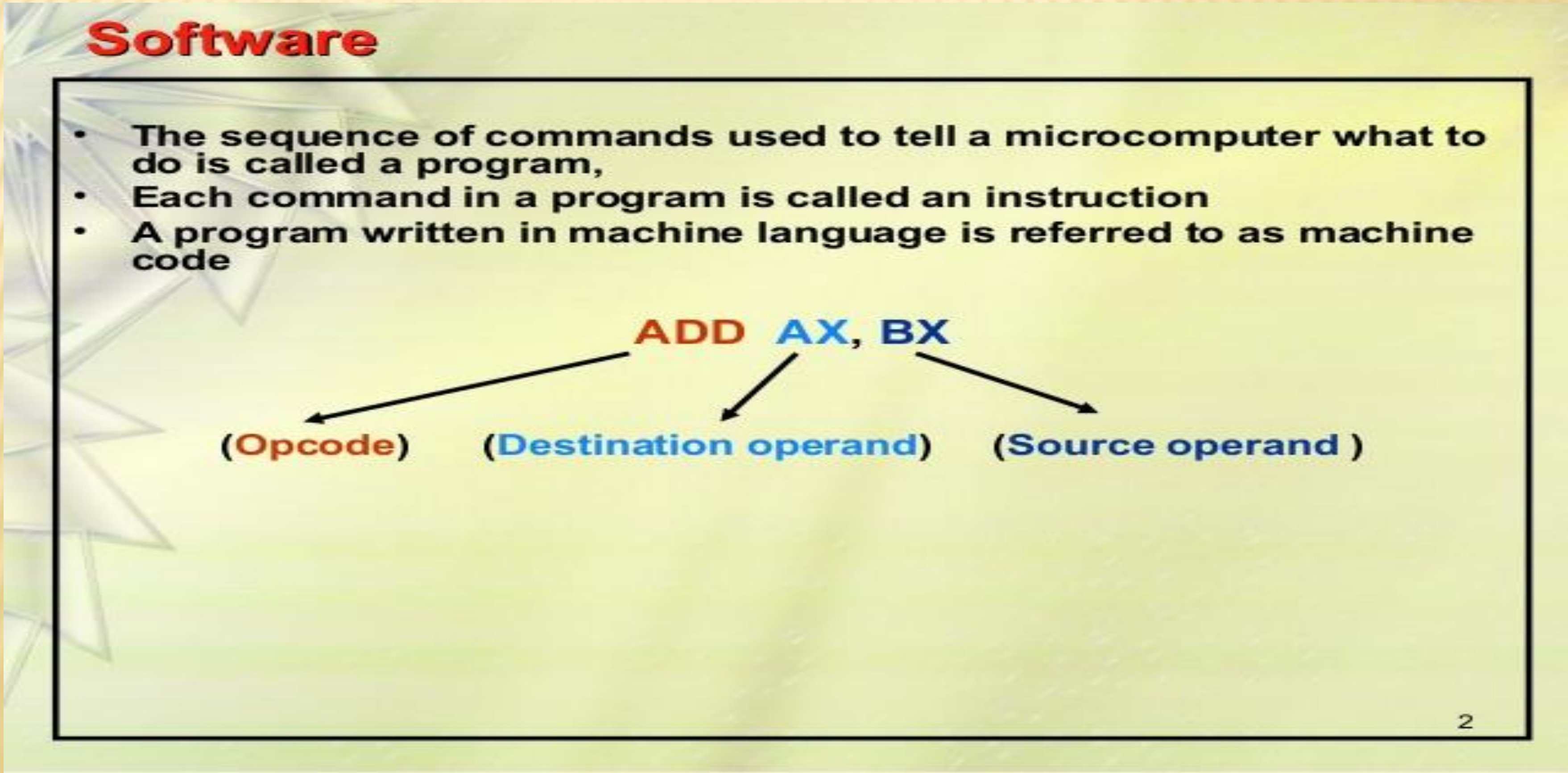
Inter segment Indirect: The effective branch address is contents of register or memory location that is accessed using any of the data related addressing mode except immediate mode. It can be used only for unconditional branch instruction.

Intersegment Direct: Replaces the content of IP with part of the instruction and the contents of CS with another part of the instruction. This mode is provide a way of branching from one code segment to another.

Intersegment Indirect: Replaces the contents of IP and CS with the contents of two consecutive words in memory that are referenced using any one of the data related addressing mode except immediate

Classification of Instruction Set □

1)Data Transfer Instructions
2)Arithmetic Instructions
3)Bit Manipulation Instructions □
4)Program Execution Transfer Instructions □
5)String Instructions □
6)Processor Control Instructions



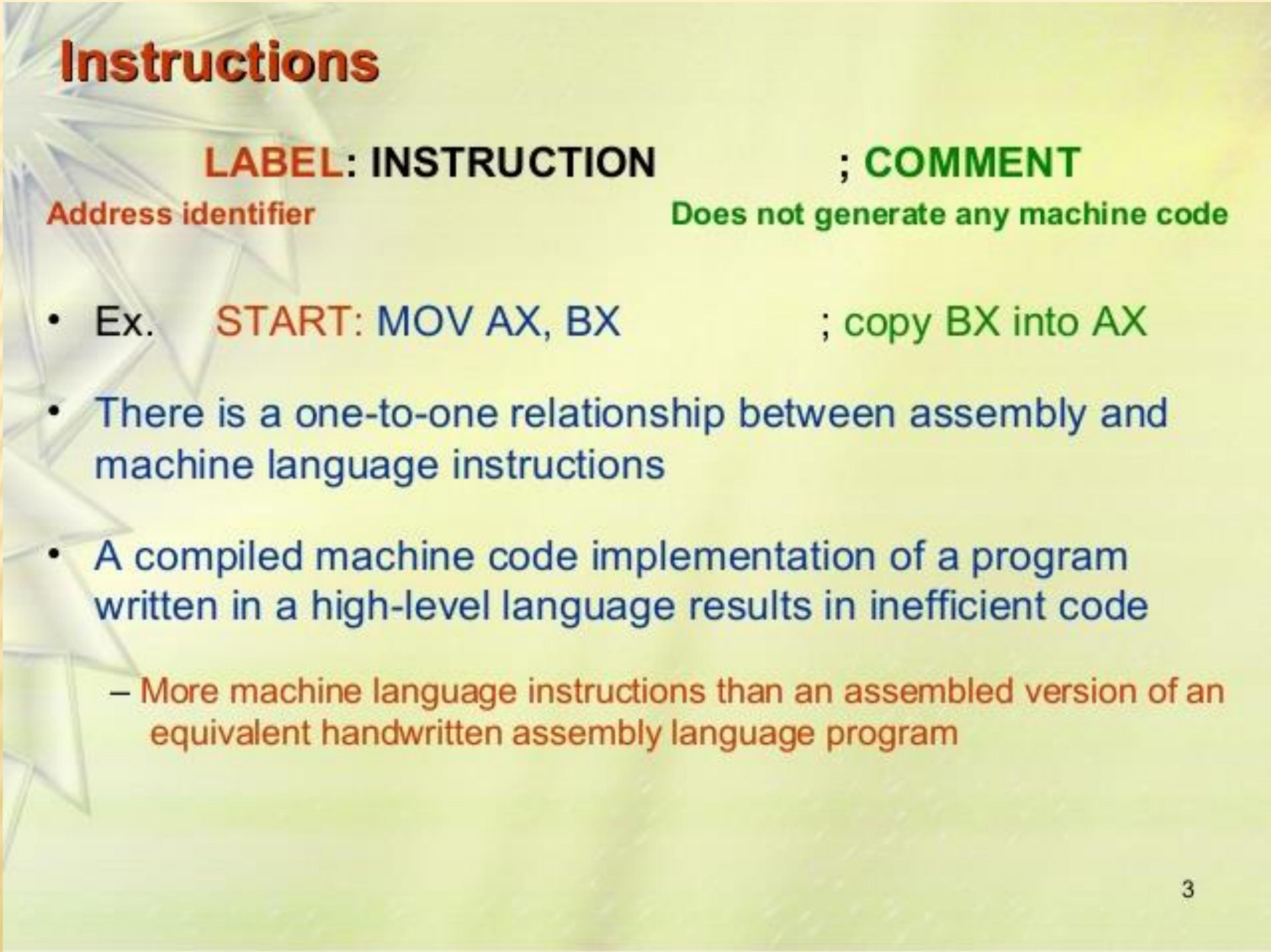
Software

- The sequence of commands used to tell a microcomputer what to do is called a program,
- Each command in a program is called an instruction
- A program written in machine language is referred to as machine code

ADD AX, BX

(Opcode) (Destination operand) (Source operand)

2

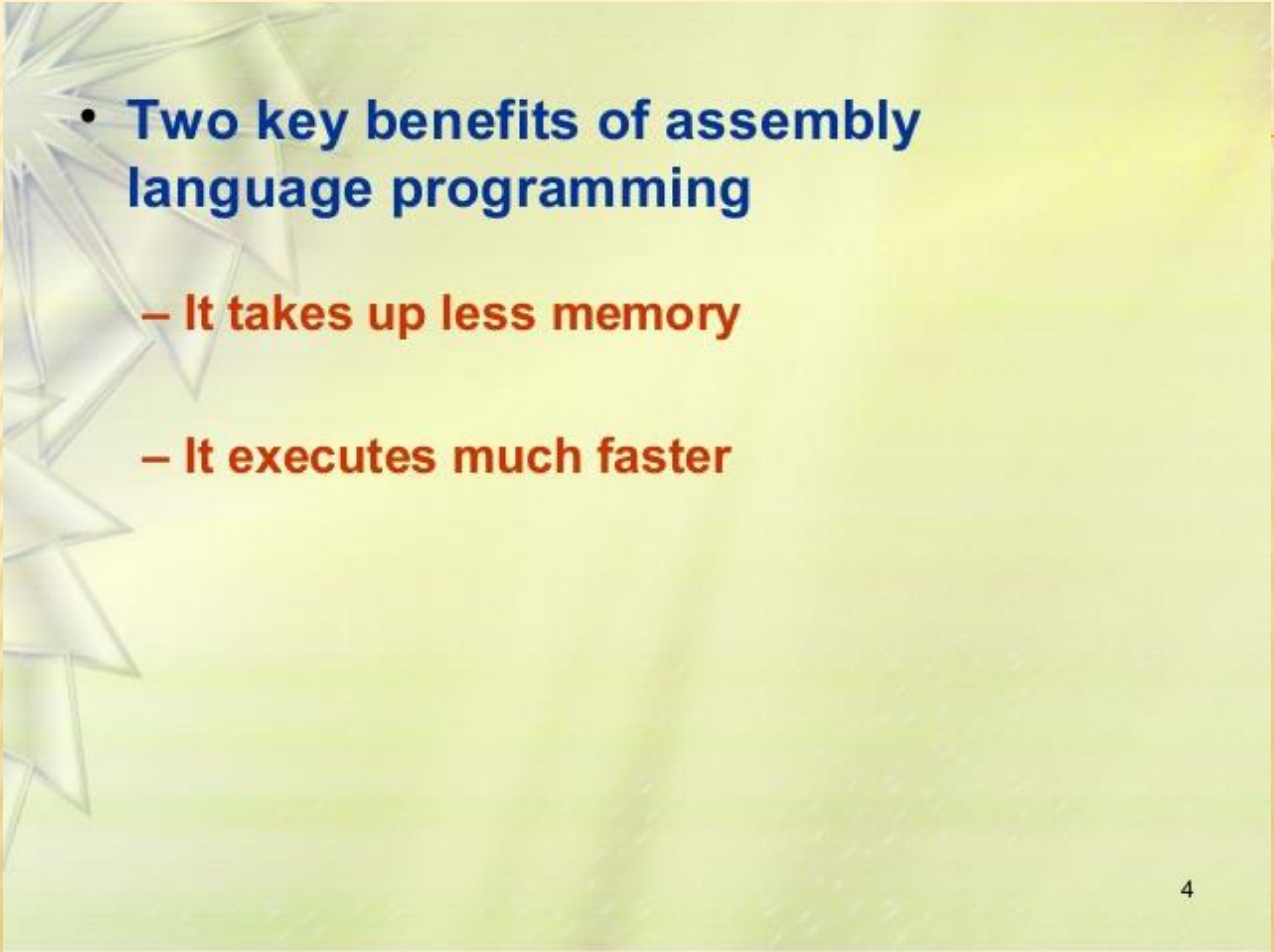


Instructions

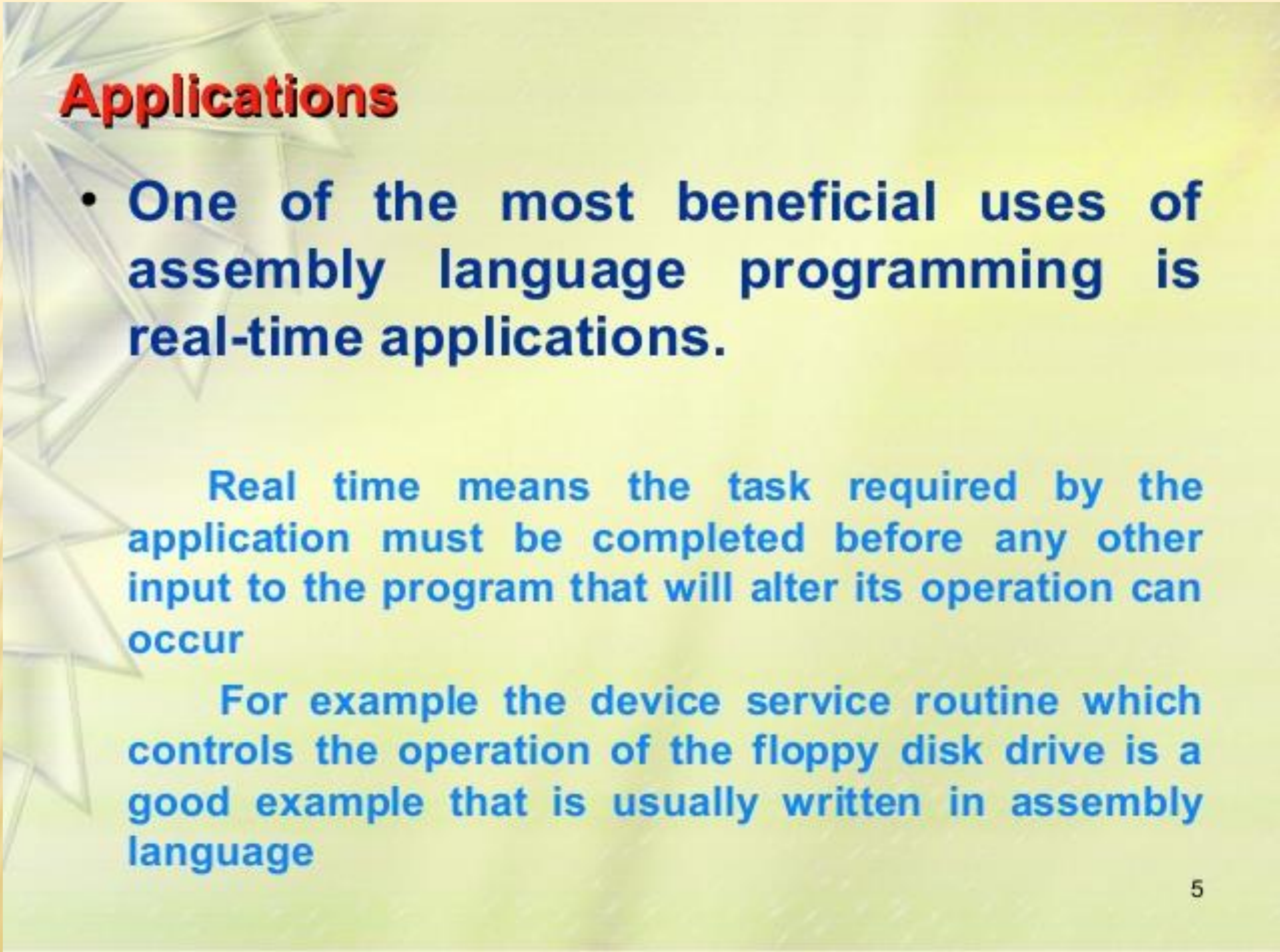
LABEL: INSTRUCTION ; COMMENT

Address identifier **Does not generate any machine code**

- Ex. START: MOV AX, BX ; copy BX into AX
- There is a one-to-one relationship between assembly and machine language instructions
- A compiled machine code implementation of a program written in a high-level language results in inefficient code
  - – More machine language instructions than an assembled version of an equivalent handwritten assembly language program



- **Two key benefits of assembly language programming**
  - **– It takes up less memory**
  - **– It executes much faster**

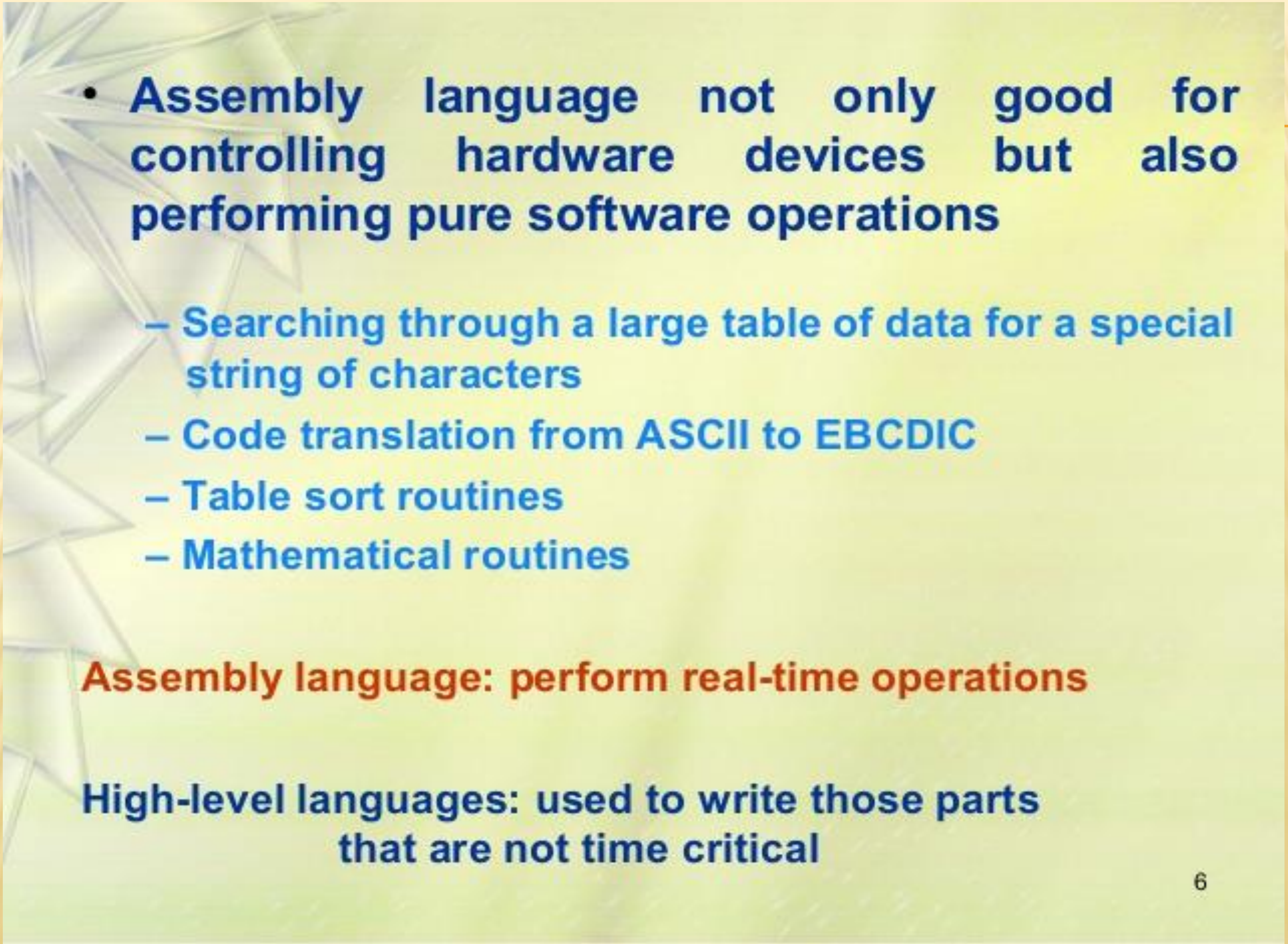


Applications

- One of the most beneficial uses of assembly language programming is real-time applications.

Real time means the task required by the application must be completed before any other input to the program that will alter its operation can occur

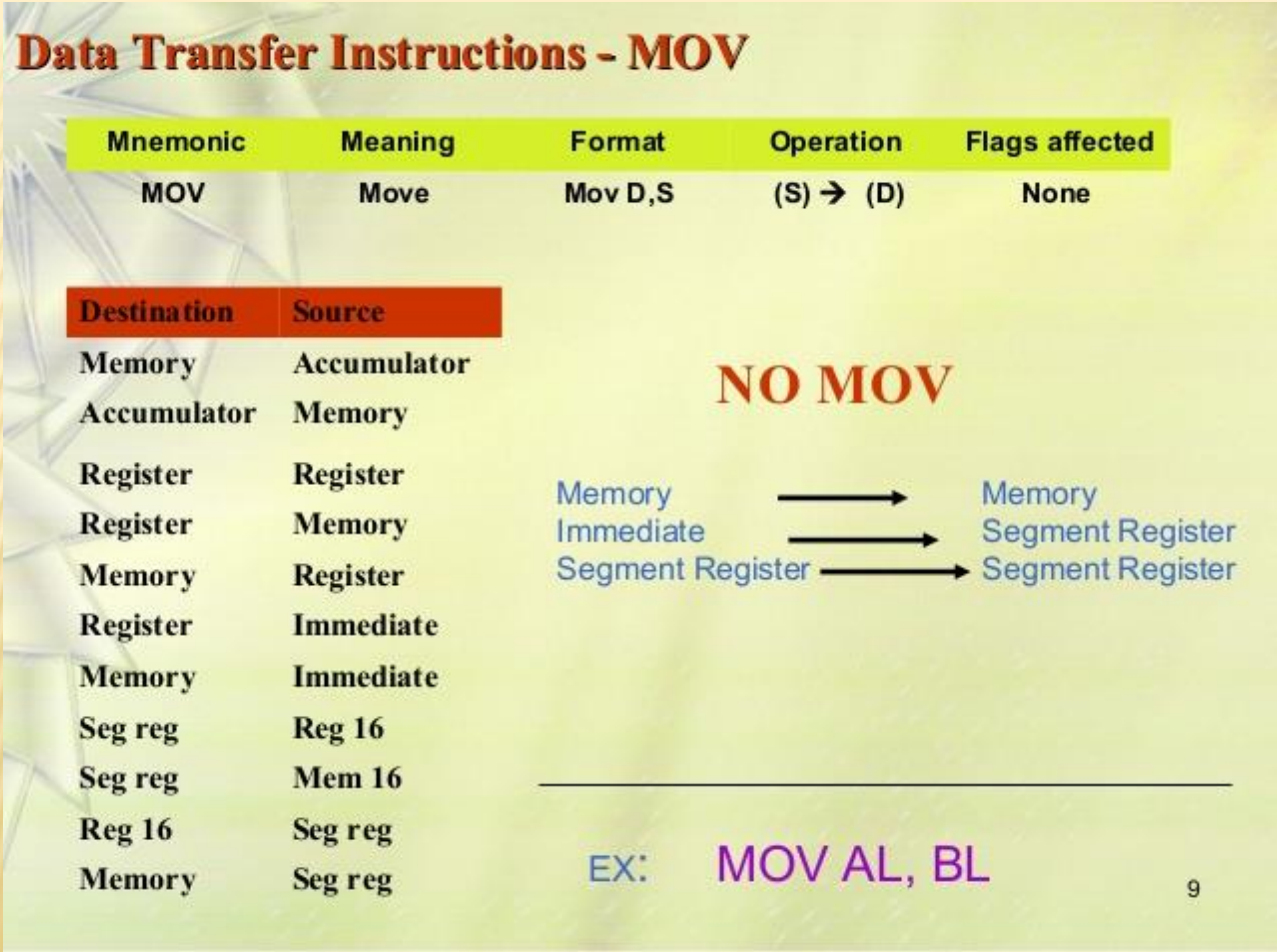
For example the device service routine which controls the operation of the floppy disk drive is a good example that is usually written in assembly language



- **Assembly language not only good for controlling hardware devices but also performing pure software operations**
  - Searching through a large table of data for a special string of characters
  - Code translation from ASCII to EBCDIC
  - Table sort routines
  - Mathematical routines

Assembly language: perform real-time operations

High-level languages: used to write those parts that are not time critical



Data Transfer Instructions - MOV

Mnemonic	Meaning	Format	Operation	Flags affected
MOV	Move	Mov D,S	(S) → (D)	None

Destination	Source
Memory	Accumulator
Accumulator	Memory
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Seg reg	Reg 16
Seg reg	Mem 16
Reg 16	Seg reg
Memory	Seg reg

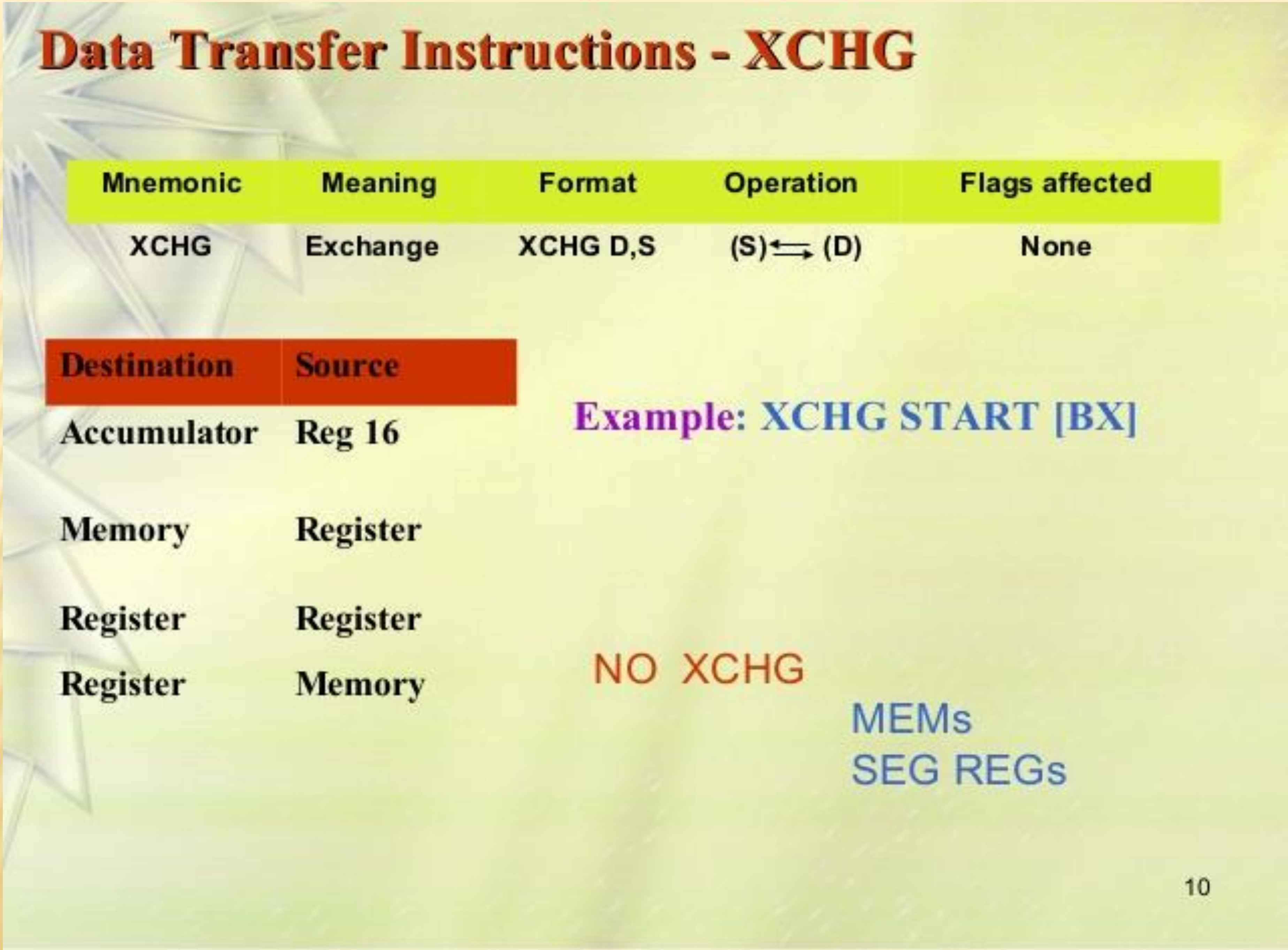
NO MOV

Memory → Memory

Immediate → Segment Register

Segment Register → Segment Register

EX: MOV AL, BL



Data Transfer Instructions - XCHG

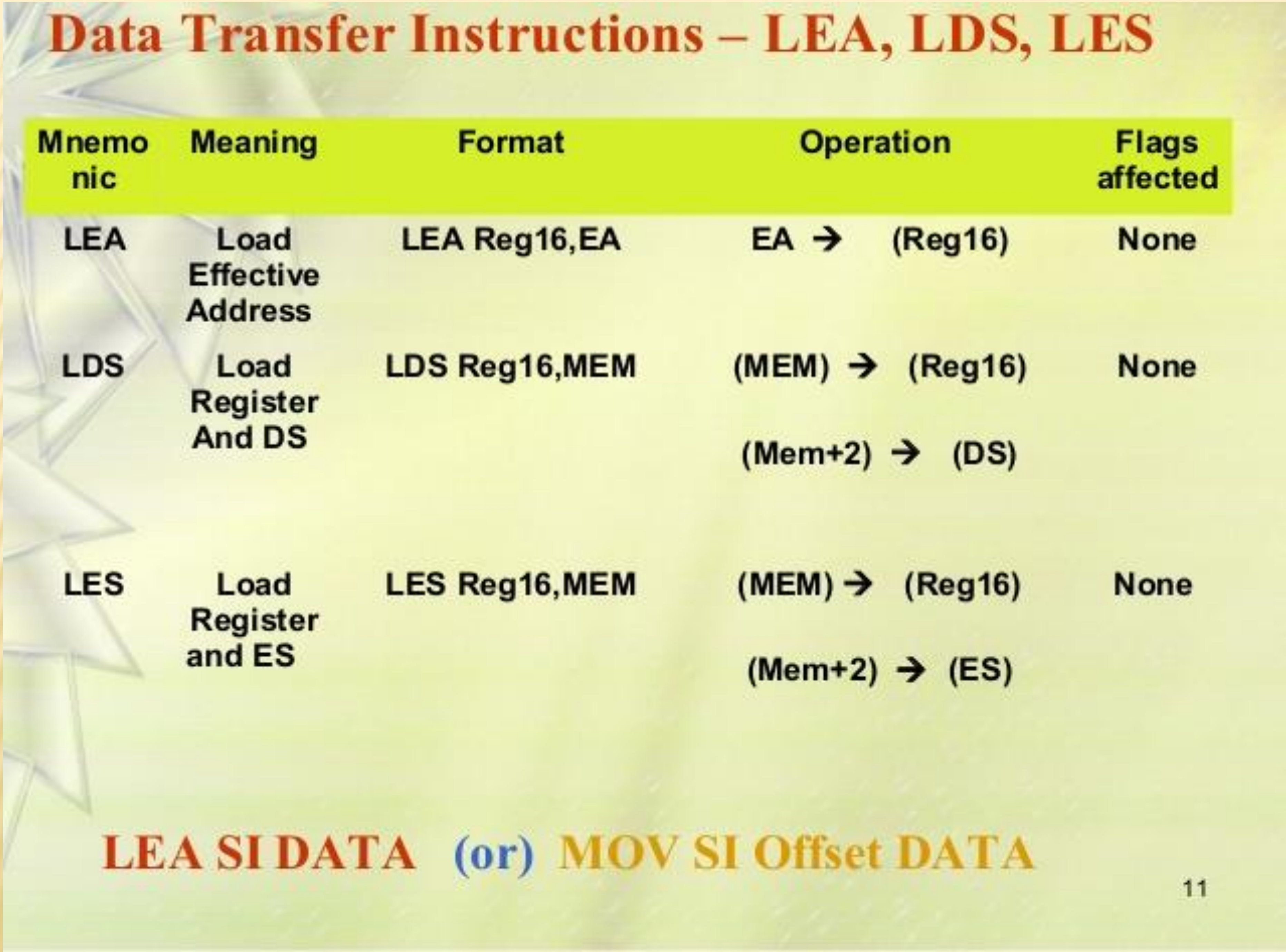
Mnemonic	Meaning	Format	Operation	Flags affected
XCHG	Exchange	XCHG D,S	(S) ⇆ (D)	None

Destination	Source
Accumulator	Reg 16
Memory	Register
Register	Register
Register	Memory

Example: XCHG START [BX]

NO XCHG

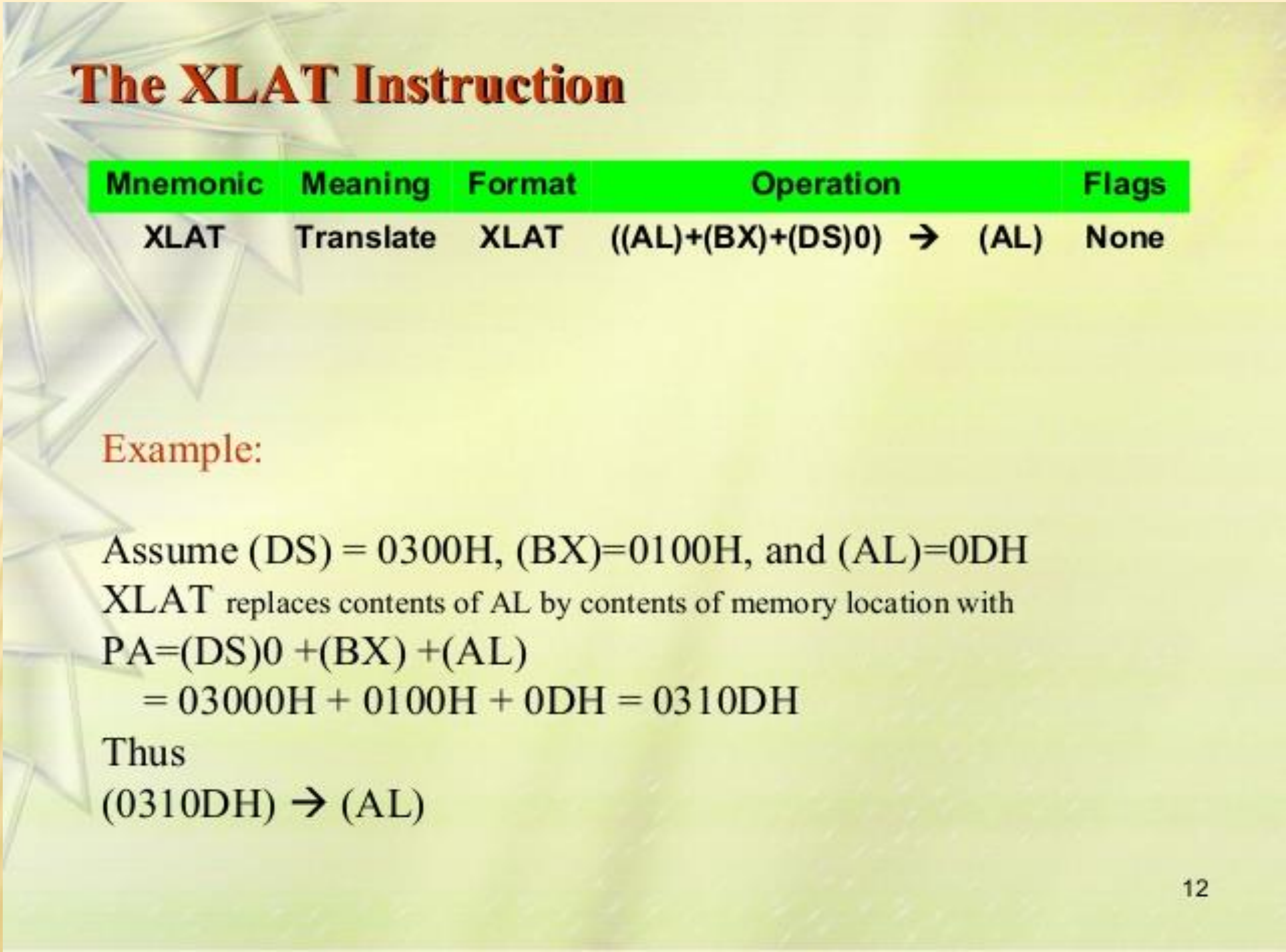
MEMs
SEG REGs



Data Transfer Instructions – LEA, LDS, LES

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load Effective Address	LEA Reg16,EA	EA → (Reg16)	None
LDS	Load Register And DS	LDS Reg16,MEM	(MEM) → (Reg16) (Mem+2) → (DS)	None
LES	Load Register and ES	LES Reg16,MEM	(MEM) → (Reg16) (Mem+2) → (ES)	None

LEA SI DATA (or) MOV SI Offset DATA



The XLAT Instruction

Mnemonic	Meaning	Format	Operation	Flags
XLAT	Translate	XLAT	((AL)+(BX)+(DS)0) → (AL)	None

Example:

Assume (DS) = 0300H, (BX)=0100H, and (AL)=0DH

XLAT replaces contents of AL by contents of memory location with

PA=(DS)0 +(BX) +(AL)

= 03000H + 0100H + 0DH = 0310DH

Thus

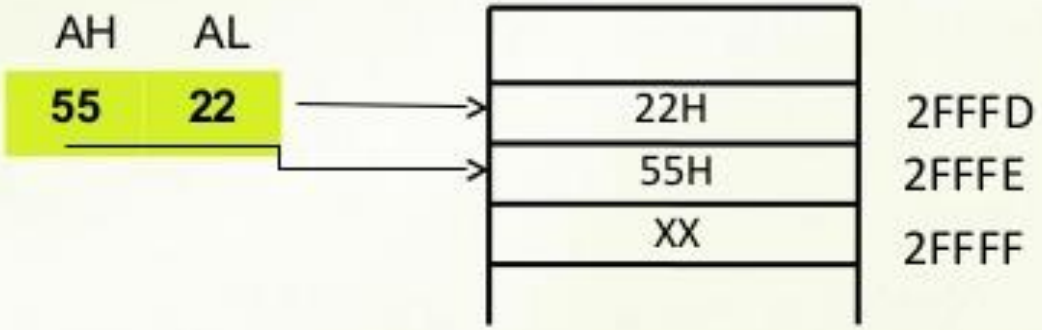
(0310DH) → (AL)

Data Transfer Instructions - PUSH & POP

PUSH: Push to Stack

Ex: PUSH AX
PUSH DS
PUSH [5000H]

PUSH AX



AH AL

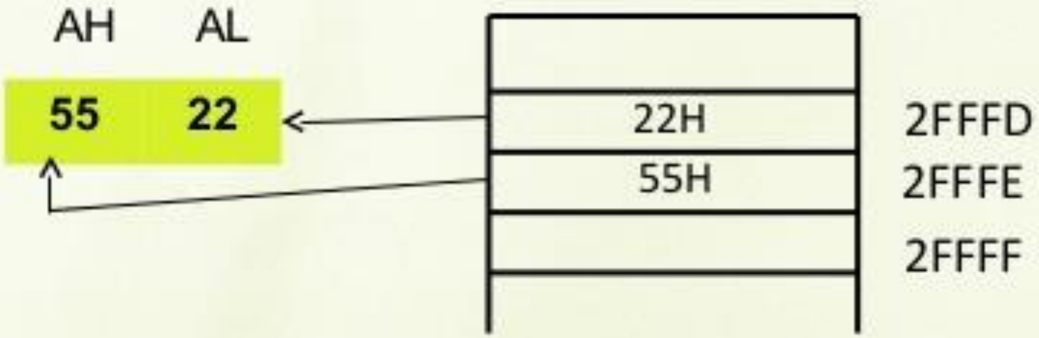
55 22

22H	2FFFD
55H	2FFFE
XX	2FFFF

POP: Pop from Stack

Ex: POP AX
POP DS
POP [5000H]

POP AX



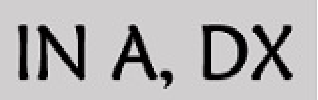
AH AL

55 22

22H	2FFFD
55H	2FFFE
	2FFFF

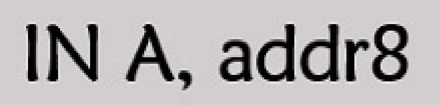
IN and OUT Instructions

The data present in the input port is transferred to accumulator and its address is present in the DX register given in the operand.



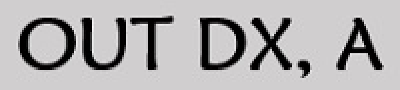
```
IN A, DX
```

The data of the input port is moved to the accumulator whose memory address is given in the instruc



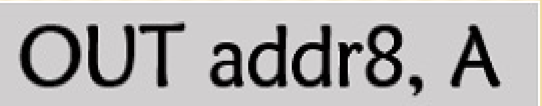
```
IN A, addr8
```

The data in the accumulator is moved to the output port whose address is specified in the DX register.



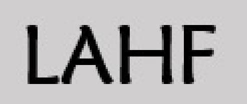
```
OUT DX, A
```

The data in the accumulator is moved to the port whose address is given in the instruction.



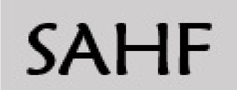
```
OUT addr8, A
```

The lower byte of the data of the flag register is moved to the higher byte register of the accumulator.

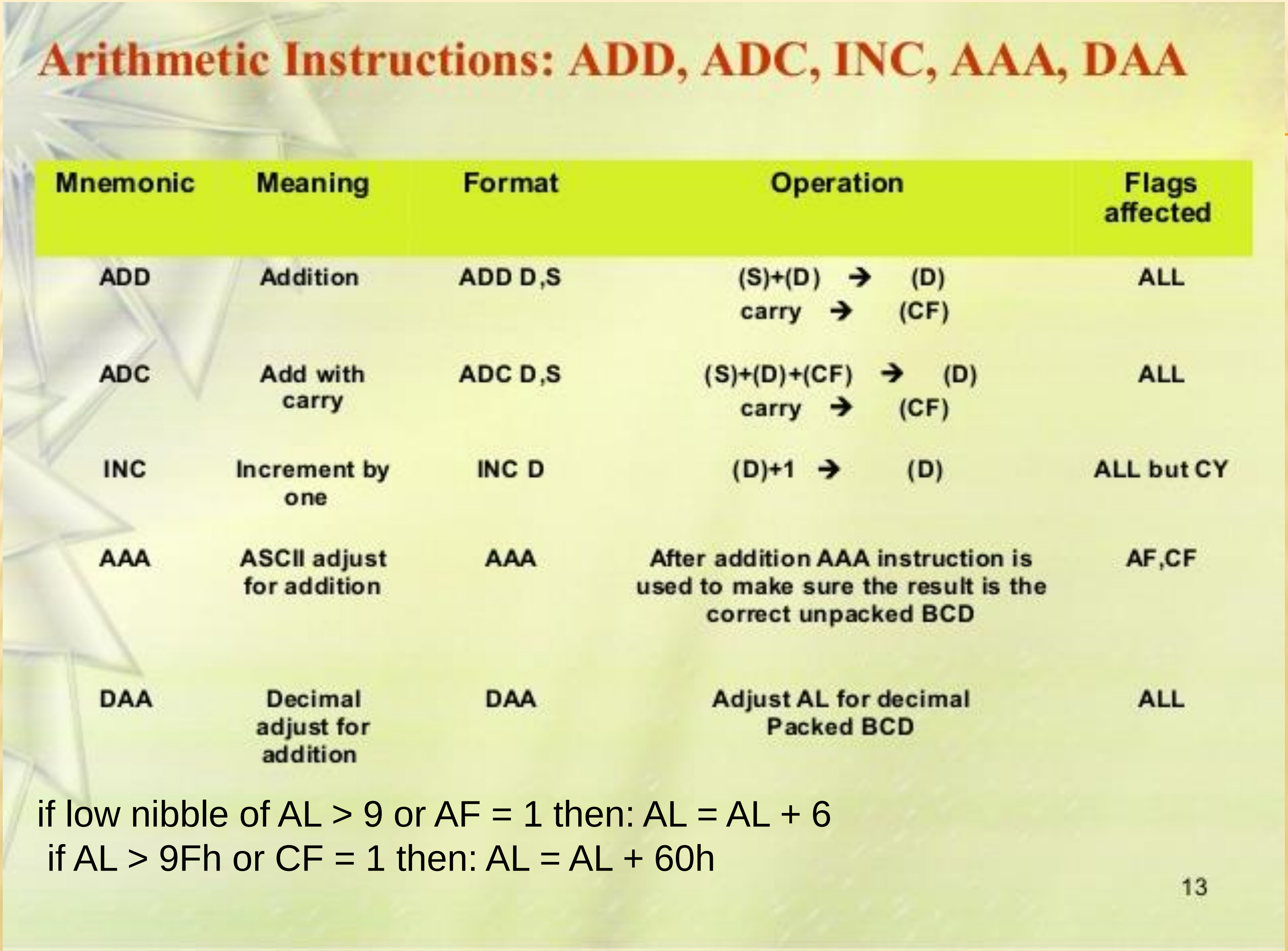


```
LAHF
```

It moves the data in the higher byte of flag register to the lower byte flag register.



```
SAHF
```

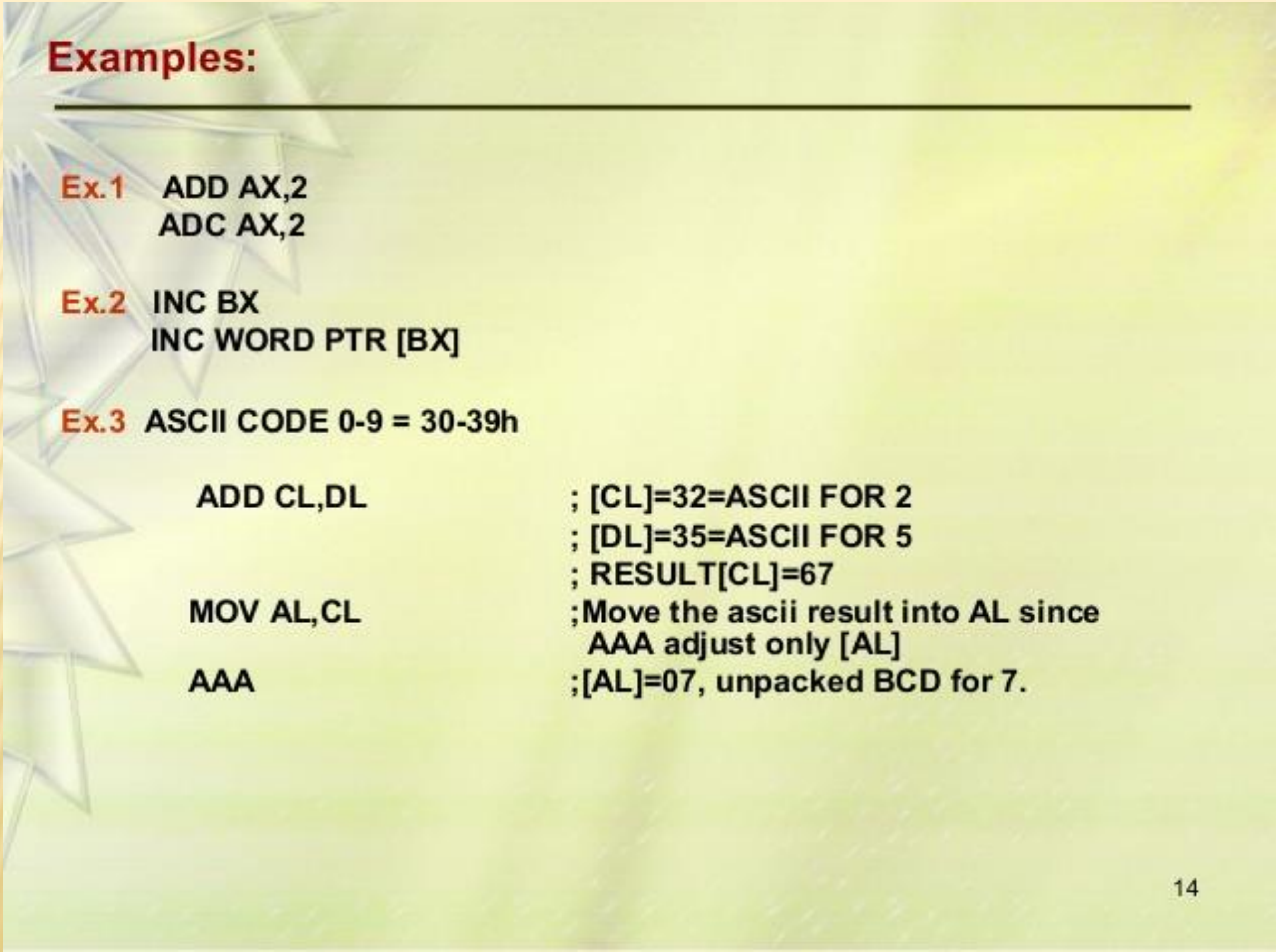


Arithmetic Instructions: ADD, ADC, INC, AAA, DAA

Mnemonic	Meaning	Format	Operation	Flags affected
ADD	Addition	ADD D,S	(S)+(D) → (D) carry → (CF)	ALL
ADC	Add with carry	ADC D,S	(S)+(D)+(CF) → (D) carry → (CF)	ALL
INC	Increment by one	INC D	(D)+1 → (D)	ALL but CY
AAA	ASCII adjust for addition	AAA	After addition AAA instruction is used to make sure the result is the correct unpacked BCD	AF,CF
DAA	Decimal adjust for addition	DAA	Adjust AL for decimal Packed BCD	ALL

if low nibble of AL > 9 or AF = 1 then: AL = AL + 6

if AL > 9Fh or CF = 1 then: AL = AL + 60h



Examples:

Ex.1

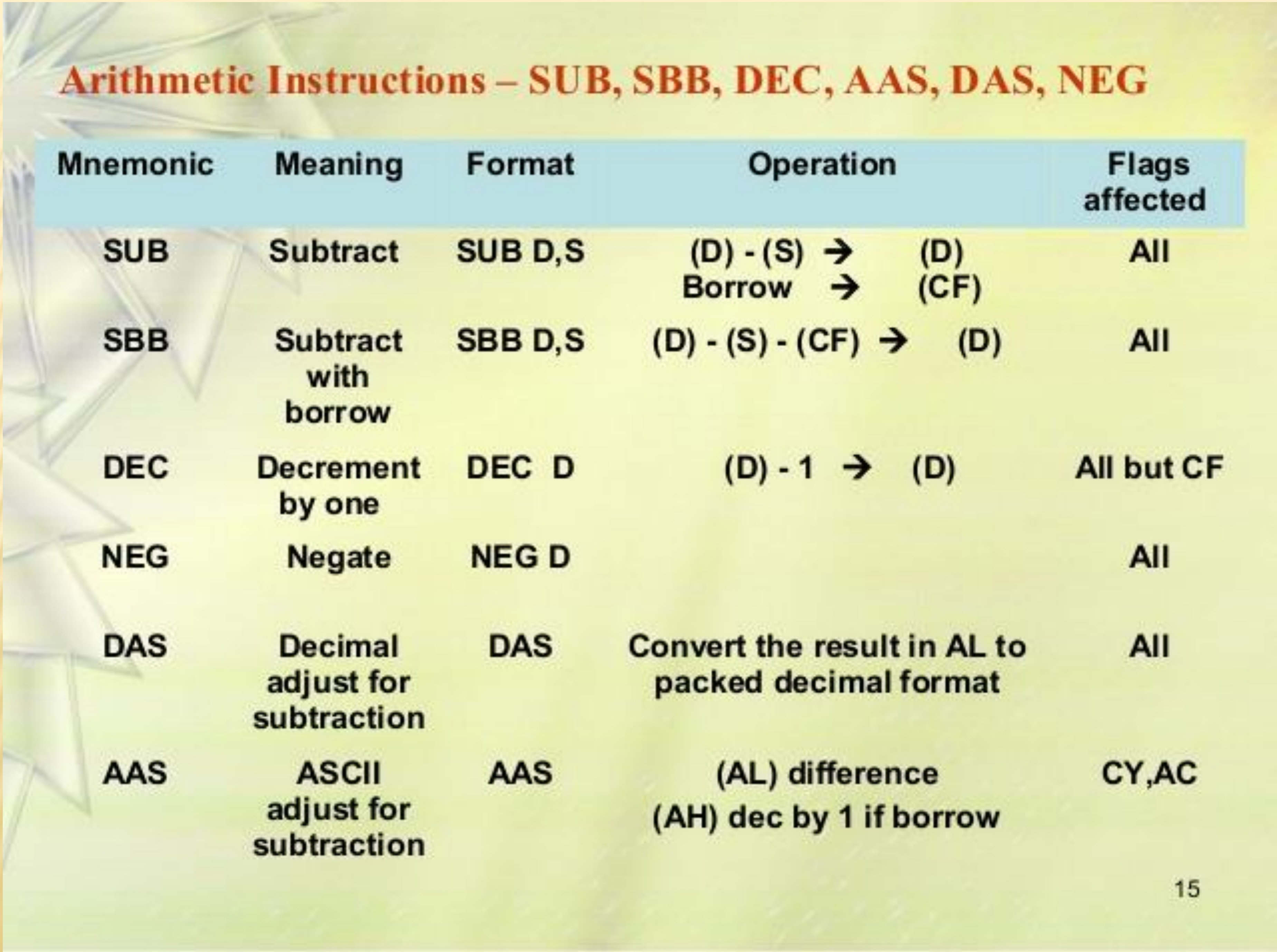
```
ADD AX,2
ADC AX,2
```

Ex.2

```
INC BX
INC WORD PTR [BX]
```

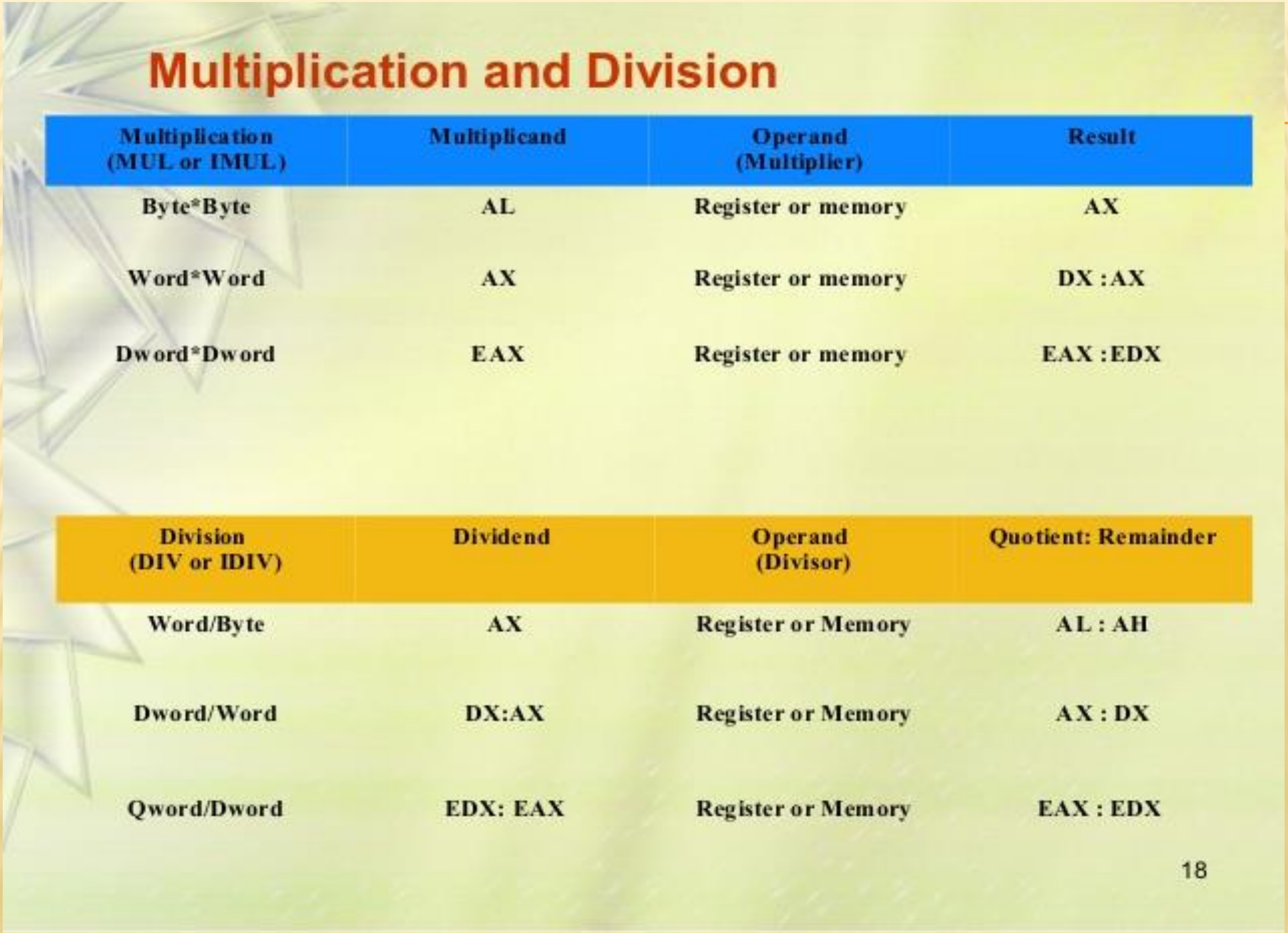
Ex.3 ASCII CODE 0-9 = 30-39h

```
ADD CL,DL          ; [CL]=32=ASCII FOR 2
                   ; [DL]=35=ASCII FOR 5
                   ; RESULT[CL]=67
MOV AL,CL          ;Move the ascii result into AL since
                     AAA adjust only [AL]
AAA                ;[AL]=07, unpacked BCD for 7.
```



Arithmetic Instructions – SUB, SBB, DEC, AAS, DAS, NEG

Mnemonic	Meaning	Format	Operation	Flags affected
SUB	Subtract	SUB D,S	(D) - (S) → (D) Borrow → (CF)	All
SBB	Subtract with borrow	SBB D,S	(D) - (S) - (CF) → (D)	All
DEC	Decrement by one	DEC D	(D) - 1 → (D)	All but CF
NEG	Negate	NEG D		All
DAS	Decimal adjust for subtraction	DAS	Convert the result in AL to packed decimal format	All
AAS	ASCII adjust for subtraction	AAS	(AL) difference (AH) dec by 1 if borrow	CY,AC



Multiplication and Division

Multiplication (MUL or IMUL)	Multiplicand	Operand (Multiplier)	Result
Byte*Byte	AL	Register or memory	AX
Word*Word	AX	Register or memory	DX :AX
Dword*Dword	EAX	Register or memory	EAX :EDX

Division (DIV or IDIV)	Dividend	Operand (Divisor)	Quotient: Remainder
Word/Byte	AX	Register or Memory	AL : AH
Dword/Word	DX:AX	Register or Memory	AX : DX
Qword/Dword	EDX: EAX	Register or Memory	EAX : EDX

Multiplication and Division Examples

Ex1: Assume that each instruction starts from these values:
AL = 85H, BL = 35H, AH = 0H

1. MUL BL → AL . BL = 85H * 35H = 1B89H → AX = 1B89H

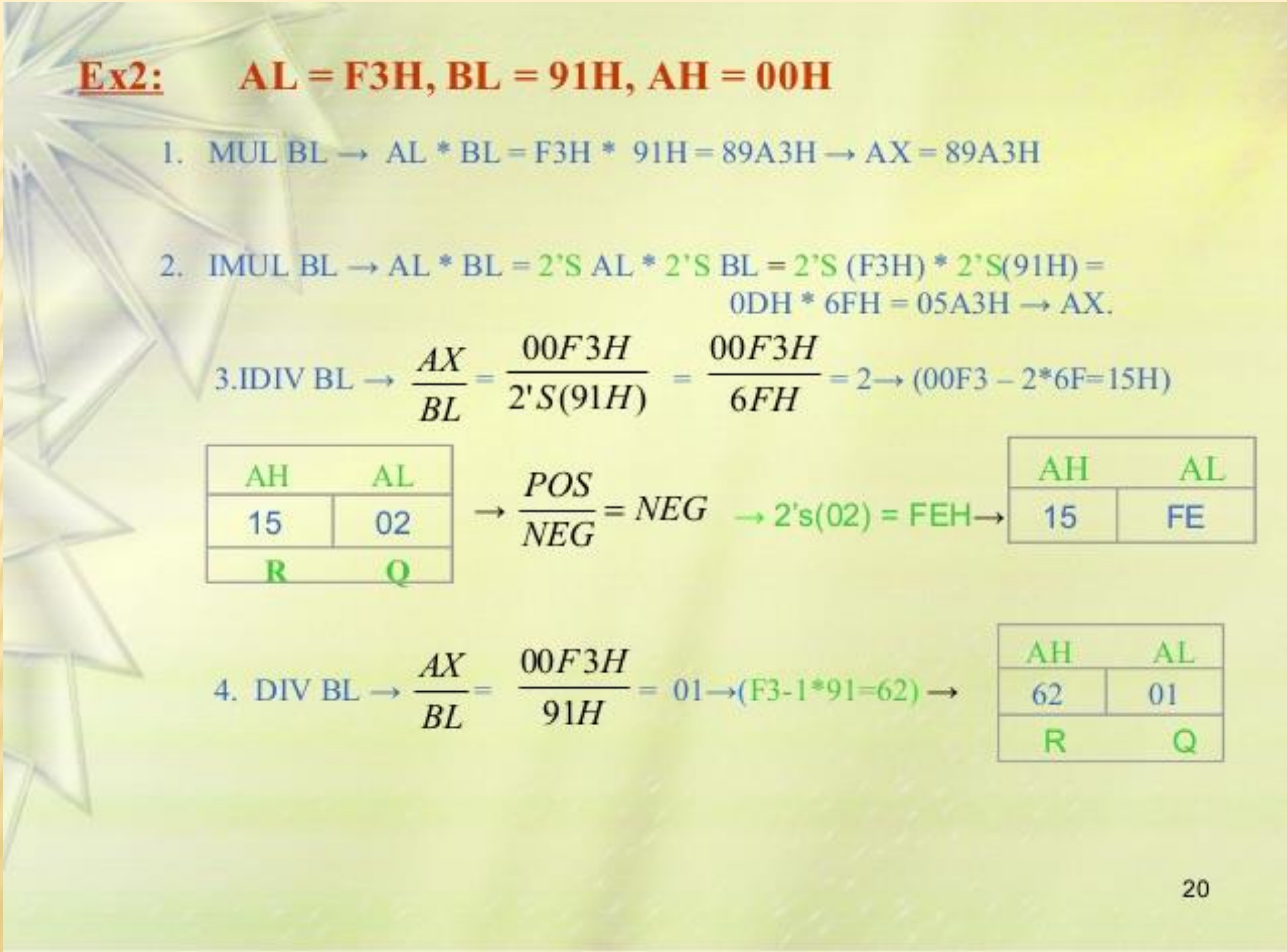
2. IMUL BL → AL . BL = 2'S AL * BL = 2'S (85H) * 35H
   = 7BH * 35H = 1977H→ 2's comp → E689H → AX.

- DIV BL → $\frac{AX}{BL} = \frac{0085H}{35H}$ = 02 (85-02*35=1B) →

AH	AL
1B	02

4. IDIV BL → $\frac{AX}{BL} = \frac{0085H}{35H} =$

AH	AL
1B	02



Ex2: AL = F3H, BL = 91H, AH = 00H

1. MUL BL → AL * BL = F3H * 91H = 89A3H → AX = 89A3H

2. IMUL BL → AL * BL = 2'S AL * 2'S BL = 2'S (F3H) * 2'S(91H) =
   0DH * 6FH = 05A3H → AX.

3.IDIV BL → $\frac{AX}{BL} = \frac{00F3H}{2'S(91H)} = \frac{00F3H}{6FH}$ = 2→ (00F3 – 2*6F=15H)

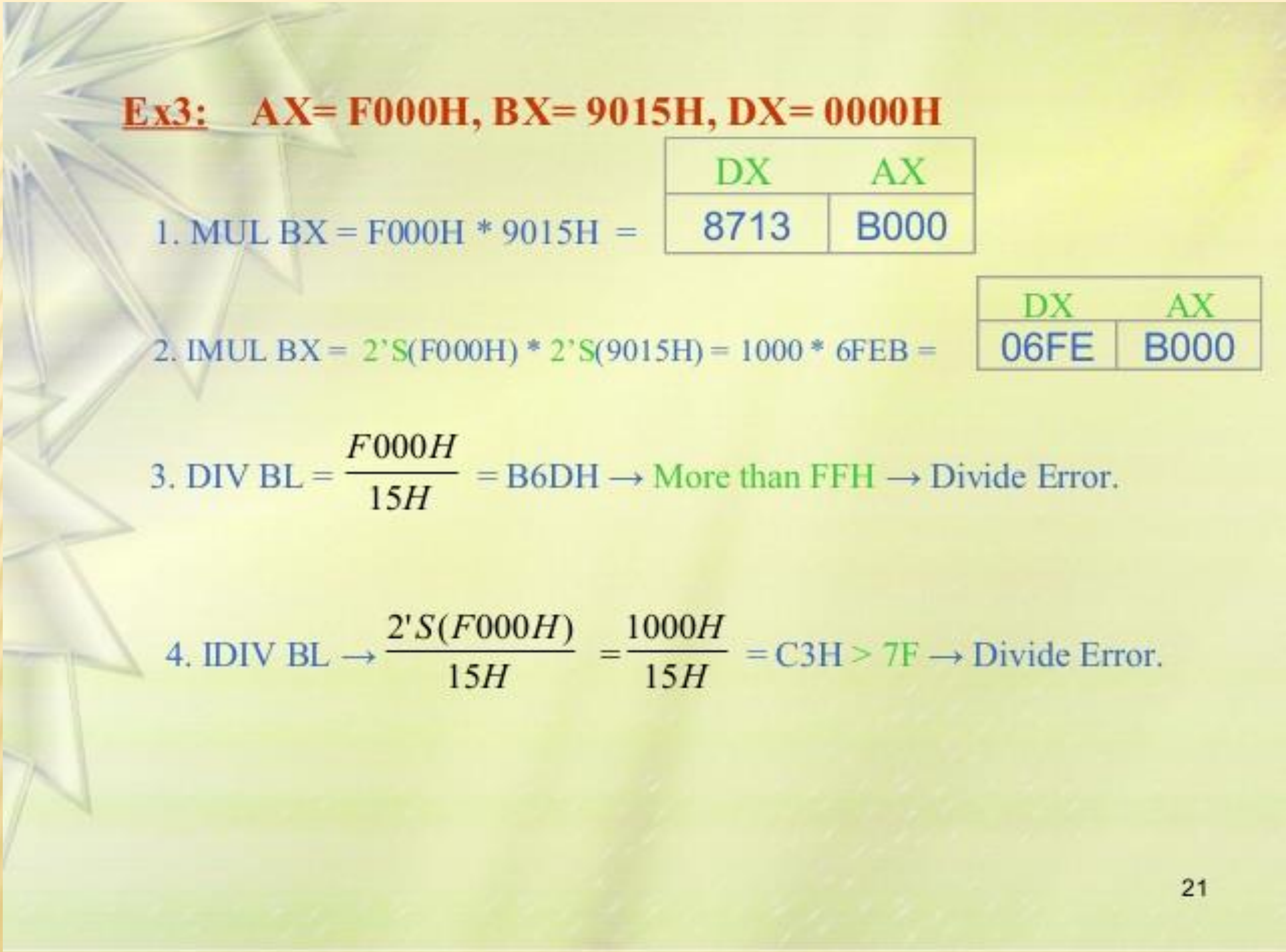
AH	AL
15	02
R	Q

→ $\frac{POS}{NEG} = NEG$ → 2's(02) = FEH→

AH	AL
15	FE

4. DIV BL → $\frac{AX}{BL} = \frac{00F3H}{91H}$ = 01→(F3-1*91=62) →

AH	AL
62	01
R	Q



Ex3: **AX= F000H, BX= 9015H, DX= 0000H**

1. MUL BX = F000H * 9015H =

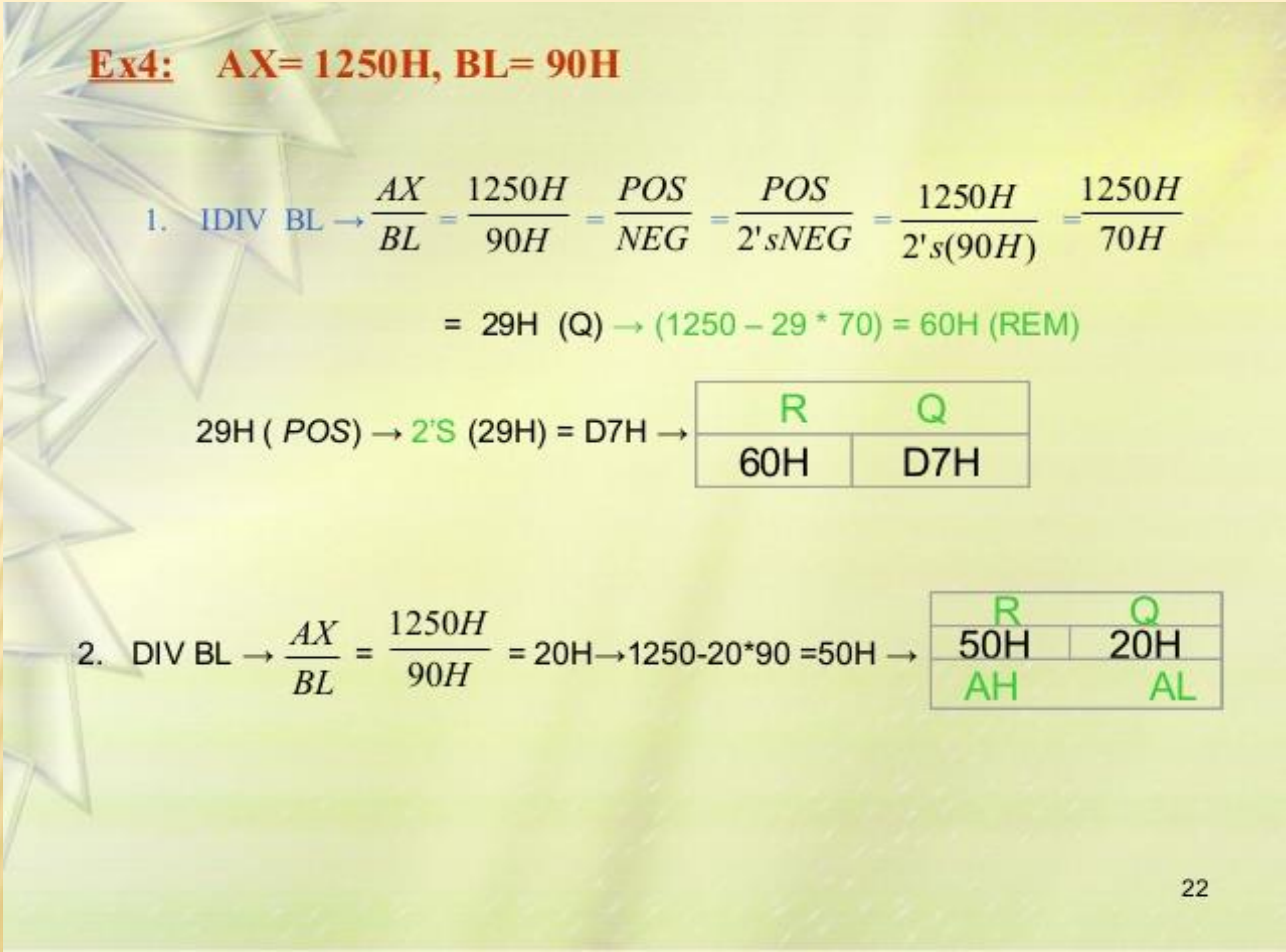
DX	AX
8713	B000

2. IMUL BX = 2'S(F000H) * 2'S(9015H) = 1000 * 6FEB =

DX	AX
06FE	B000

3. DIV BL = $\dfrac{F000H}{15H}$ = B6DH → More than FFH → Divide Error.

4. IDIV BL → $\dfrac{2'S(F000H)}{15H} = \dfrac{1000H}{15H}$ = C3H > 7F → Divide Error.



Ex4: **AX= 1250H, BL= 90H**

1. IDIV BL → $\frac{AX}{BL} = \frac{1250H}{90H} = \frac{POS}{NEG} = \frac{POS}{2'sNEG} = \frac{1250H}{2's(90H)} = \frac{1250H}{70H}$

$= 29H\ (Q) \rightarrow (1250 - 29 * 70) = 60H\ (REM)$

29H (*POS*) → 2'S (29H) = D7H →

R	Q
60H	D7H

2. DIV BL → $\frac{AX}{BL} = \frac{1250H}{90H}$ = 20H→1250-20*90 =50H →

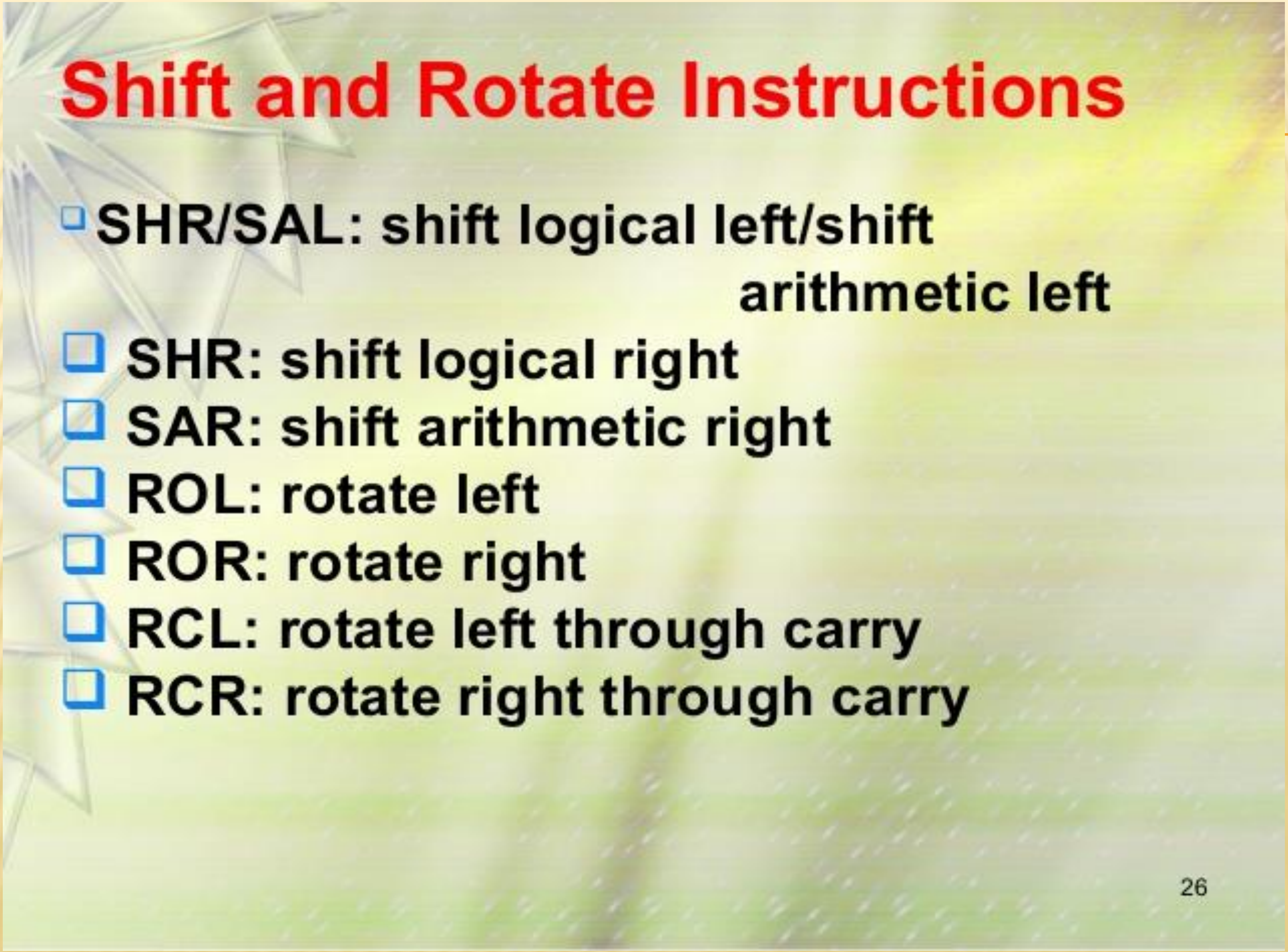
R	Q
50H	20H
AH	AL

Logical Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND D,S	(S) · (D) → (D)	OF, SF, ZF, PF, CF AF undefined
OR	Logical Inclusive OR	OR D,S	(S)+(D) → (D)	OF, SF, ZF, PF, CF AF undefined
XOR	Logical Exclusive OR	XOR D,S	(S)⊕ (D)→(D)	OF, SF, ZF, PF, CF AF undefined
NOT	LOGICAL NOT	NOT D	$\overline{(D)}$ → (D)	None

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

Destination
Register
Memory

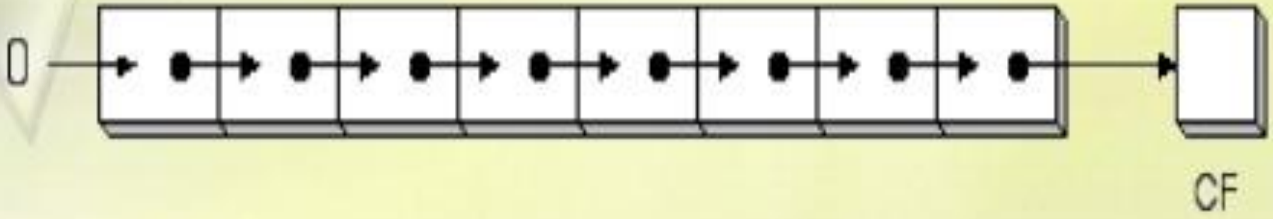


Shift and Rotate Instructions

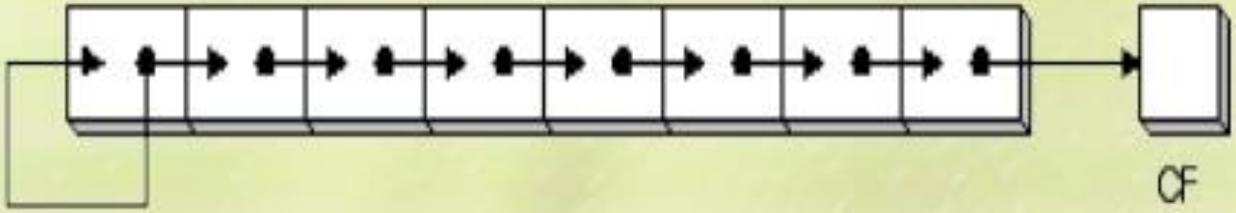
- SHR/SAL: shift logical left/shift arithmetic left
- □ SHR: shift logical right
- □ SAR: shift arithmetic right
- □ ROL: rotate left
- □ ROR: rotate right
- □ RCL: rotate left through carry
- □ RCR: rotate right through carry

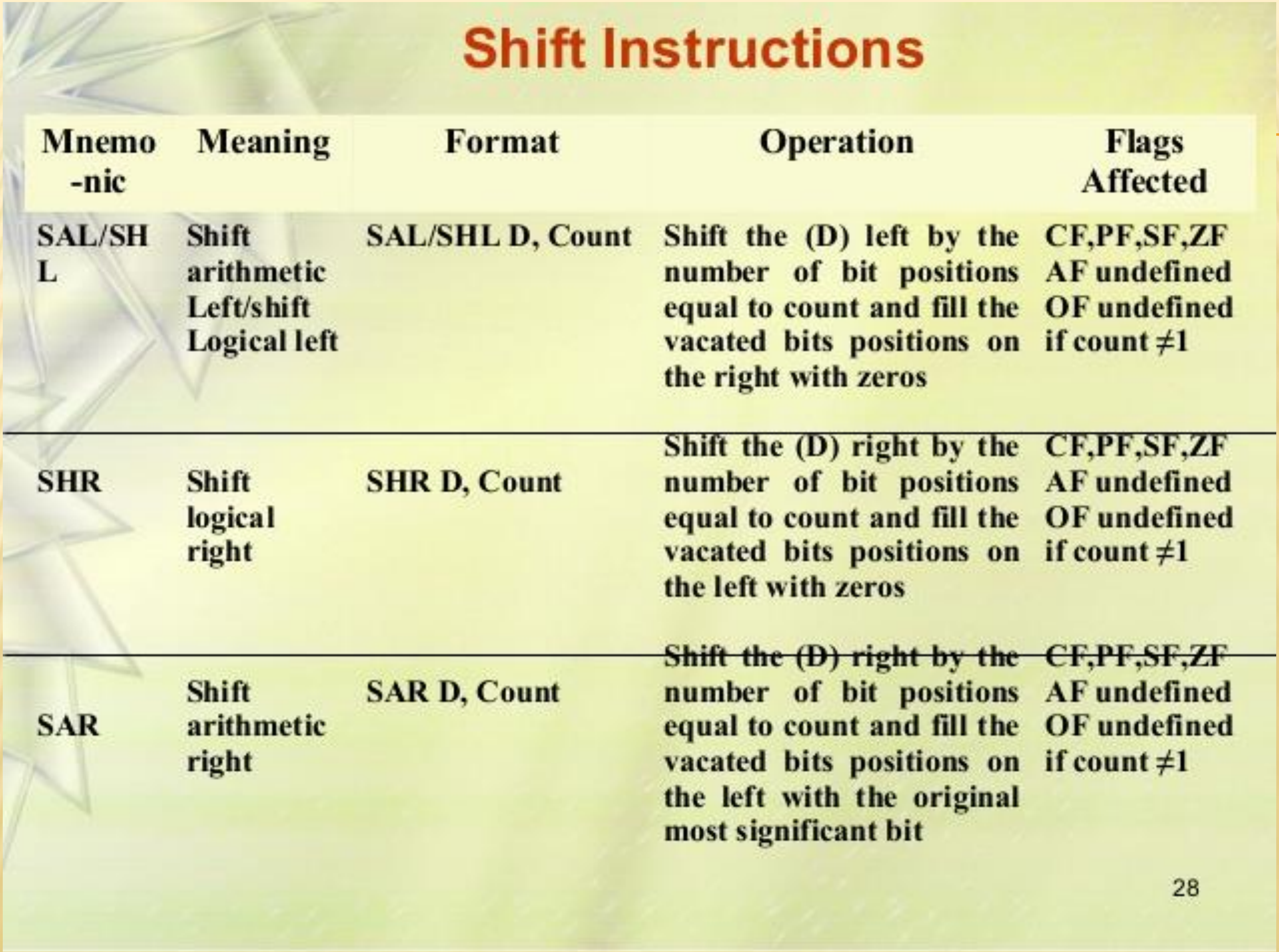
Logical vs Arithmetic Shifts

- A logical shift fills the newly created bit position with zero:



- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



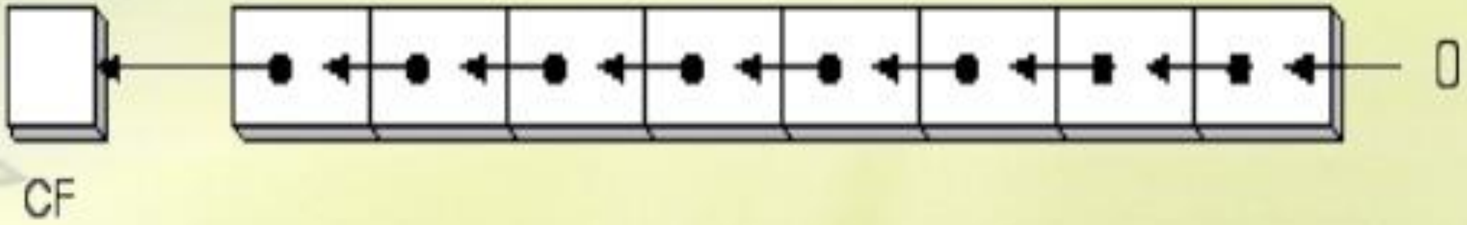


Shift Instructions

Mnemo -nic	Meaning	Format	Operation	Flags Affected
SAL/SHL	Shift arithmetic Left/shift Logical left	SAL/SHL D, Count	Shift the (D) left by the number of bit positions equal to count and fill the vacated bits positions on the right with zeros	CF,PF,SF,ZF AF undefined OF undefined if count ≠1
SHR	Shift logical right	SHR D, Count	Shift the (D) right by the number of bit positions equal to count and fill the vacated bits positions on the left with zeros	CF,PF,SF,ZF AF undefined OF undefined if count ≠1
SAR	Shift arithmetic right	SAR D, Count	Shift the (D) right by the number of bit positions equal to count and fill the vacated bits positions on the left with the original most significant bit	CF,PF,SF,ZF AF undefined OF undefined if count ≠1

SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.



- Operand types:

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

Fast Multiplication

Shifting left 1 bit multiplies a number by 2

```
mov dl,5

shl dl,1
```

Before: 00000101 = 5

After: 00001010 = 10

Shifting left *n* bits multiplies the operand by

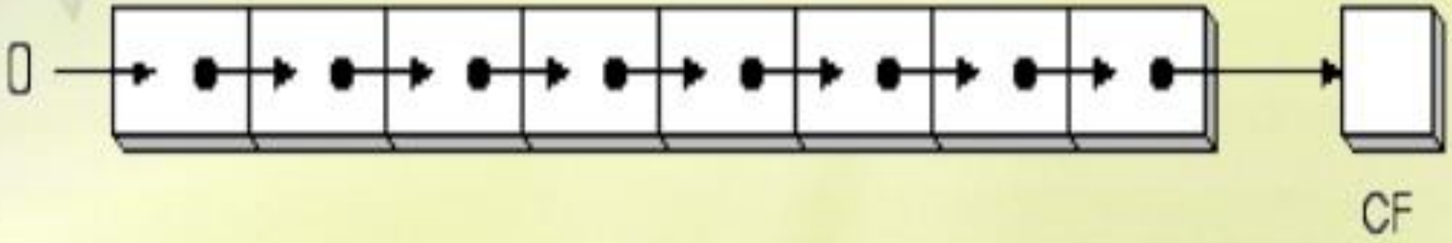
2^n

For example, $5 * 2^2 = 20$

```
mov dl,5
shl dl,2                ; DL = 20
```

SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

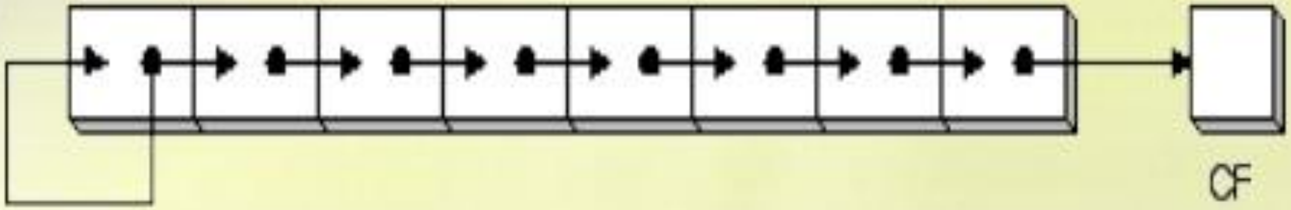


Shifting right *n* bits divides the operand by 2^n

```
MOV DL,80
SHR DL,1                      ; DL = 40
SHR DL,2                      ; DL = 10
```

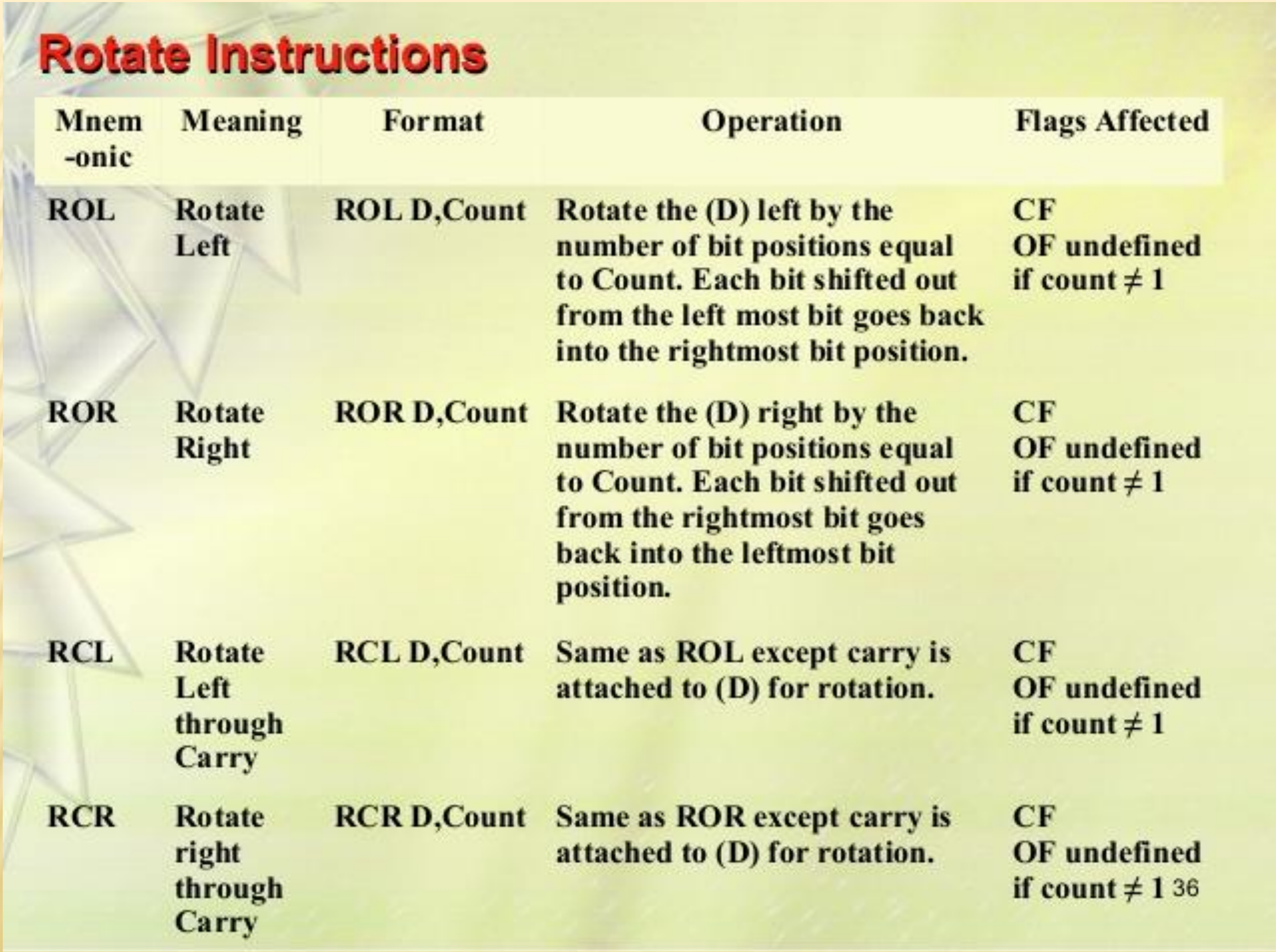
SAR Instruction

- **SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.**



An arithmetic shift preserves the number's sign.

```
MOV DL,-80
SAR DL,1                          ; DL = -40
SAR DL,2                          ; DL = -10
```

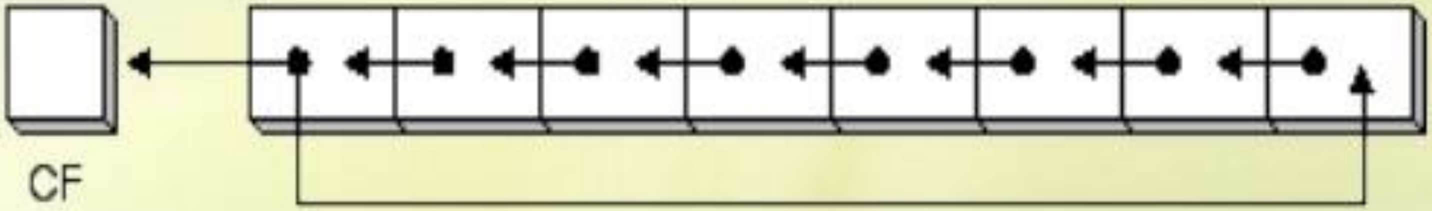


Rotate Instructions

Mnem -onic	Meaning	Format	Operation	Flags Affected
ROL	Rotate Left	ROL D,Count	Rotate the (D) left by the number of bit positions equal to Count. Each bit shifted out from the left most bit goes back into the rightmost bit position.	CF OF undefined if count ≠ 1
ROR	Rotate Right	ROR D,Count	Rotate the (D) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost bit goes back into the leftmost bit position.	CF OF undefined if count ≠ 1
RCL	Rotate Left through Carry	RCL D,Count	Same as ROL except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1
RCR	Rotate right through Carry	RCR D,Count	Same as ROR except carry is attached to (D) for rotation.	CF OF undefined if count ≠ 1

ROL Instruction

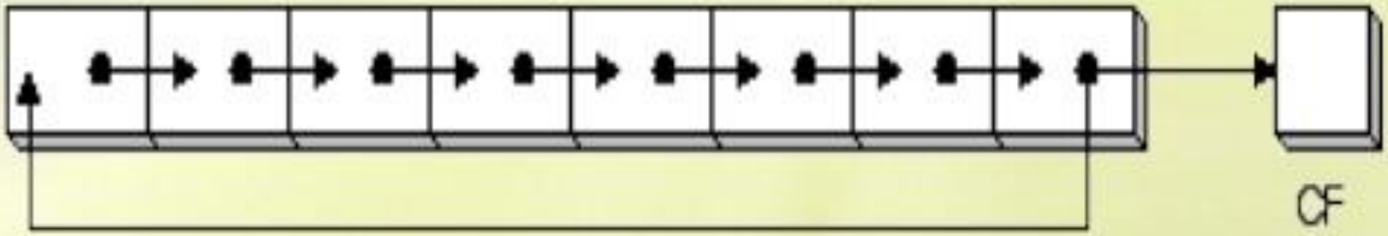
- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost



```
MOV Al,11110000b
ROL Al,1                    ; AL = 1100001b

MOV Dl,3Fh
ROL Dl,4                    ; DL = F3h
```

ROR Instruction

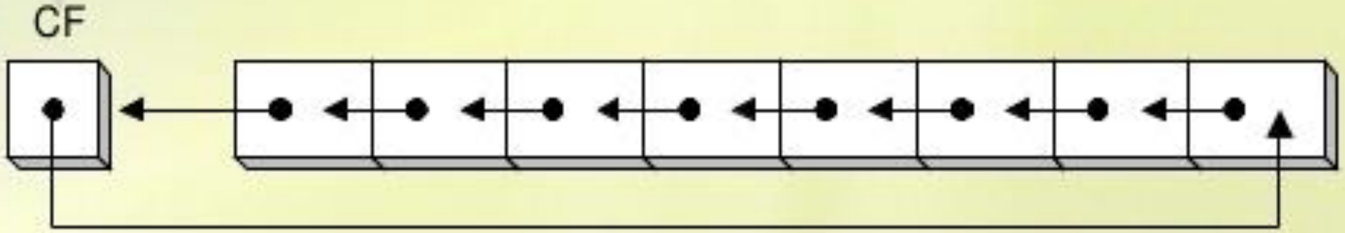
- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost



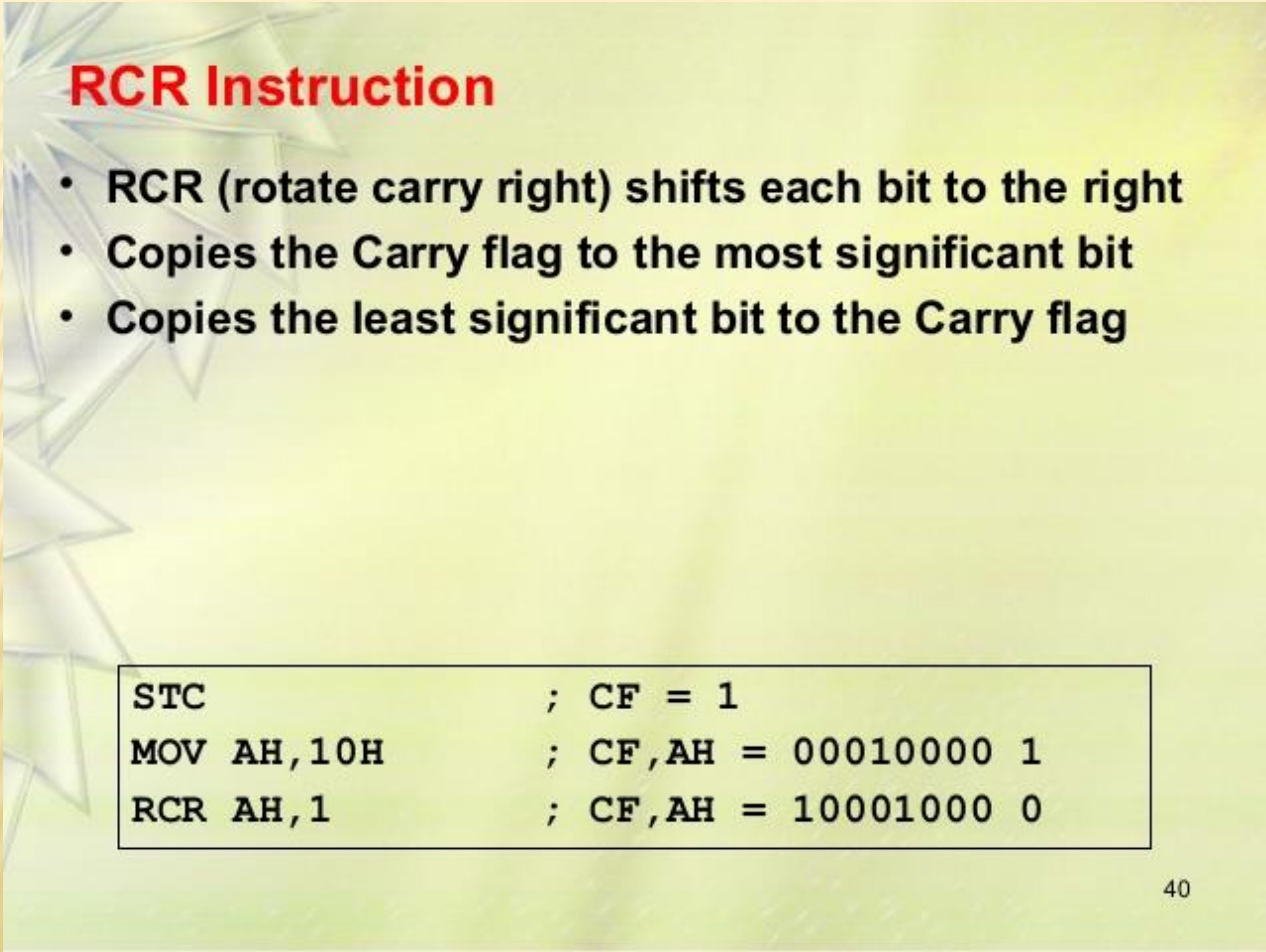
```
MOV AL,11110000b
ROR AL,1                    ; AL = 01111000b

MOV DL,3Fh
ROR DL,4                    ; DL = F3h
```

RCL Instruction

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag



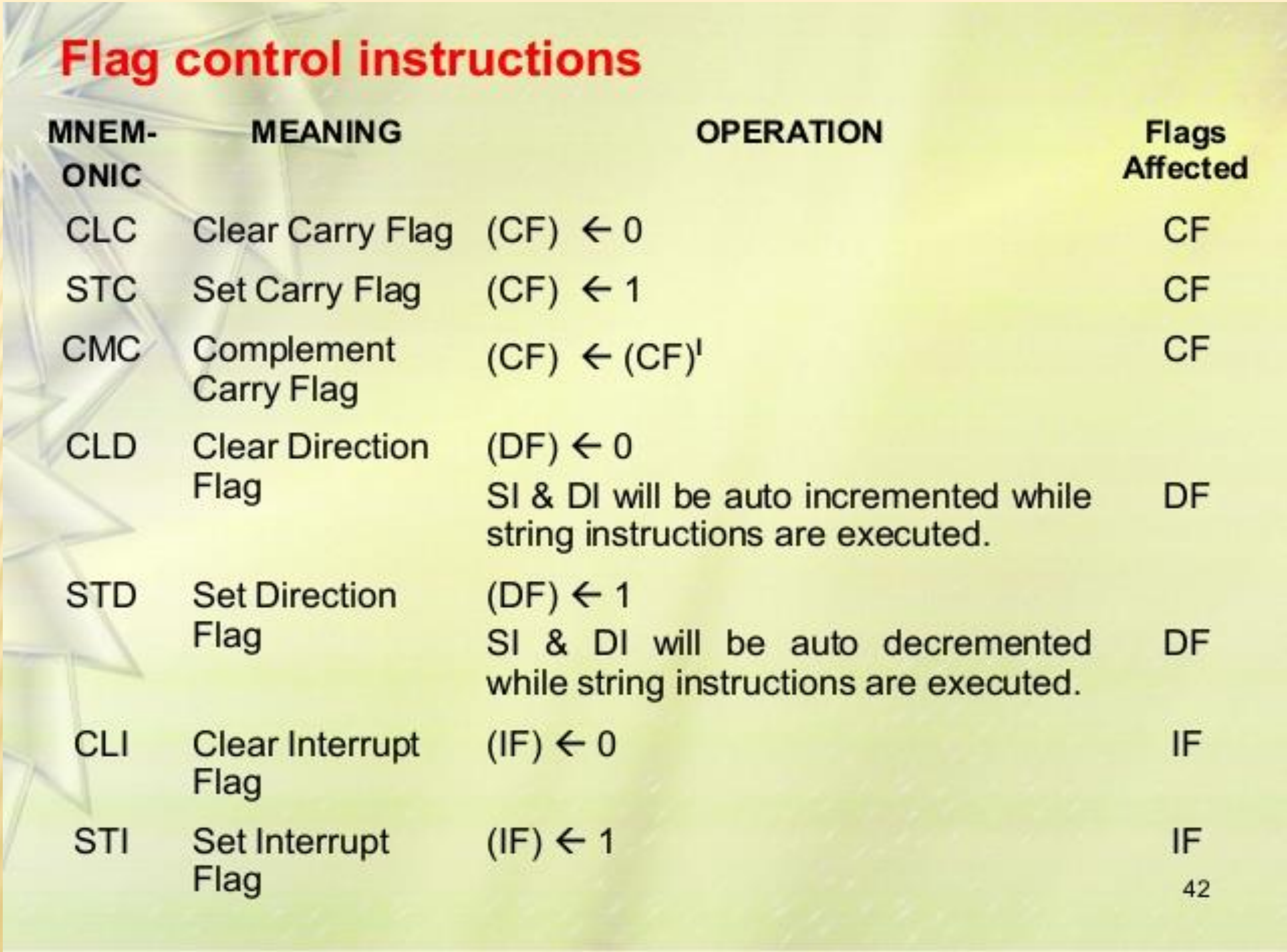
```
CLC                 ; CF = 0
MOV BL,88H          ; CF,BL = 0 10001000b
RCL BL,1            ; CF,BL = 1 00010000b
RCL BL,1            ; CF,BL = 0 00100001b
```



RCR Instruction

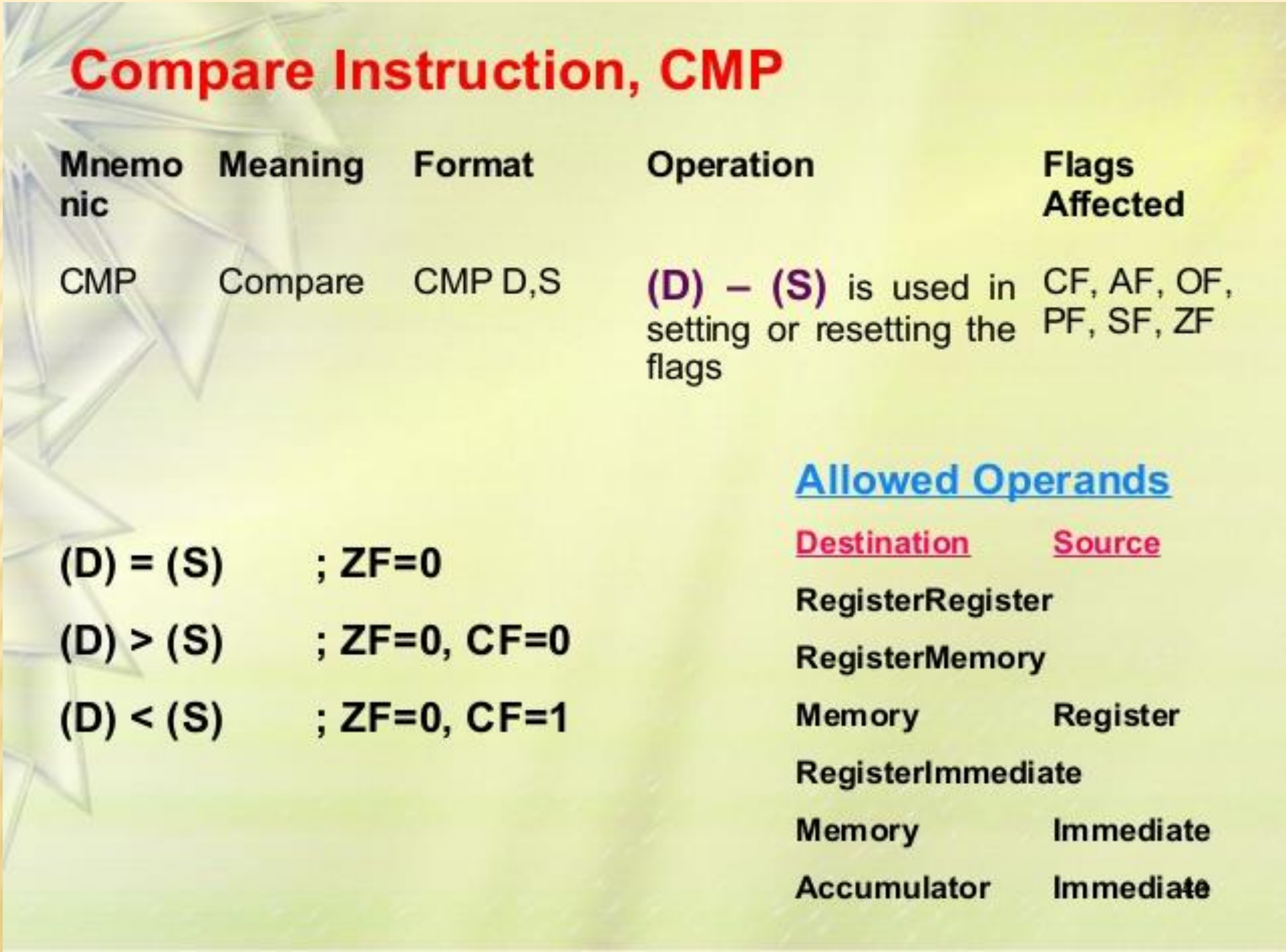
- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag

```
STC                 ; CF = 1
MOV AH,10H          ; CF,AH = 00010000 1
RCR AH,1            ; CF,AH = 10001000 0
```



Flag control instructions

MNEM- ONIC	MEANING	OPERATION	Flags Affected
CLC	Clear Carry Flag	(CF) ← 0	CF
STC	Set Carry Flag	(CF) ← 1	CF
CMC	Complement Carry Flag	(CF) ← (CF)^{I}	CF
CLD	Clear Direction Flag	(DF) ← 0 SI & DI will be auto incremented while string instructions are executed.	DF
STD	Set Direction Flag	(DF) ← 1 SI & DI will be auto decremented while string instructions are executed.	DF
CLI	Clear Interrupt Flag	(IF) ← 0	IF
STI	Set Interrupt Flag	(IF) ← 1	IF



Compare Instruction, CMP

Mnemonic	Meaning	Format	Operation	Flags Affected
CMP	Compare	CMP D,S	(D) – (S) is used in setting or resetting the flags	CF, AF, OF, PF, SF, ZF

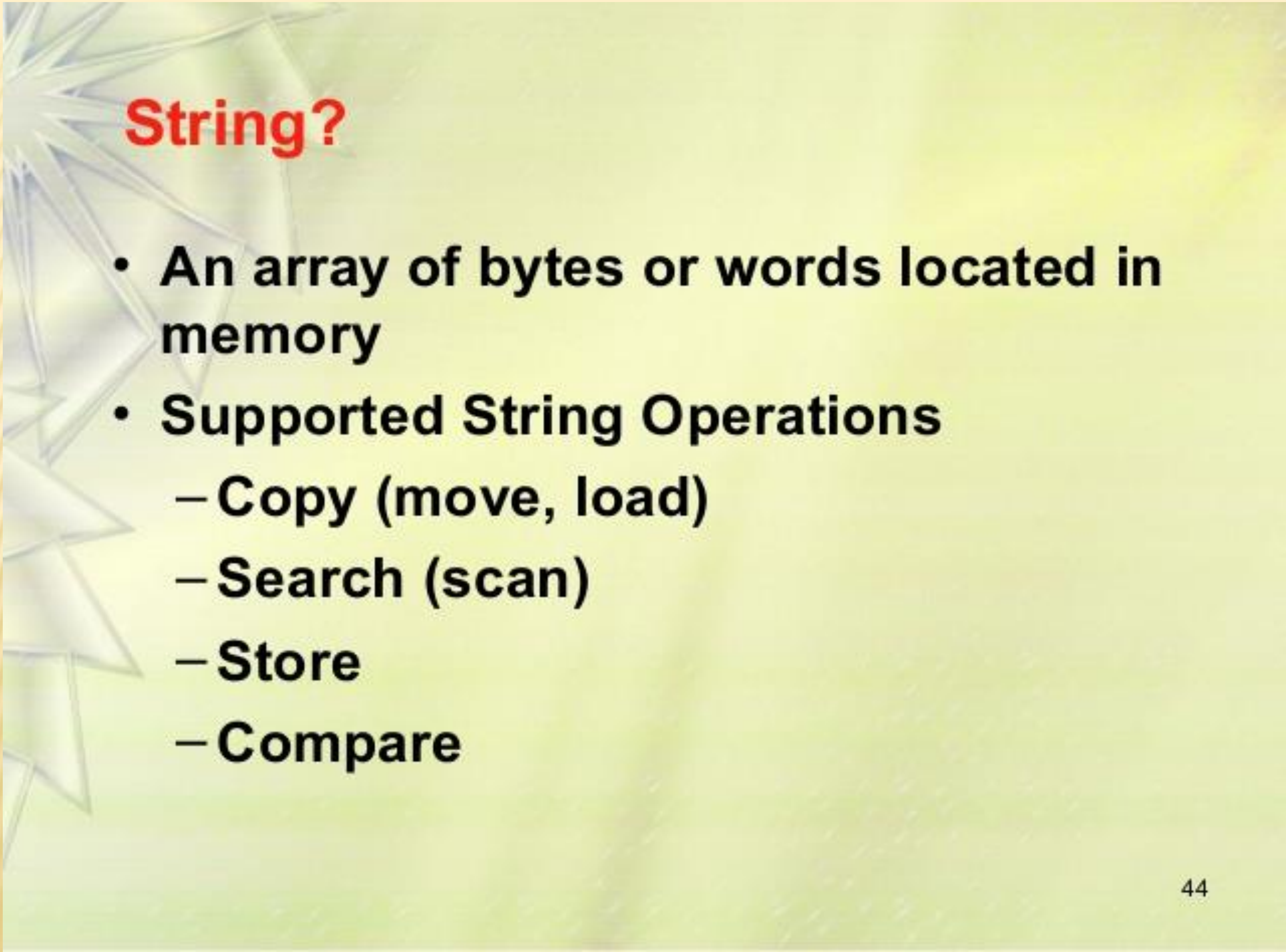
(D) = (S) ; ZF=0

(D) > (S) ; ZF=0, CF=0

(D) < (S) ; ZF=0, CF=1

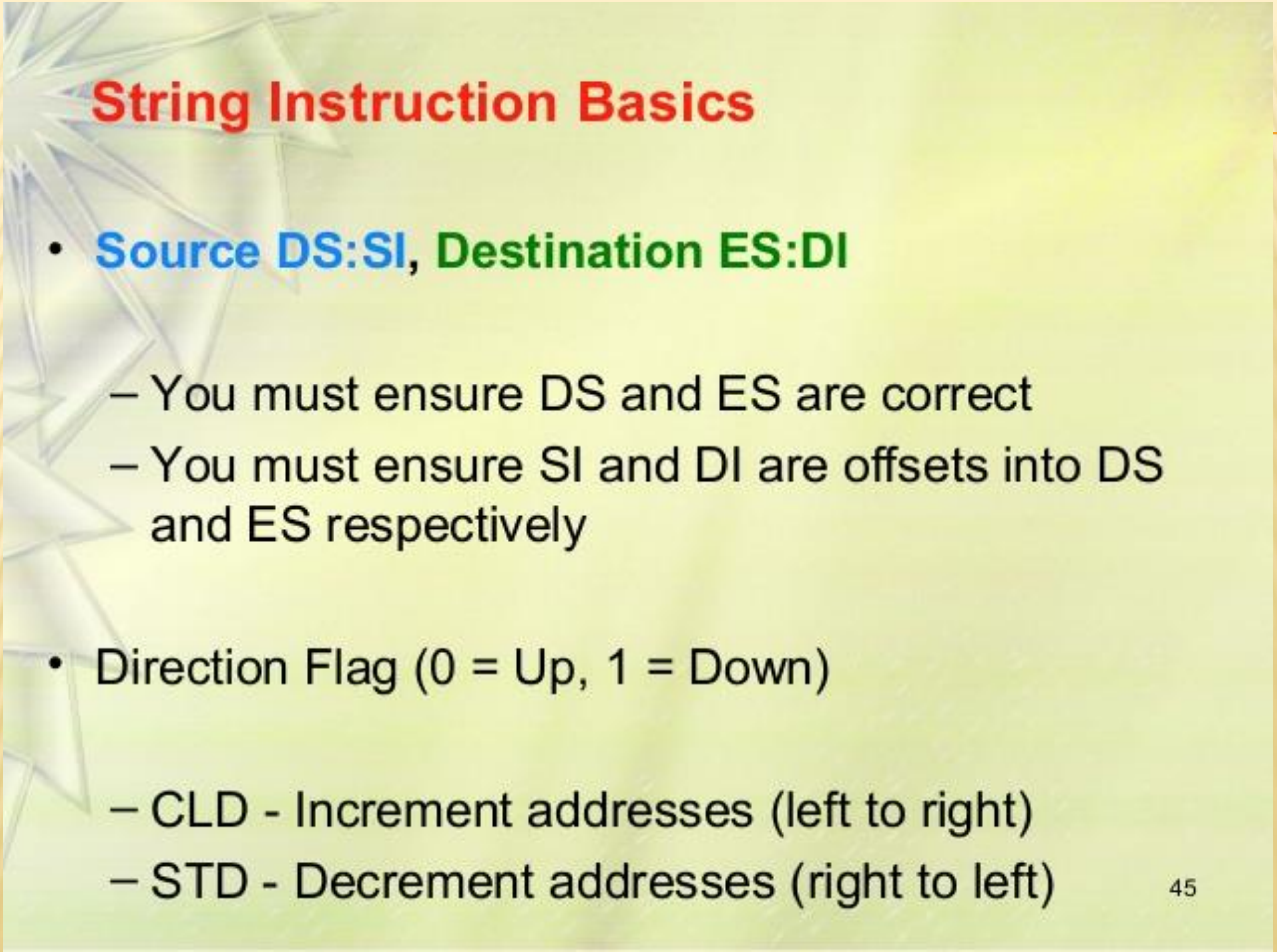
Allowed Operands

Destination	Source
RegisterRegister	
RegisterMemory	
Memory	Register
RegisterImmediate	
Memory	Immediate
Accumulator	Immediate



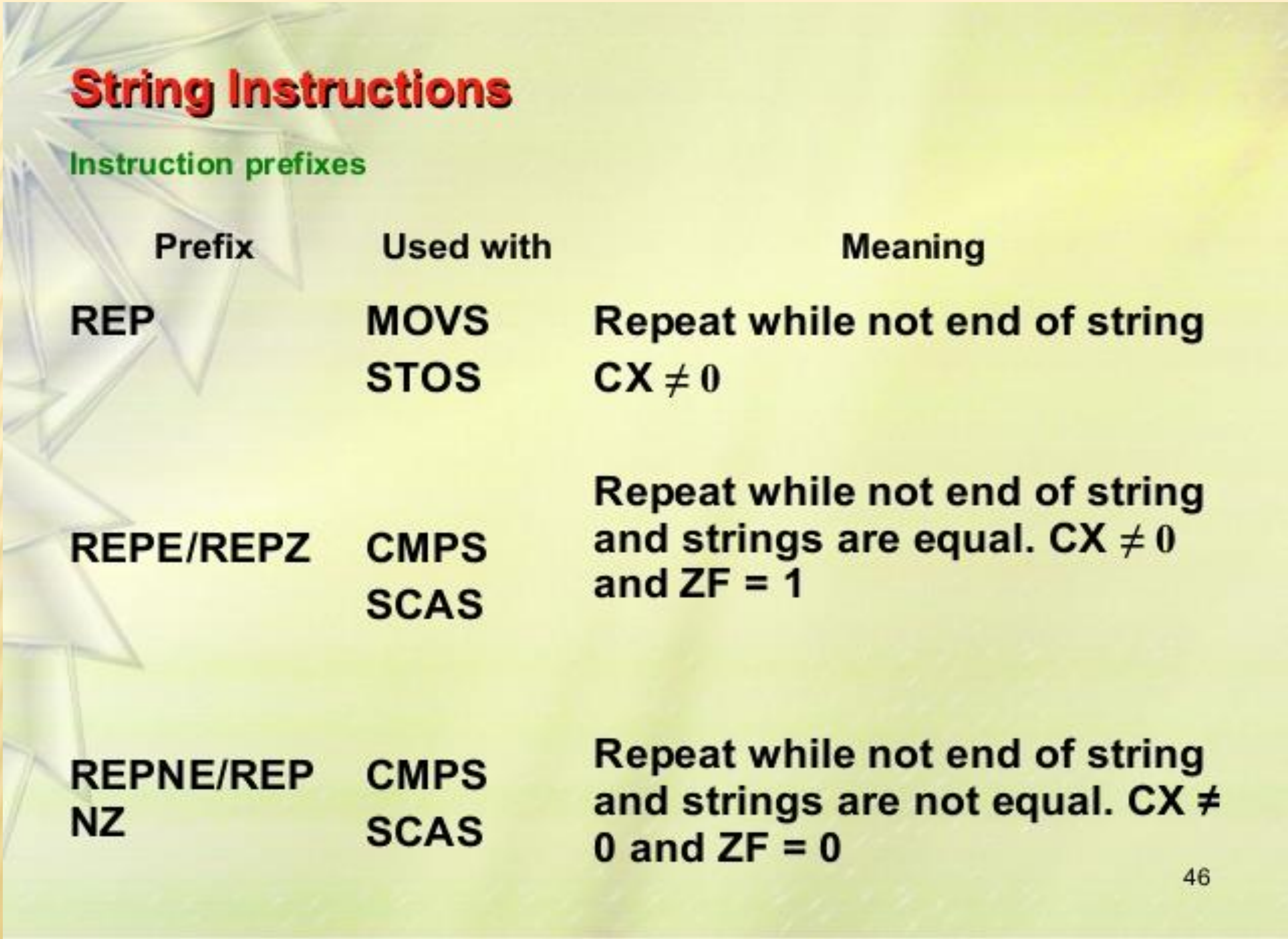
String?

- An array of bytes or words located in memory
- Supported String Operations
  - Copy (move, load)
  - Search (scan)
  - Store
  - Compare



String Instruction Basics

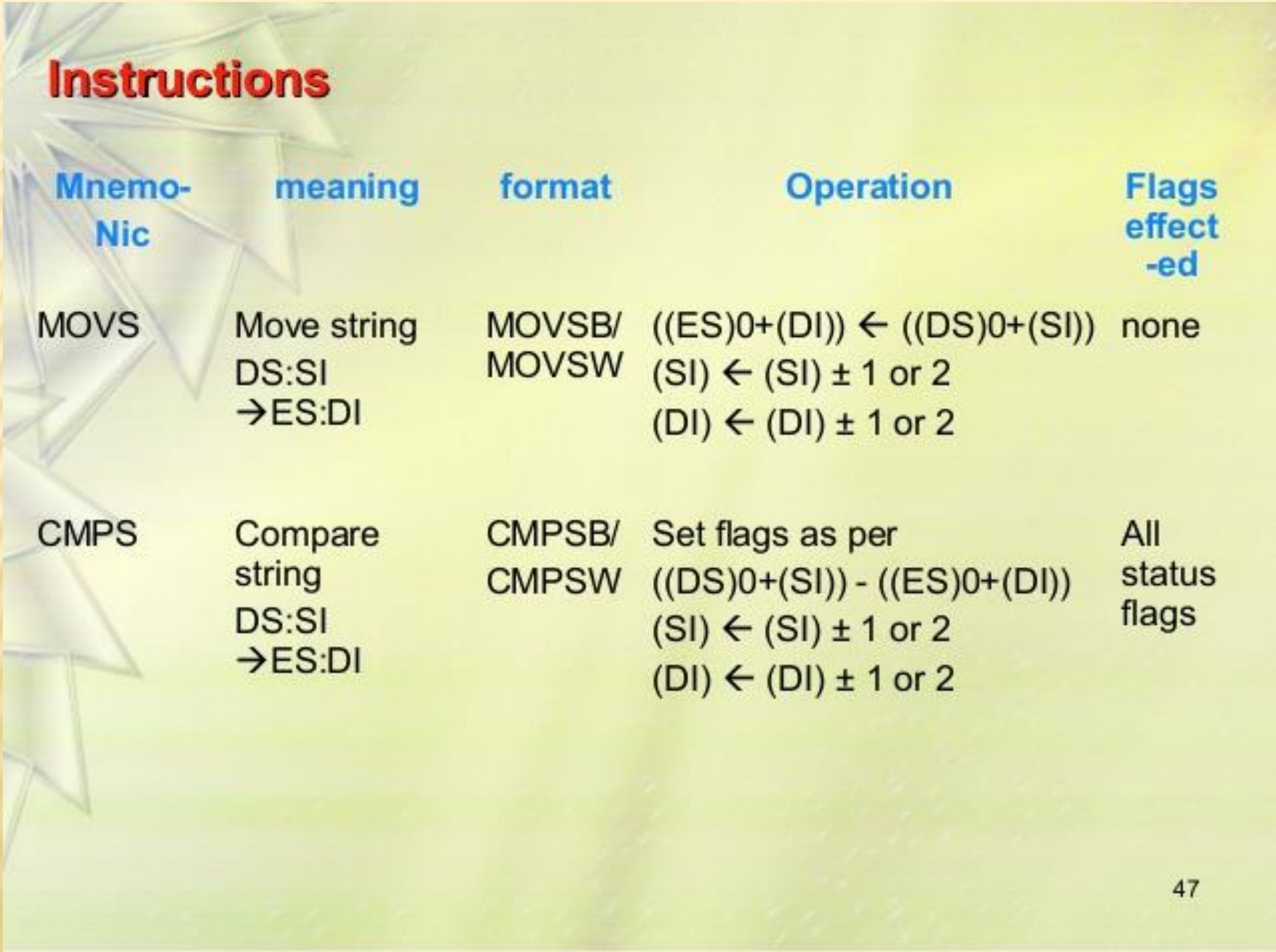
- Source DS:SI, Destination ES:DI
  - You must ensure DS and ES are correct
  - You must ensure SI and DI are offsets into DS and ES respectively
- Direction Flag (0 = Up, 1 = Down)
  - CLD - Increment addresses (left to right)
  - STD - Decrement addresses (right to left)



String Instructions

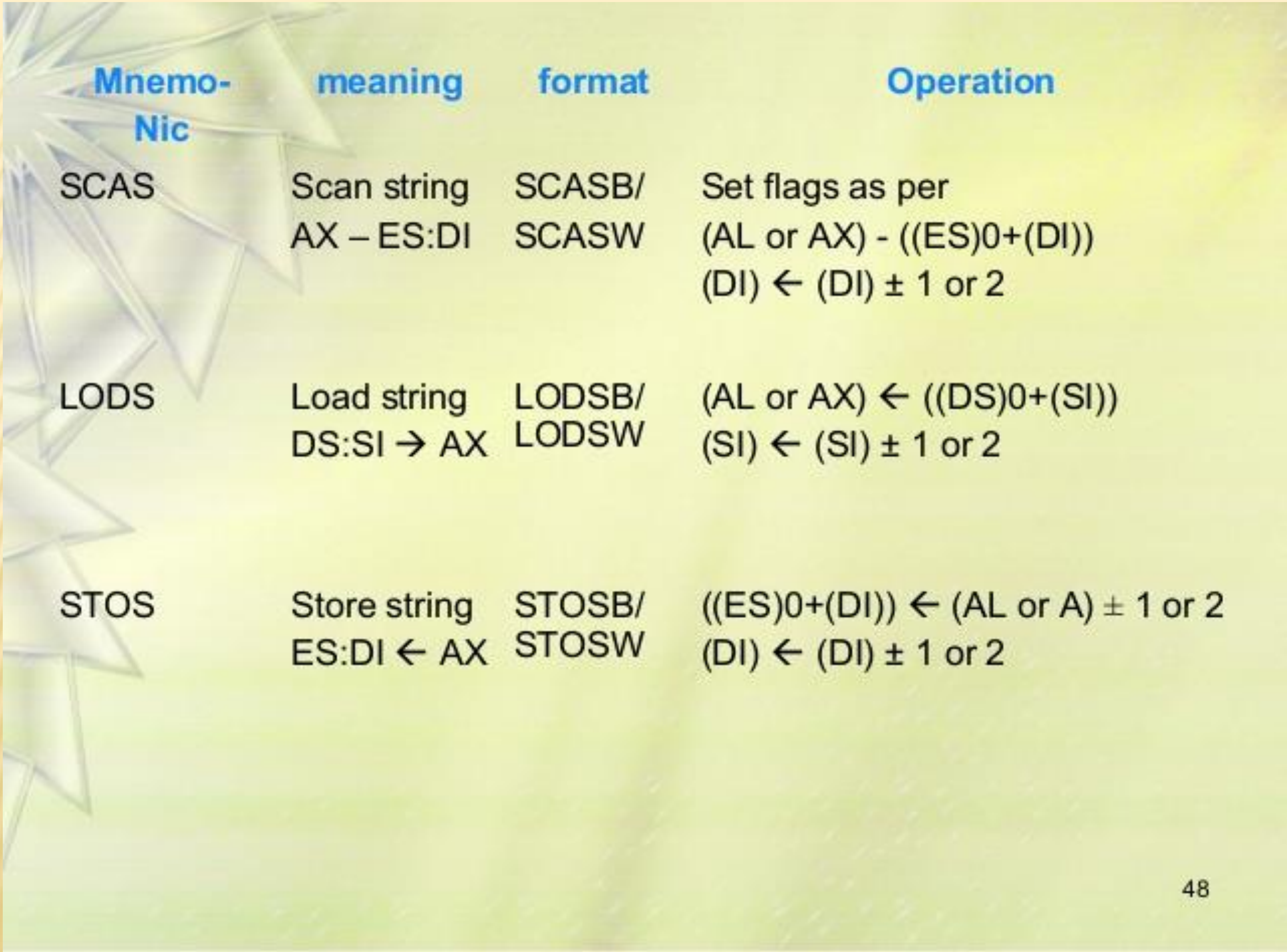
Instruction prefixes

Prefix	Used with	Meaning
REP	MOVS STOS	Repeat while not end of string $CX \neq 0$
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal. $CX \neq 0$ and $ZF = 1$
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal. $CX \neq 0$ and $ZF = 0$

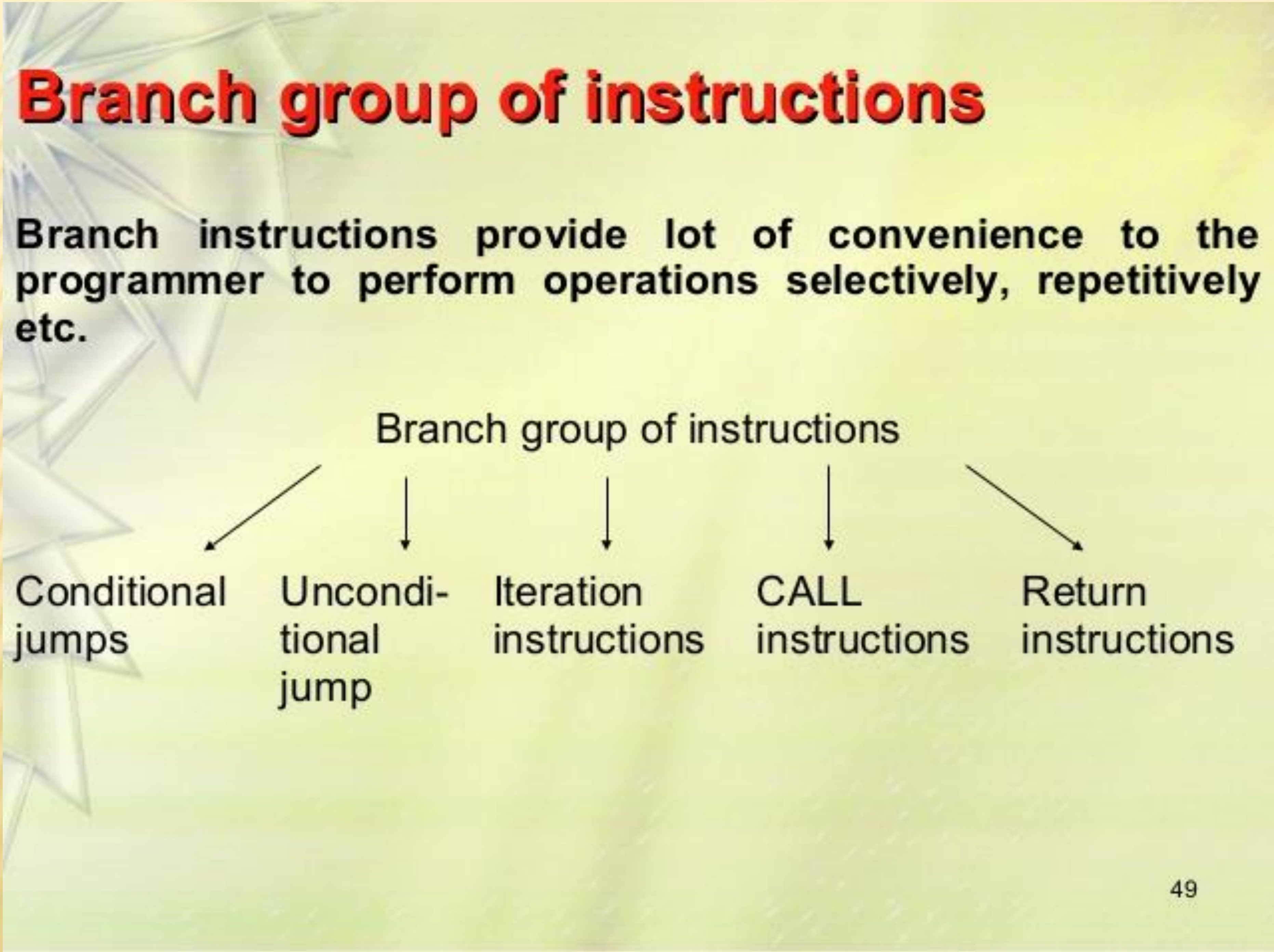


Instructions

MnemoNic	meaning	format	Operation	Flags effected
MOVS	Move string DS:SI →ES:DI	MOVSB/ MOVSW	((ES)0+(DI)) ← ((DS)0+(SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	none
CMPS	Compare string DS:SI →ES:DI	CMPSB/ CMPSW	Set flags as per ((DS)0+(SI)) - ((ES)0+(DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	All status flags



Mnemo- Nic	meaning	format	Operation
SCAS	Scan string AX – ES:DI	SCASB/ SCASW	Set flags as per (AL or AX) - ((ES)0+(DI)) (DI) ← (DI) ± 1 or 2
LODS	Load string DS:SI → AX	LODSB/ LODSW	(AL or AX) ← ((DS)0+(SI)) (SI) ← (SI) ± 1 or 2
STOS	Store string ES:DI ← AX	STOSB/ STOSW	((ES)0+(DI)) ← (AL or A) ± 1 or 2 (DI) ← (DI) ± 1 or 2



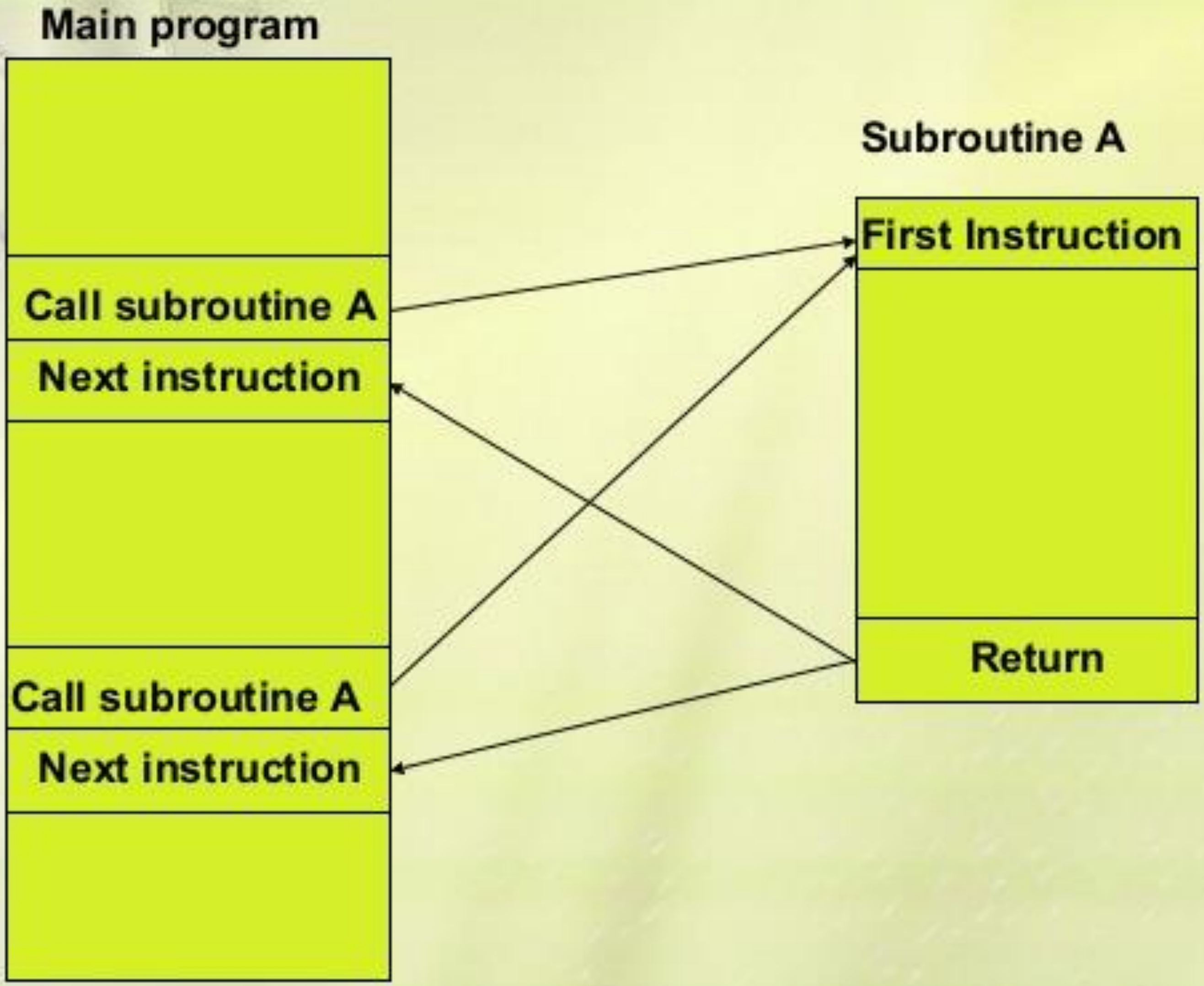
Branch group of instructions

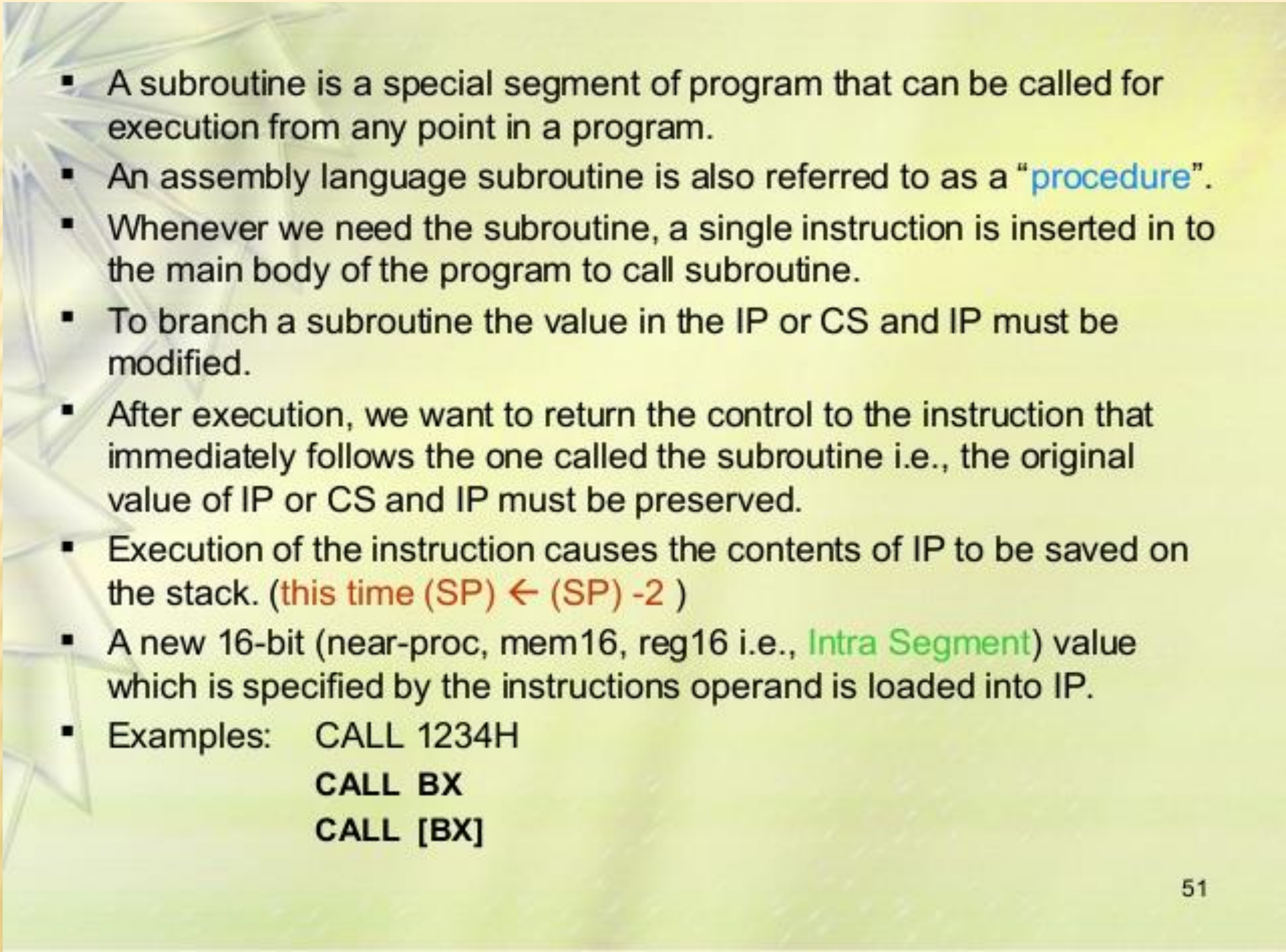
Branch instructions provide lot of convenience to the programmer to perform operations selectively, repetitively etc.

Branch group of instructions

- Conditional jumps
- Unconditional jump
- Iteration instructions
- CALL instructions
- Return instructions

SUBROUTINE & SUBROUTINE HANDILING INSTRUCTIONS





- A subroutine is a special segment of program that can be called for execution from any point in a program.
- An assembly language subroutine is also referred to as a “procedure”.
- Whenever we need the subroutine, a single instruction is inserted in to the main body of the program to call subroutine.
- To branch a subroutine the value in the IP or CS and IP must be modified.
- After execution, we want to return the control to the instruction that immediately follows the one called the subroutine i.e., the original value of IP or CS and IP must be preserved.
- Execution of the instruction causes the contents of IP to be saved on the stack. (this time (SP) ← (SP) -2)
- A new 16-bit (near-proc, mem16, reg16 i.e., Intra Segment) value which is specified by the instructions operand is loaded into IP.
- Examples:

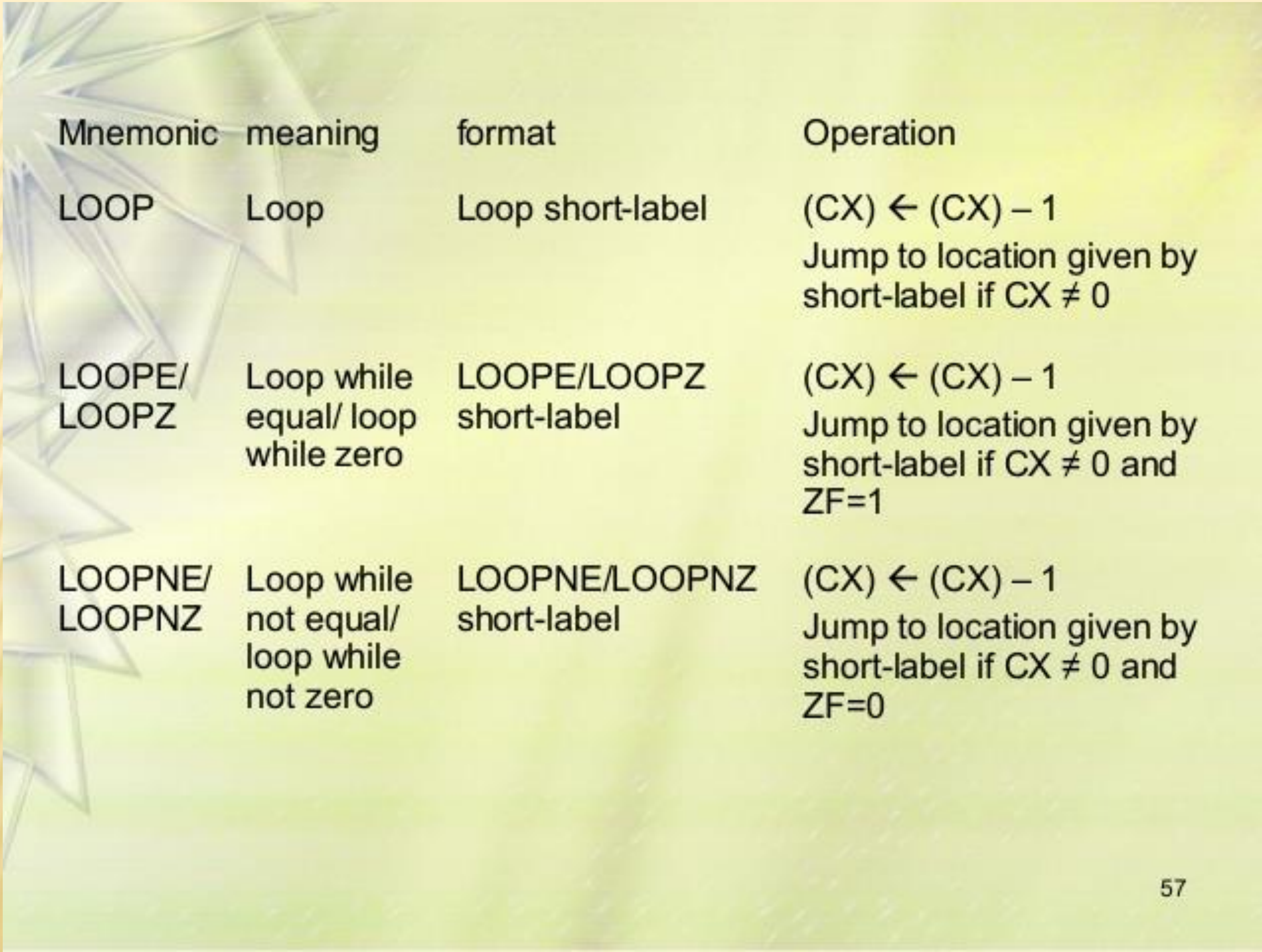
```
CALL 1234H
CALL  BX
CALL  [BX]
```

RETURN

- Every subroutine must end by executing an instruction that returns control to the main program. This is the return (RET) instruction.
- By execution the value of IP or IP and CS that were saved in the stack to be returned back to their corresponding regs. (this time (SP) ← (SP)+2)

Mnem-onic	Meaning	Format	Operation	Flags Affected
RET	Return	RET or RET operand	Return to the main program by restoring IP (and CS for far-proc). If operands is present, it is added to the contents of SP.	None

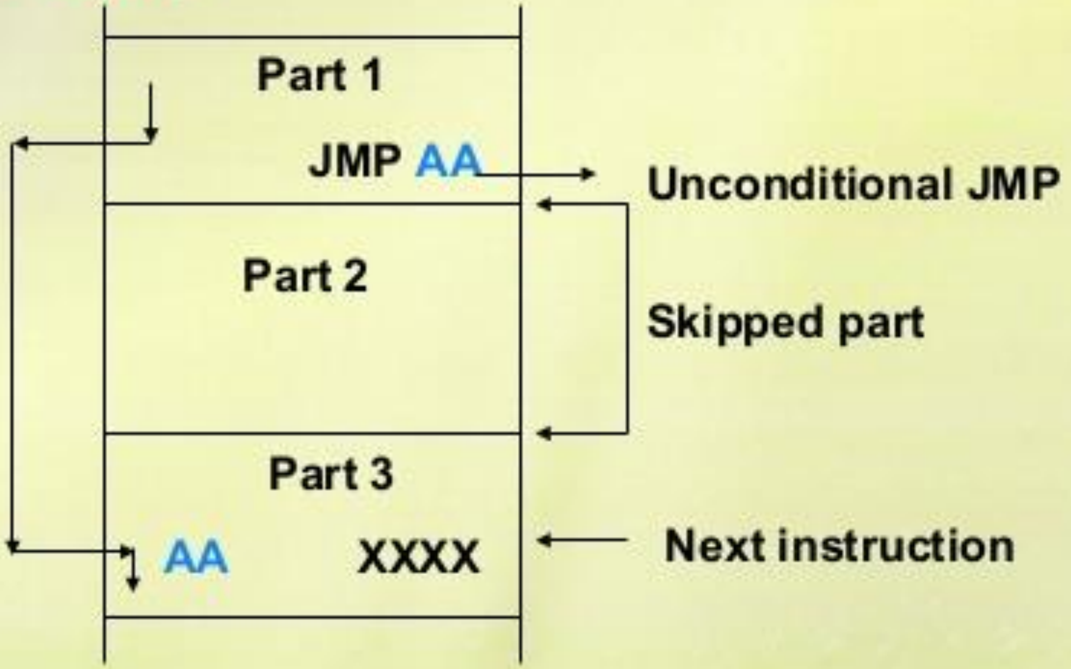
Operand
None
Disp16



Mnemonic	meaning	format	Operation
LOOP	Loop	Loop short-label	$(CX) \leftarrow (CX) – 1$ Jump to location given by short-label if $CX \neq 0$
LOOPE/ LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ short-label	$(CX) \leftarrow (CX) – 1$ Jump to location given by short-label if $CX \neq 0$ and $ZF=1$
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ short-label	$(CX) \leftarrow (CX) – 1$ Jump to location given by short-label if $CX \neq 0$ and $ZF=0$

Control flow and JUMP instructions

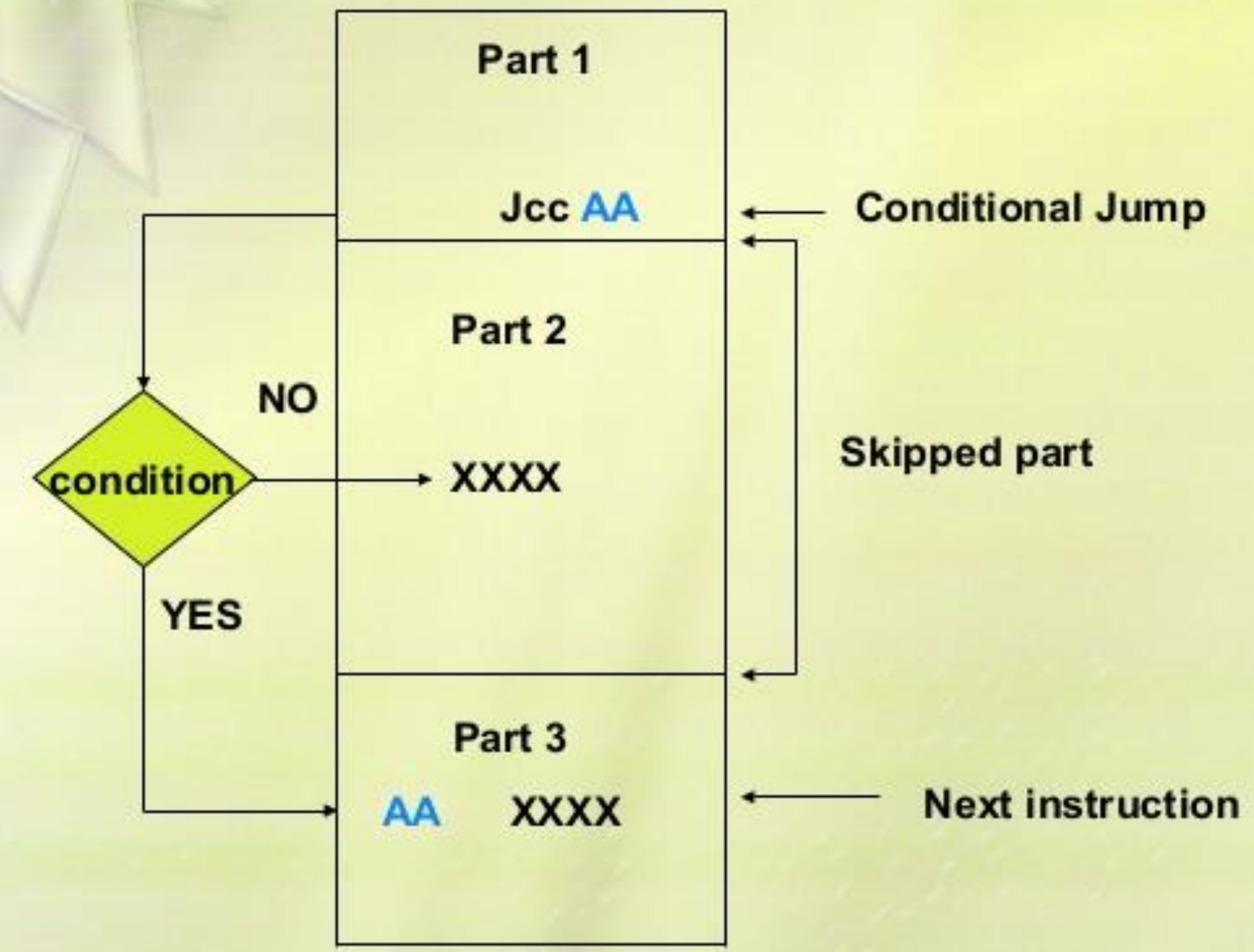
Unconditional Jump

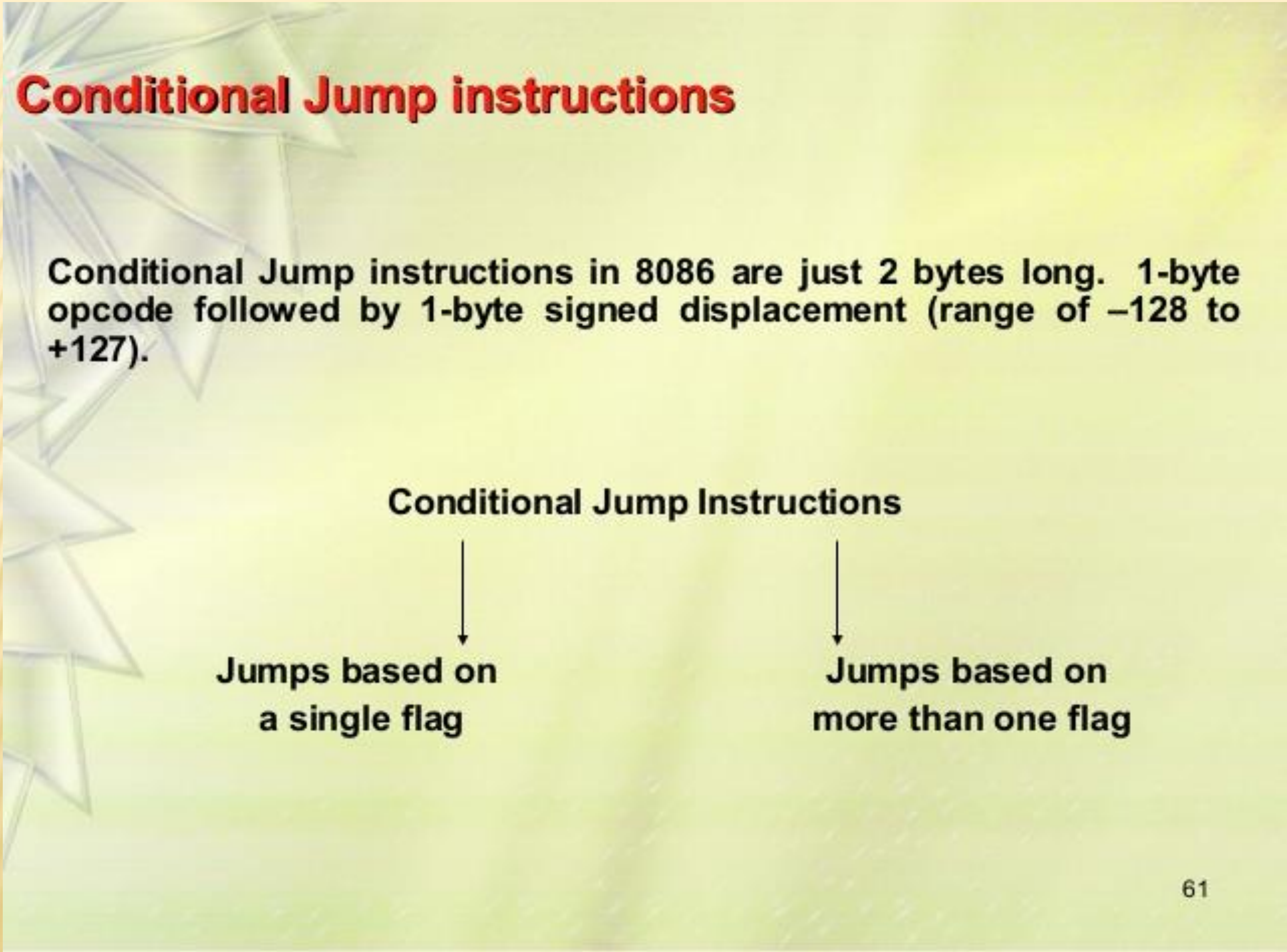


JMP → unconditional jump

JMP Operand

Conditional Jump





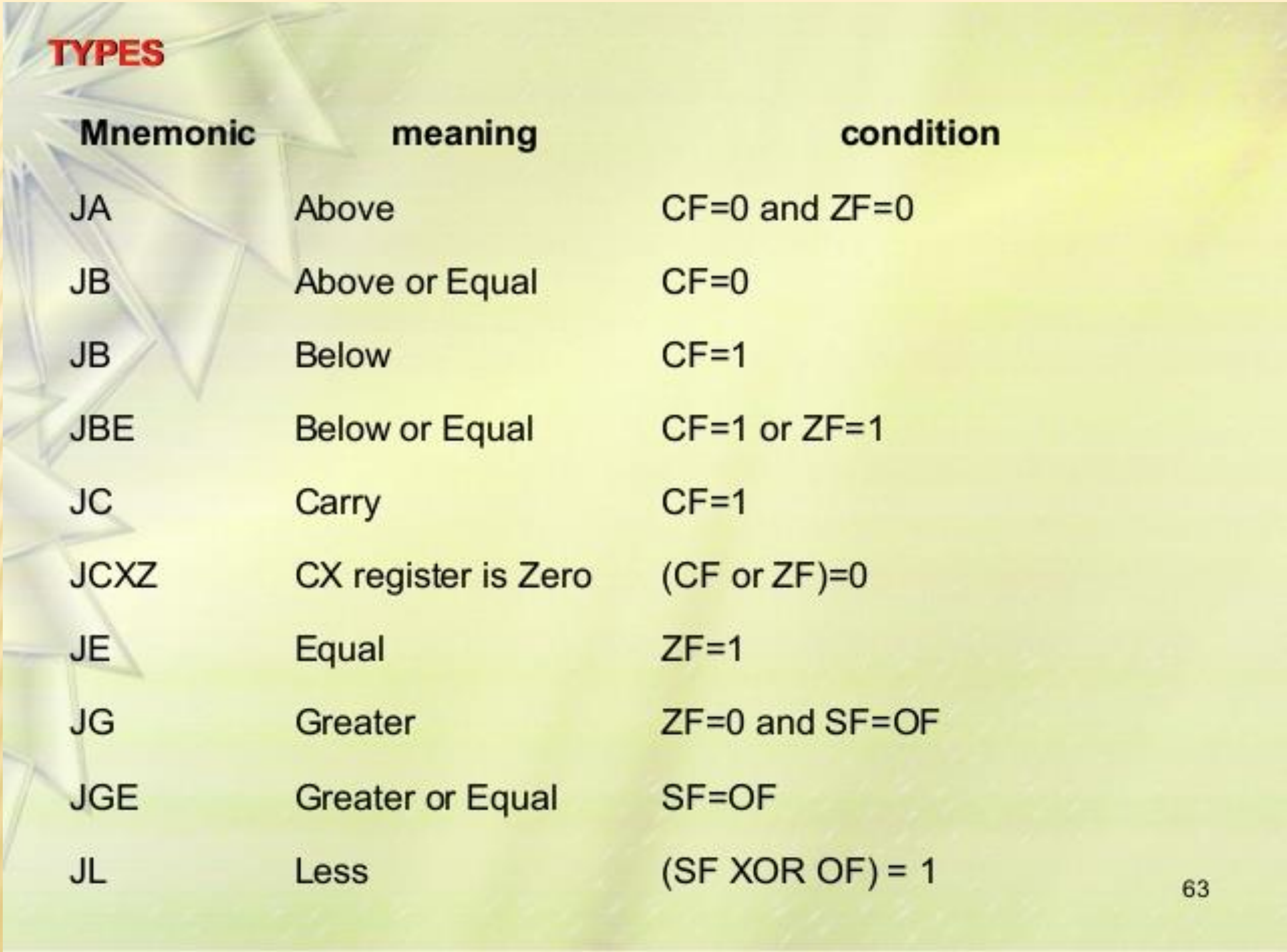
Conditional Jump instructions

Conditional Jump instructions in 8086 are just 2 bytes long. 1-byte opcode followed by 1-byte signed displacement (range of –128 to +127).

Conditional Jump Instructions

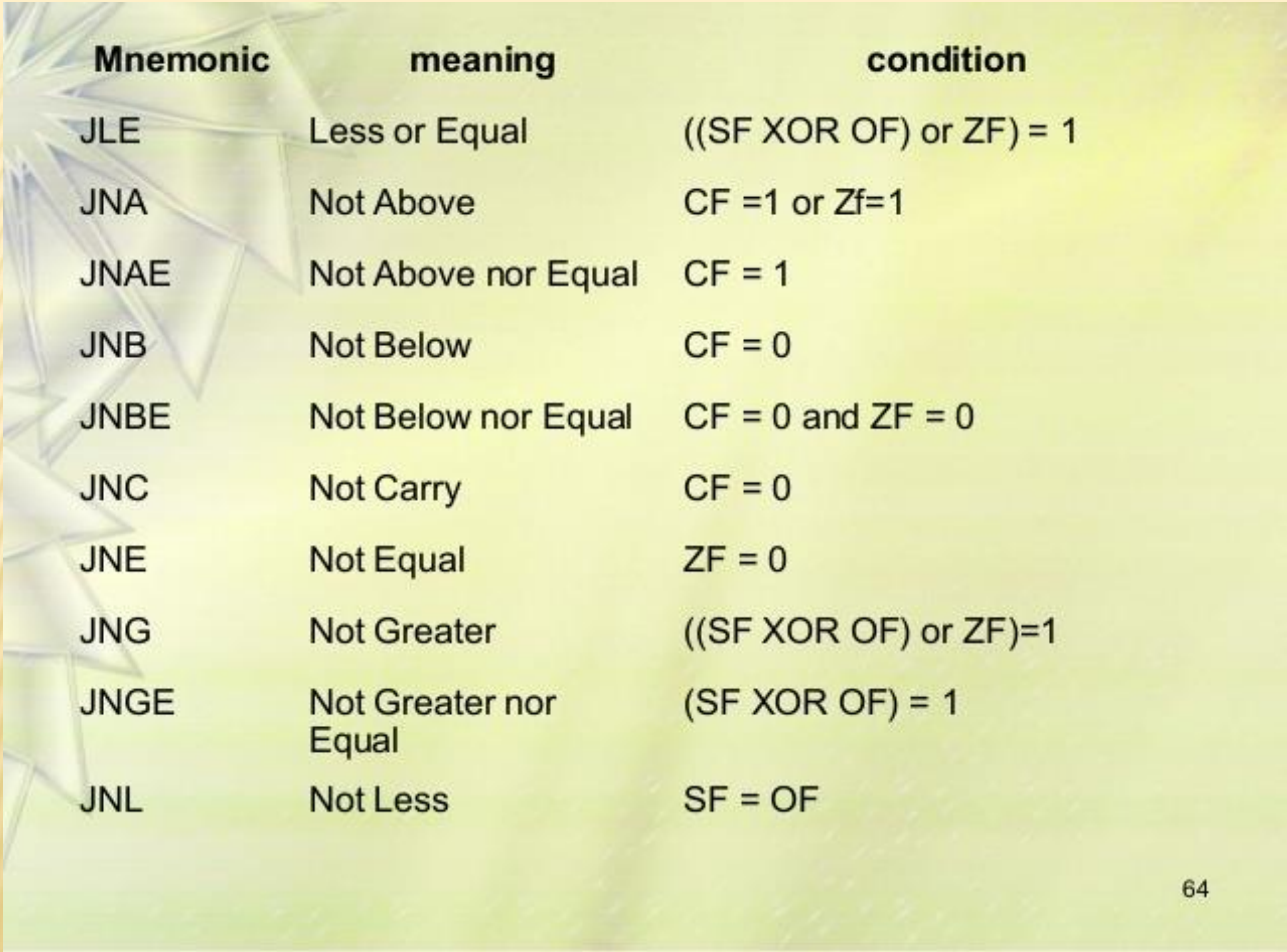
Jumps based on a single flag

Jumps based on more than one flag

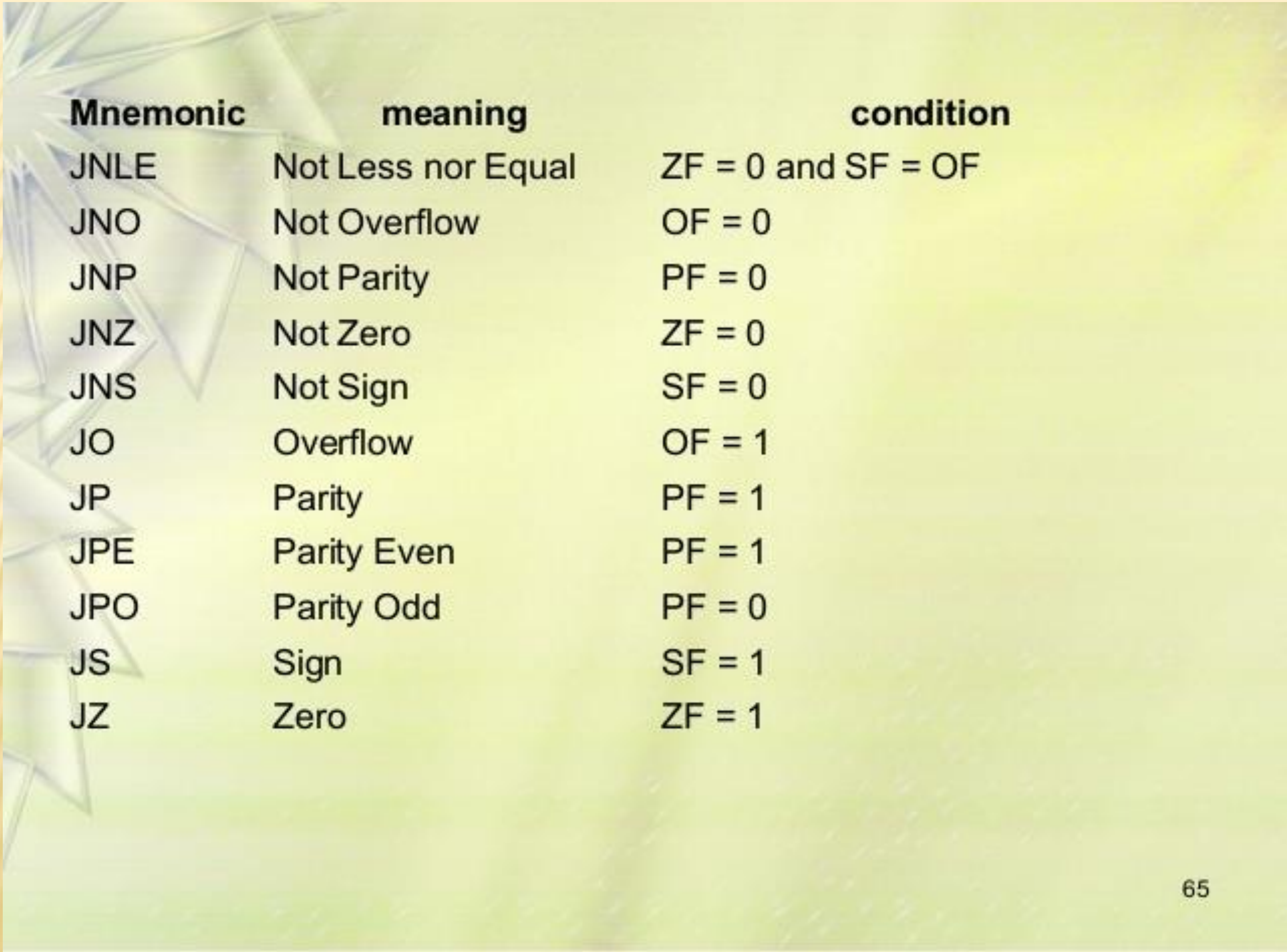


TYPES

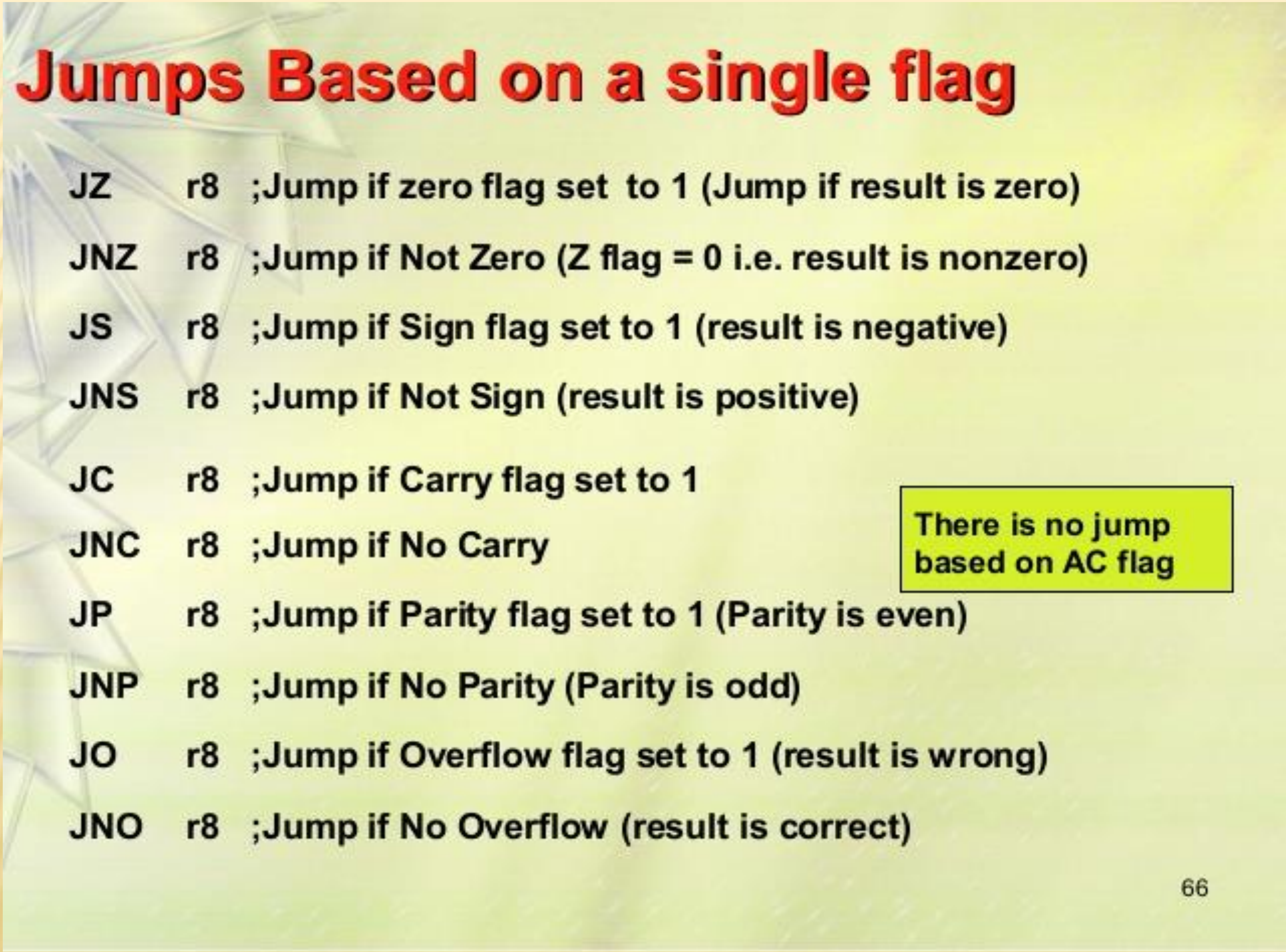
Mnemonic	meaning	condition
JA	Above	CF=0 and ZF=0
JB	Above or Equal	CF=0
JB	Below	CF=1
JBE	Below or Equal	CF=1 or ZF=1
JC	Carry	CF=1
JCXZ	CX register is Zero	(CF or ZF)=0
JE	Equal	ZF=1
JG	Greater	ZF=0 and SF=OF
JGE	Greater or Equal	SF=OF
JL	Less	(SF XOR OF) = 1



Mnemonic	meaning	condition
JLE	Less or Equal	((SF XOR OF) or ZF) = 1
JNA	Not Above	CF =1 or Zf=1
JNAE	Not Above nor Equal	CF = 1
JNB	Not Below	CF = 0
JNBE	Not Below nor Equal	CF = 0 and ZF = 0
JNC	Not Carry	CF = 0
JNE	Not Equal	ZF = 0
JNG	Not Greater	((SF XOR OF) or ZF)=1
JNGE	Not Greater nor Equal	(SF XOR OF) = 1
JNL	Not Less	SF = OF



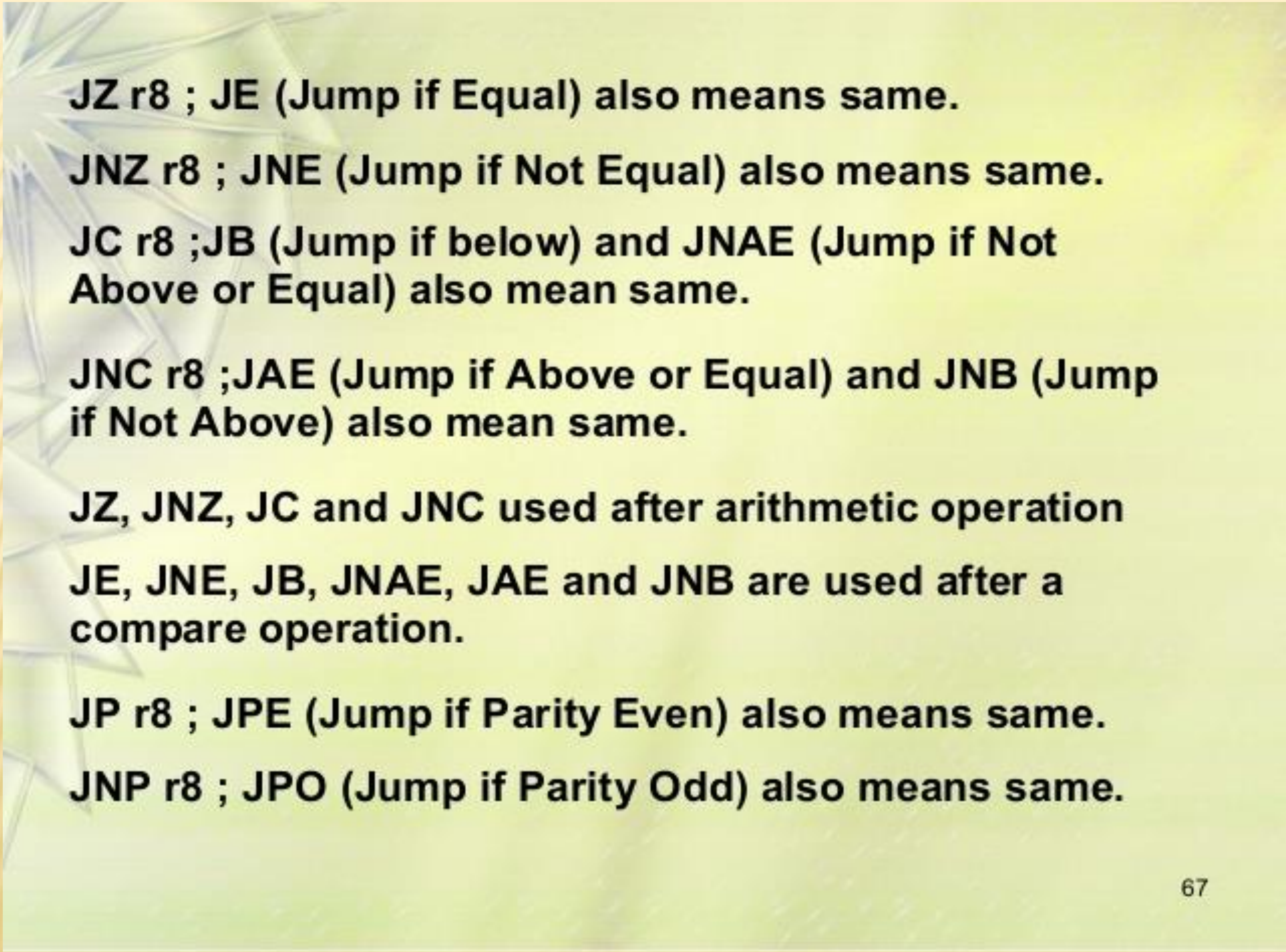
Mnemonic	meaning	condition
JNLE	Not Less nor Equal	ZF = 0 and SF = OF
JNO	Not Overflow	OF = 0
JNP	Not Parity	PF = 0
JNZ	Not Zero	ZF = 0
JNS	Not Sign	SF = 0
JO	Overflow	OF = 1
JP	Parity	PF = 1
JPE	Parity Even	PF = 1
JPO	Parity Odd	PF = 0
JS	Sign	SF = 1
JZ	Zero	ZF = 1



Jumps Based on a single flag

```
JZ    r8  ;Jump if zero flag set  to 1 (Jump if result is zero)
JNZ   r8  ;Jump if Not Zero (Z flag = 0 i.e. result is nonzero)
JS    r8  ;Jump if Sign flag set to 1 (result is negative)
JNS   r8  ;Jump if Not Sign (result is positive)

JC    r8  ;Jump if Carry flag set to 1
JNC   r8  ;Jump if No Carry
JP    r8  ;Jump if Parity flag set to 1 (Parity is even)
JNP   r8  ;Jump if No Parity (Parity is odd)
JO    r8  ;Jump if Overflow flag set to 1 (result is wrong)
JNO   r8  ;Jump if No Overflow (result is correct)
```

There is no jump based on AC flag



JZ r8 ; JE (Jump if Equal) also means same.

JNZ r8 ; JNE (Jump if Not Equal) also means same.

JC r8 ;JB (Jump if below) and JNAE (Jump if Not Above or Equal) also mean same.

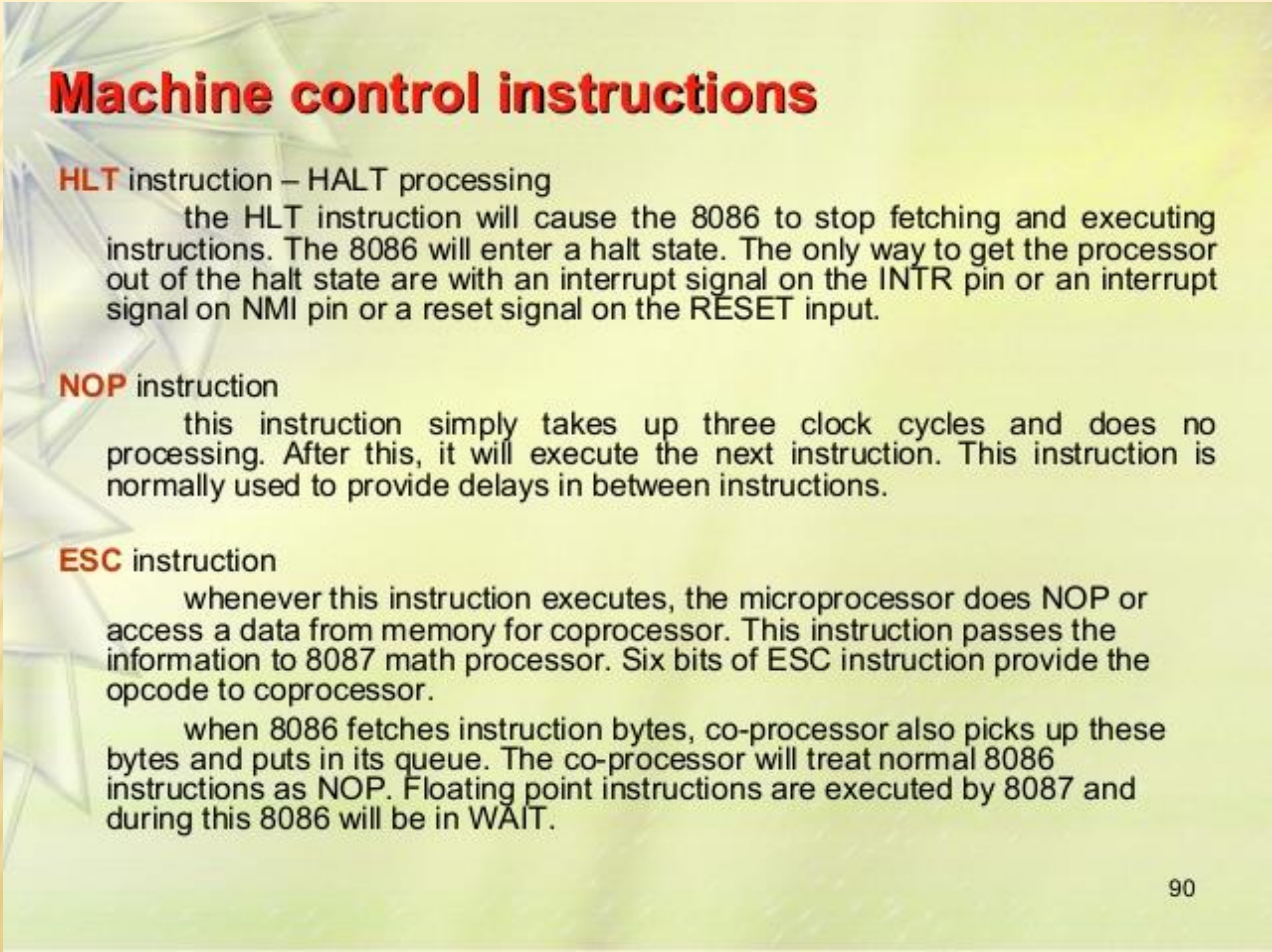
JNC r8 ;JAE (Jump if Above or Equal) and JNB (Jump if Not Above) also mean same.

JZ, JNZ, JC and JNC used after arithmetic operation

JE, JNE, JB, JNAE, JAE and JNB are used after a compare operation.

JP r8 ; JPE (Jump if Parity Even) also means same.

JNP r8 ; JPO (Jump if Parity Odd) also means same.



Machine control instructions

HLT instruction – HALT processing

the HLT instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state. The only way to get the processor out of the halt state are with an interrupt signal on the INTR pin or an interrupt signal on NMI pin or a reset signal on the RESET input.

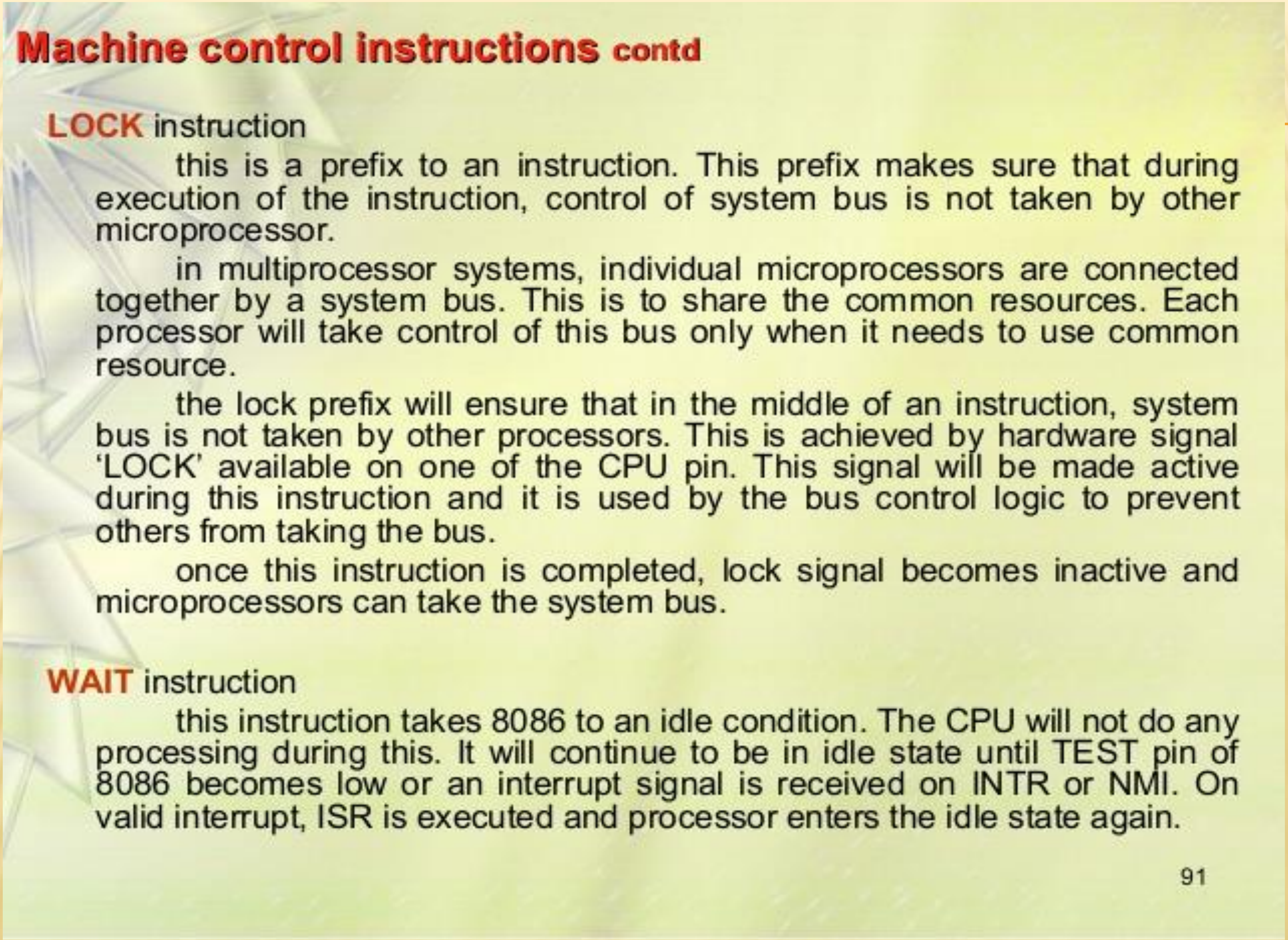
NOP instruction

this instruction simply takes up three clock cycles and does no processing. After this, it will execute the next instruction. This instruction is normally used to provide delays in between instructions.

ESC instruction

whenever this instruction executes, the microprocessor does NOP or access a data from memory for coprocessor. This instruction passes the information to 8087 math processor. Six bits of ESC instruction provide the opcode to coprocessor.

when 8086 fetches instruction bytes, co-processor also picks up these bytes and puts in its queue. The co-processor will treat normal 8086 instructions as NOP. Floating point instructions are executed by 8087 and during this 8086 will be in WAIT.



Machine control instructions contd

LOCK instruction

this is a prefix to an instruction. This prefix makes sure that during execution of the instruction, control of system bus is not taken by other microprocessor.

in multiprocessor systems, individual microprocessors are connected together by a system bus. This is to share the common resources. Each processor will take control of this bus only when it needs to use common resource.

the lock prefix will ensure that in the middle of an instruction, system bus is not taken by other processors. This is achieved by hardware signal 'LOCK' available on one of the CPU pin. This signal will be made active during this instruction and it is used by the bus control logic to prevent others from taking the bus.

once this instruction is completed, lock signal becomes inactive and microprocessors can take the system bus.

WAIT instruction

this instruction takes 8086 to an idle condition. The CPU will not do any processing during this. It will continue to be in idle state until TEST pin of 8086 becomes low or an interrupt signal is received on INTR or NMI. On valid interrupt, ISR is executed and processor enters the idle state again.

Assembler directives

- An assembler is a program that is used to convert an assembly language program into an equivalent machine language program.
- The assembler finds the address of each label and substitutes the value of each constant and variable in the assembly language program during the assembly process, to generate the machine language code.
- While performing these operations, the assembler may find syntax errors.
- They are reported to the programmer at the end of the assembly process.
- The logical and other programming errors are not found by the assembler.
- For completing these tasks the assembler needs some commands from the programmer .The required storage class for a particular constant or a variable such as byte, word, or double word,
- The logical name of the segments such as CODE, STACK, or DATA, the type of procedures or routines such as FAR, NEAR, PUBLIC, or EXTRN, the end of a segment etc.
- These types of commands are given to the assembler using a predefined alphabetical strings called Assembler directives.
- Assembler directives are directions for the assembler, and not the instructions for the 8086.

- **Assembler Directives for variable and Constant Definition**

- (i) **DB, DW, DD, DQ, and DT:**

- DB (define byte),DW(define word),DD(define double word), DQ (define quad word), and DT (define ten bytes) are used to reserve one byte, one word (i.e. 2 bytes), one double word(i.e. 2 words), one quad word(i.e. 4 words) and ten bytes in memory, respectively for storing constants, variables, or strings.

- **Example:DATA1 DB 20h** ; Reserve one byte for storing DATA1and assign the value 20h to it.

- **ARRAY DB 10h, 20h, 30h ;** Reserve three bytes for storing ARRAY and initialize it with the values ; 10h, 20h and 30h

- **CITY DB “NARELA”** ;Store the ASCII code of the characters specified within double quotes in the array or a list named CITY

- **DATA2 DW 1020h ;** Reserve one word for storing DATA2 and Assign the

- The directive **DUP** (duplicate) is used to reserve a series of bytes, words, double words, or ten bytes and is used with DB, DW, DD and DT, respectively.
- The reserved area can be either filled with a specific value or left uninitialized.
- **Example:ARRAY DB 20 DUP(0)** ;Reserve 20 bytes in the memory ; for the array named ARRAY and initialize all the elements of the array to 0 (due to presence of 0 within the bracket near the DUP
- **ARRAY1 DB 25 DUP (?)** ; Reserve 25 bytes in the memory for the array named ARRAY1 and keep all the elements of the array uninitialized (due to the question mark present within the bracket near the DUP
- **ARRAY2 DB 50 DUP (64h)** ; Reserves 50 bytes in the memory for the array named ARRAY2 and initializes all the elements of the array to 64h.
- EQU: The directive EQU(equivalent) is used to assign a value to a data name **Example: NUMBER EQU 50h** ; Assign the value 50h to NUMBER.
- NAME EQU “RAMESH” ; Assign the string “RAMESH” to NAME
- **Assembler Directives Related to Code(Program) Location:** (i) ORG:
- The ORG (origin) directive directs the assembler to start the memory allocation for a particular segment (data, code, or stack) form the declared offset address in the ORG statement. While starting the assembly process for a memory segment, the assembler initializes a location counter (LC) to

- When the ORG directive is not mentioned , LC is initialized with the offset address 0000h.
- When the ORG directive is mentioned at the beginning of the statement, LC is initialized with the offset address specified in the ORG directive.
- **Example:** **ORG 100h** When this directive is placed at the beginning of the code segment, the location counter is initialized with 0100h and the first instruction is stored from the offset address 0100h within the code segment.
- If it is placed in the data segment, the next data storage starts from the offset address 0100h within the data segment.
- (ii) **EVEN:** The EVEN directive updates the location counter to next even address, if the current location counter content is not an even number.
- Example: **ARRAY2 DW 20 DUP (0)** These statements in a segment declare an array named ARRAY2 having 20 words, starting at an even address.
- The advantage of storing an array of words starting at an even address is that the 8086 takes just one memory read/write cycle to read/write the entire word
- Otherwise the 8086 takes two memory read/write cycles to read/write to the word.

Example:

```
EVEN
RESULT      PROC NEAR
                 :
                 :      Instruction in Result Procedure
                 :
RESULT      ENDP
```

Here the procedure RESULT, which is of type NEAR, is stored starting at an even address in the code segment. The ENDP directive indicates the end of the RESULT procedure.

(iii) **LENGTH:** This directive is used to determine the length of an array or string in bytes.

Example: **MOV CX, LENGTH ARRAY CX** is loaded with the number of bytes in the ARRAY.

(iv) **OFFSET:** This operator is used to determine the offset of a data item in a segment containing it.

Example: **MOV BX, OFFSET TABLE**

If the data item named TABLE is present in the data segment, this statement places the offset address of TABLE, in the BX register.

Assembler Directives for Segment Declaration

(i) **SEGMENT and ENDS**: The SEGMENT and ENDS directives indicate the start and end of a segment, respectively. In some cases, the segment may be assigned a type such as **PUBLIC** (i.e. it can be used by other modules of the program while linking) or **GLOBAL** (i.e. it can be accessed by any other module).

Example: This example indicates the declaration of a code segment named CODE 1.

```
          CODE 1 SEGMENT
          :            Instructions of CODE 1 segment
   CODE 1 ENDS
```

(ii) **ASSUME:** The ASSUME directive is used to inform the assembler, the name of the logical segments to be assumed for different segments used in the program.

This statement informs the assembler that the segment address where the logical segments CODE1 and DATA1 are loaded in memory during execution is to be stored in CS and DS registers, respectively.

```
                     ASSUME CS : CODE 1, DS: DATA1
```

(iii) **GROUP:** This directive is used to form a logical group of segments with a similar purpose. The Assembler passes information to the linker/loader to form the code, such that the group declared segments or operands lie within a 64 Kb memory segment. All such segments can be addressed using the same segment address.

Example: This statement directs the loader/linker to prepare an executable file (.exe) such that the CODE1, DATA1, and STACK1 segments lie within a 64KB memory segment that is named

PROGRAM1 GROUP CODE1, DATA1, STACK1

The Assembler directives for declaring procedures:

(i) **PROC :** The PROC directive indicates the start of a named procedure. The NEAR and FAR directive specify the type of the procedure:

Example: This statement indicates the beginning of a procedure named SQUARE_ROOT, which is to be called by a program located in the same segment. The **FAR** directive is used for procedures to be called by the programs present in code segments other than the one in which this procedure is present.

For example, SALARY PROC FAR indicates the beginning of a FAR type procedure named SALARY

SQUARE_ROOT PROC NEAR

(ii) **ENDP:** The ENDP directive is used to indicate the end of a procedure. To mark the end of a particular procedure, the name of the procedure may appear as prefix with the directive ENDP.

Example: SALARY PROC NEAR

…… ; Code of SALARY Procedure

(iii) **EXTRN and PUBLIC:** The directive EXTRN (external) informs the assembler that the procedures, label/labels, and names declared after this directive has/have already been defined in some other segments and in the segments where they actually appear, they must be declared in public, using the **PUBLIC** directive.

```
Example: MODULE1 SEGMENT
                 PUBLIC SQURE_ROOT
                 SQUARE_ROOT PROC FAR
                             ….                    ;CODE OF SQUARE_ROOT
PROCEDURE
                 SQUARE_ROOT ENDP
                   MODULE1 ENDS ;
```

iii) **EXTRN and PUBLIC (continued): NOTE:** If one wants to call the procedure named **SQUARE_ROOT** appearing in **MODULE1** from **MODULE2,** it must be declared using the statement **PUBLIC SQUARE_ROOT** in **MODULE1** and it must be declared external using the statement **EXTRN SQUARE_ROOT** in **MODULE2.** If a jump or a call address is external, it must be represented as NEAR or FAR. If data are defined as external, their size must be represented as BYTE, WORD, or DWORD.

```
MODULE2
SEGMENT
              EXTRN SQUARE_ROOT FAR
                            ……                         ; CODE OF MODULE2
```

MACRO and ENDM

- Suppose a number of instructions occur repeatedly in the main program, the program listing becomes lengthy.
- In such a situation, a macro definition, i.e. a label, is assigned with the repeatedly appearing string of instructions.
- The process of assigning a label or macro name to the repeatedly appearing string of instructions is called macro definition.
- The macro name is then used throughout the main program to refer to that string of instructions.
- Defining a **MACRO**
- **CALCULATE MACRO**
- **MOV AX, [BX]**
- **ADD AX, [BX + 2]**
- **MOV [SI], AX**
- **ENDM**

CALCULATE is the macro name and the macro is used to add two successive data in the memory, whose offset address is present in BX and the result is stored in the memory at the offset address in SI.

Using parameters in macro definition, the programmer specifies the parameters of the macro that are likely to be changed each time the macro is called. The macro given before (CALCULATE) can be modified to calculate the result for the different sets of data and store it in a different memory locations as follows:

```
CALCULATE MACRO OPERAND, RESULT
                  MOV BX, OFFSET OPERAND
                  MOV AX, [BX]
                 ADD AX, [BX + 2]
                 MOV SI, OFFSET RESULT
                 MOV [SI], AX
                 ENDM
```

Example: Program to find the average of 10 byte-type data stored in an array in data segment.

```
        ASSUME CS: CODE1, DS: DATA1
        DATA1 SEGMENT                         ;data segment ; starts
       ARRAY DB 12h, 23h, 44h, 56h,
                0ABh, 73h, 44h, 0ABh,
                0EEh, 0Ah                      ; 10 bytes are stored
```

```
          COUNT EQU 10                  ;Count is the number of bytes in the array
          AVERAGE DB 01 DUP(0)   ;Reserve one byte ; to store the result
          DATA1 ENDS                    ;data segment ; ends

          CODE1 SEGMENT                  ; Code segment ; starts
  START:MOV AX, DATA1                  ; Segment address of DATA1 is moved to AX
          MOV DS, AX                    ;MOV AX contents to DS
          MOV SI, OFFSET ARRAY     ;Move offset address of ARRAY to SI
          XOR AX, AX                     ;Clear AX and Carry Flag
          MOV BX, 0000h                 ;Clear BX
          MOV CX, COUNT                  ;Move COUNT to CX
  NEXT: MOV BL, [SI]                   ;Move one byte ; from array into BL
         ADD AX, BX                     ;Add AX and BX
         INC SI                          ;Increment SI to point to next byte
         LOOP NEXT                      ;Repeat Loop NEXT CX times
         MOV DH, COUNT                 ;MOV Count to DH
         DIV DH                         ;Divide AX by CH
         MOV AVERAGE, AL              ;Store AL contents in AVERAGE
```

UNIT3
Interfacing with 8086

Memory interface

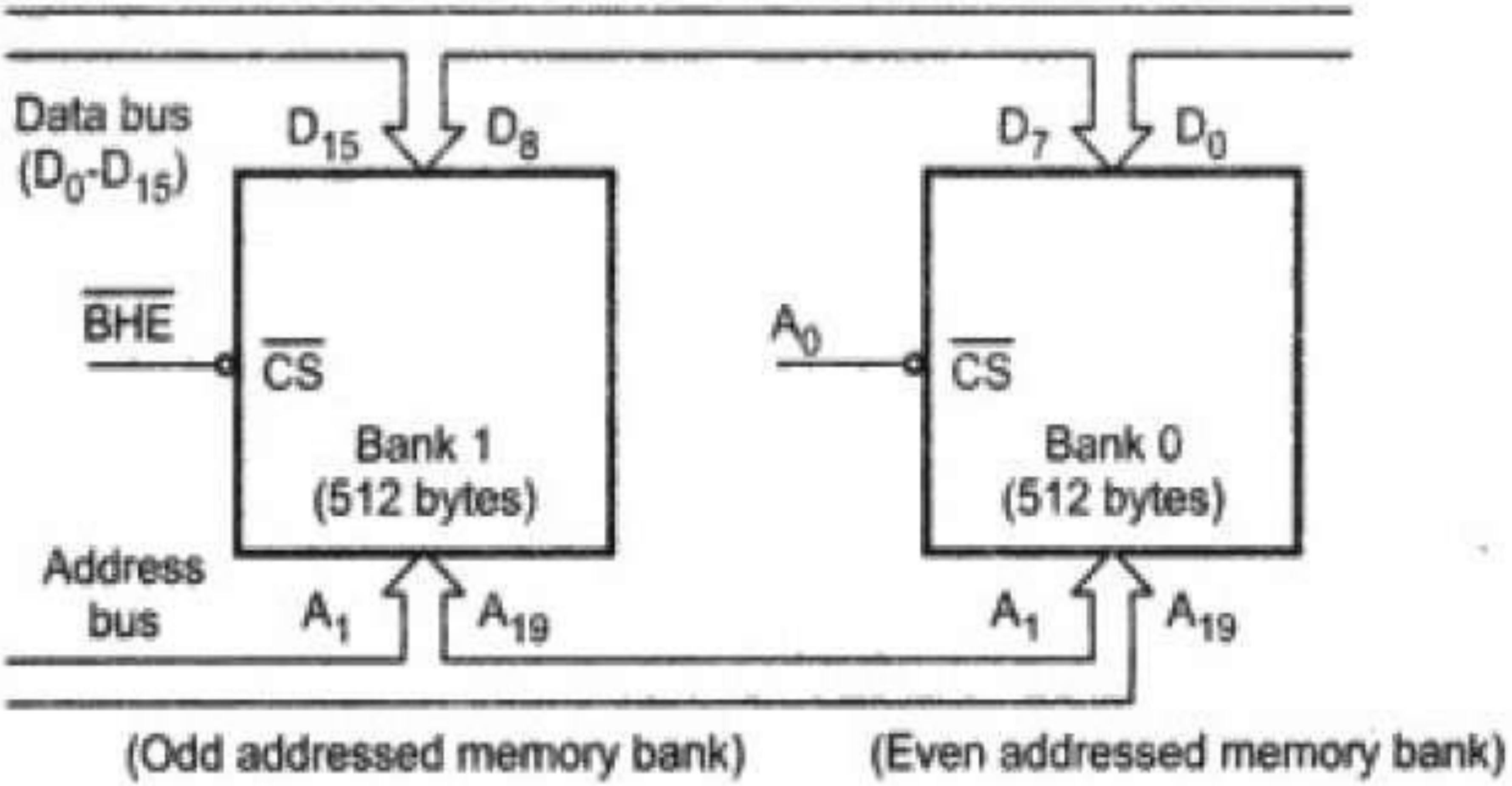
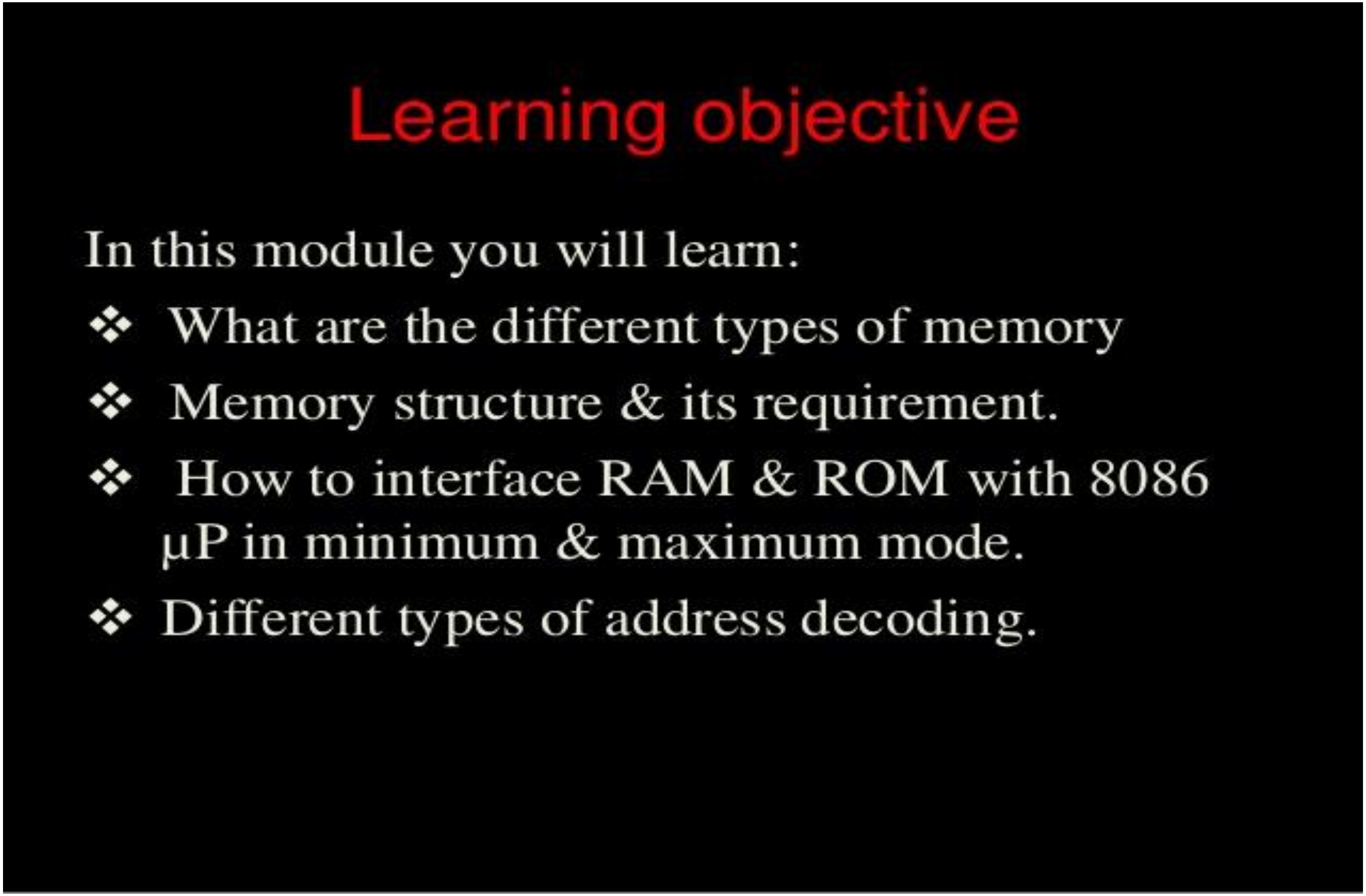


Fig. 5.2 Memory interfacing

Memory interface



Learning objective

In this module you will learn:

- What are the different types of memory
- Memory structure & its requirement.
- How to interface RAM & ROM with 8086 µP in minimum & maximum mode.
- Different types of address decoding.

Memory fundamentals

- Memory capacity
  - The no. of bits that a semiconductor memory chip can store is called its chip capacity.
- Memory Organization:
  - Each memory chip contains 2^N locations, where N is the no. of address pins on the chip.
  - Each location contains M bits, where M is the no. of data pins on the chip.
  - The entire chip will contain 2^N x M bits.
  - E.g. for 4K x 4, 2^{12}=4096 locations, each location holding 4 bits, so N=12 & M=4.

Memory types

1) ROM (Read Only Memory)

2) PROM (programmable memory)

3) EPROM (Erasable programmable ROM)

4) EEPROM (Electrically Erasable PROM) 500000 times

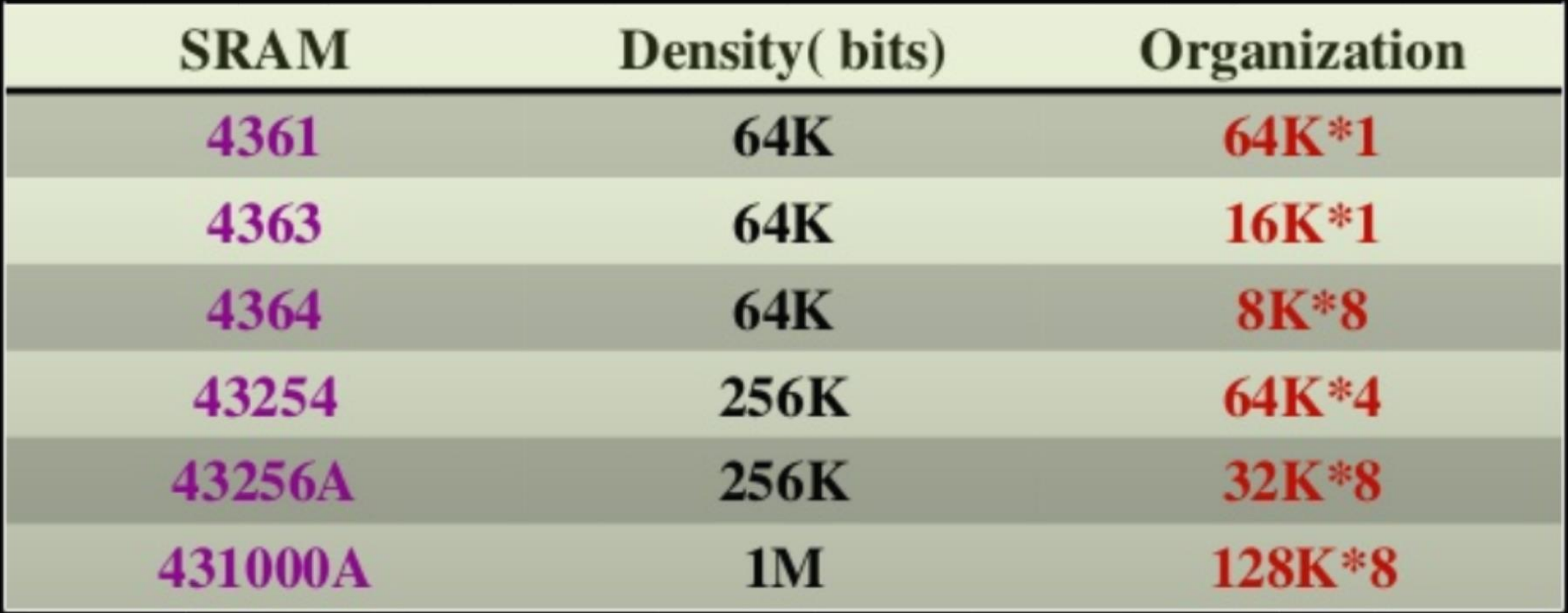
5) Flash memory EPROM

6) RAM (Random Access Memory)

Standard EPROM ic

EPROM	Density(bits)	Capacity (bytes)
2716	16K	2K*8
2732	32K	4K*8
27C64	64K	8K*8
27C128	128K	16K*8
27C256	256K	32K*8
27C512	512K	64K*8
27C010	1M	128K*8
27C020	2M	256K*8
27C040	4M	512K*8

Standard SRAM ic

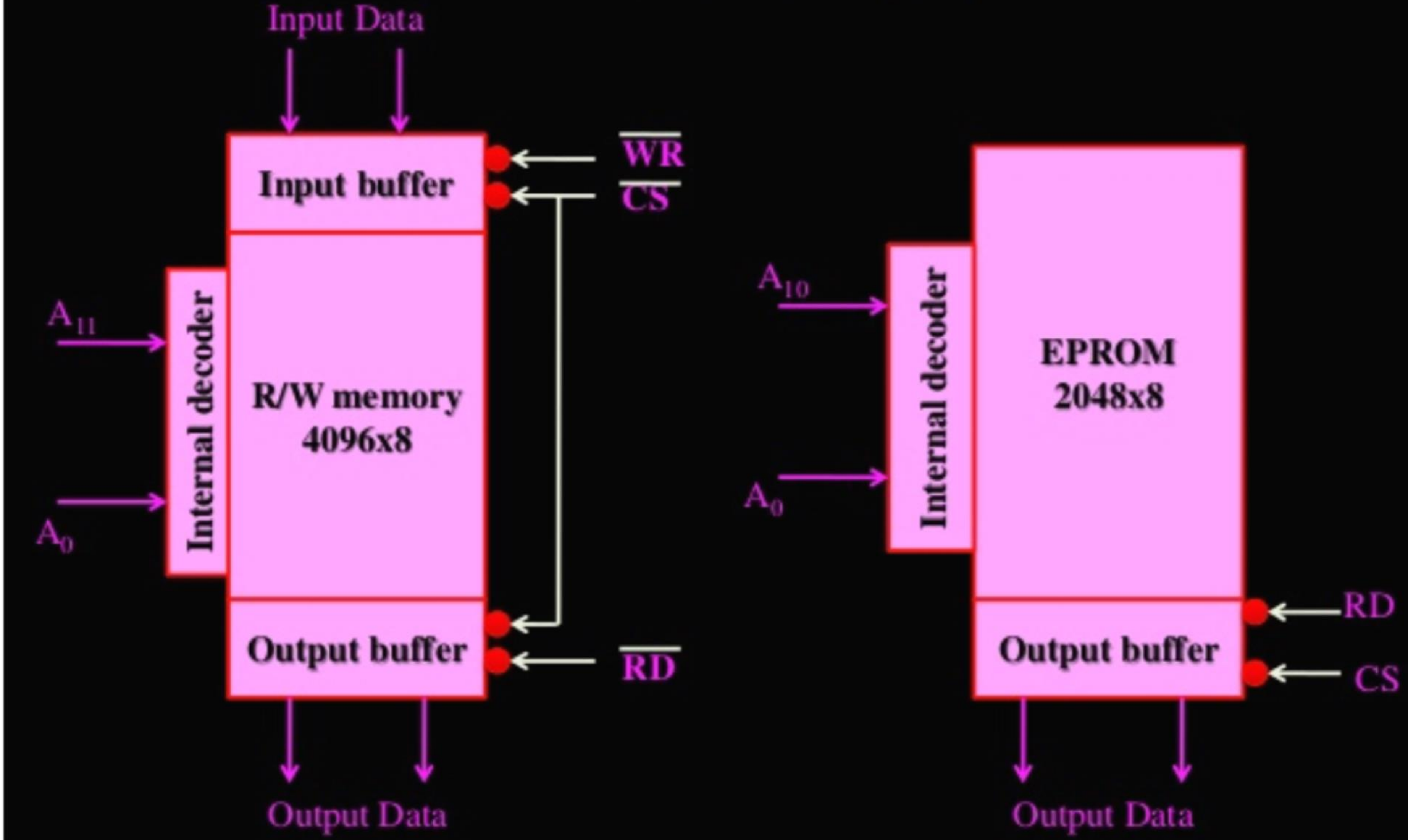


SRAM	Density(bits)	Organization
4361	64K	64K*1
4363	64K	16K*1
4364	64K	8K*8
43254	256K	64K*4
43256A	256K	32K*8
431000A	1M	128K*8

Standard DRAM ic

EPROM	Density(bits)	Capacity (bytes)
2164	64K	64 Kx1
21256	256K	256 Kx1
21464	256K	64 Kx4
421000	1M	1 Mx1
424256	1M	256 Kx4
44100	4M	4 Mx1
44400	4M	1 Mx4
44160	4M	256 Kx16
416800	16M	8 Mx2
416400	16M	4 Mx4
416160	16M	1 Mx16

MEMORY STRUCTURE & ITS REQUIREMENT



PHYSICAL STRUCTURE OF PRACTICAL MEMORY IC

1. Address Pins:

No of address pins	*No of memory location*
8	**2^8 = 256 location**
9	**2^9 = 512 location**
10	**2^{10} = 1024 = 1K location**
11	**2^{11} = 2048 = 2K location**
12	**2^{12} = 4 K**
13	**2^{13} = 8 K**
14	**2^{14} = 16 K**
15	**2^{15} = 32 K**
16	**2^{16} = 64 K**
17	**2^{17} = 128 K**
18	**2^{18} = 256 K**
19	**2^{19} = 512K**
20	**2^{20} = 1024K = 1M**

PHYSICAL STRUCTURE OF PRACTICAL MEMORY IC

2. Data pins: Number of flip flop in each location is 4/8, then data pins 4/8.

3. Control pins:

ROM/ EPROM will consist of only $\overline{RD}$ ($\overline{OE}$)

RAM will have control pins $\overline{RD}$ & $\overline{WR}$.

4. Commons pins: $\overline{CS}$ (chip select) .

$\overline{CS}$ is generated using:

i. NAND gate

ii. 3 to 8 decoder

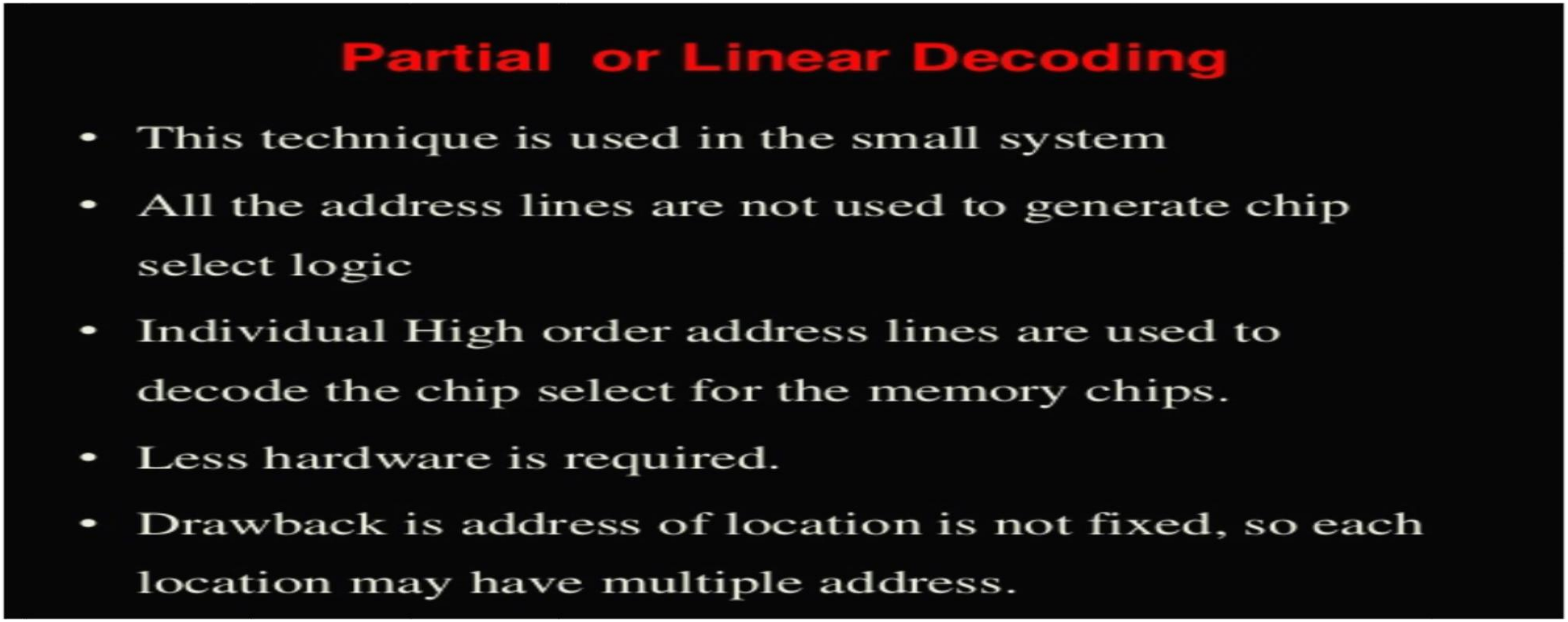
iii. PAL IC

Address decoding

- In general all the address lines are not used by the memory devices to select particular memory locations.
- The remaining line are used to generate chip select logic.
- Following two techniques are used to decode the address:

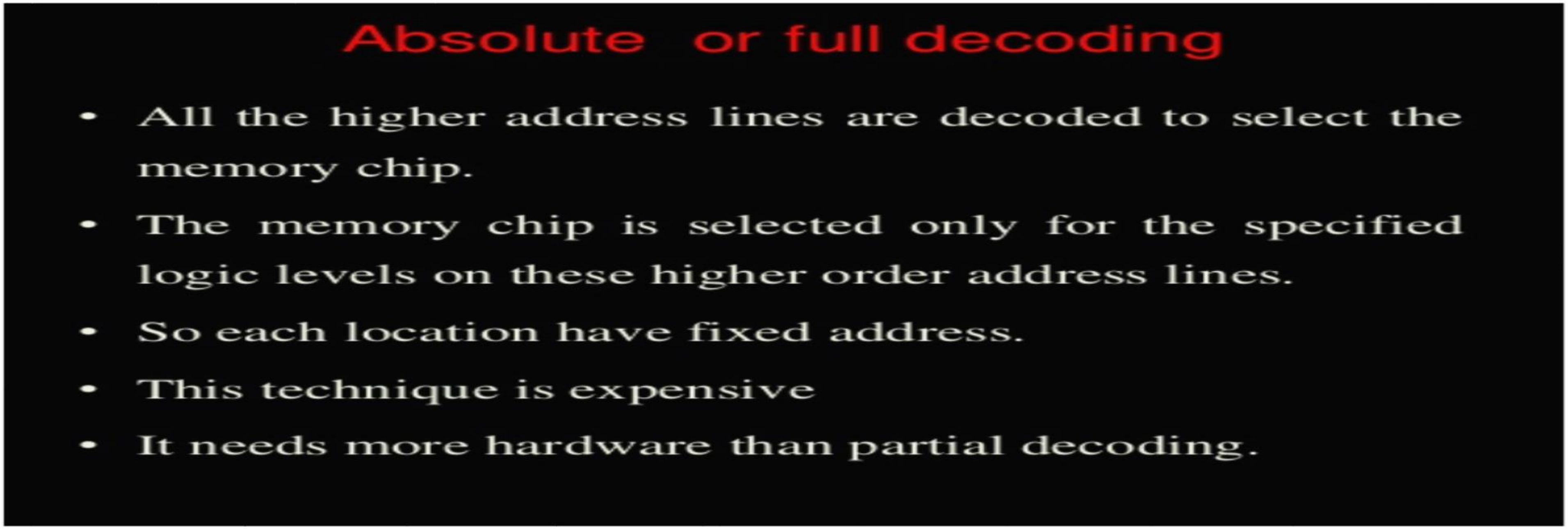
1) Absolute or Full decoding

2) Linear or Partial decoding



Partial or Linear Decoding

- This technique is used in the small system
- All the address lines are not used to generate chip select logic
- Individual High order address lines are used to decode the chip select for the memory chips.
- Less hardware is required.
- Drawback is address of location is not fixed, so each location may have multiple address.



Absolute or full decoding

- All the higher address lines are decoded to select the memory chip.
- The memory chip is selected only for the specified logic levels on these higher order address lines.
- So each location have fixed address.
- This technique is expensive
- It needs more hardware than partial decoding.

Q. 1: Interface 32 KB of RAM memory to the 8086 microprocessor system using absolute decoding with the suitable address.

Step_1: Total RAM memory = 32 KB

Half RAM capacity = 16 KB

hence,

number of RAM IC required = 2 ICs of 16 KB

so,

EVEV Bank = 1 ICs of 16 KB RAM

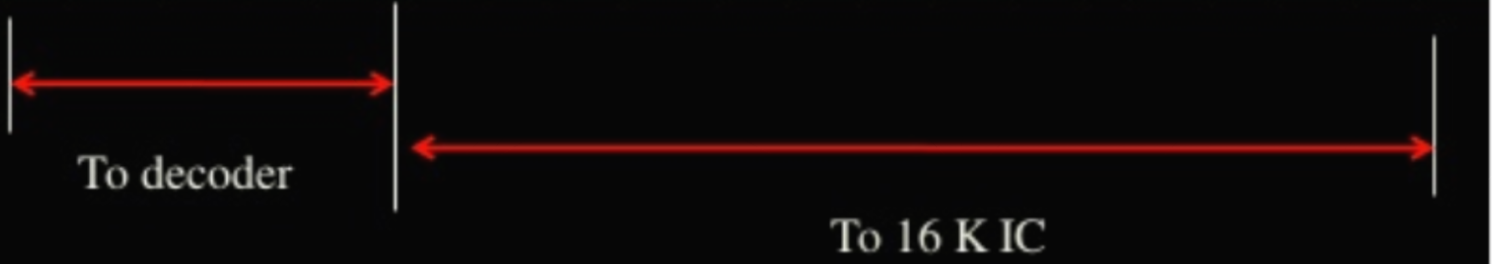
ODD Bank = 1 ICs of 16 KB RAM

Even bank	Odd bank
RAM _1 (16KB)	RAM _2 (16KB)

Step_2: Number of address lines required = 15 address lines

Step_3: Address decoding table

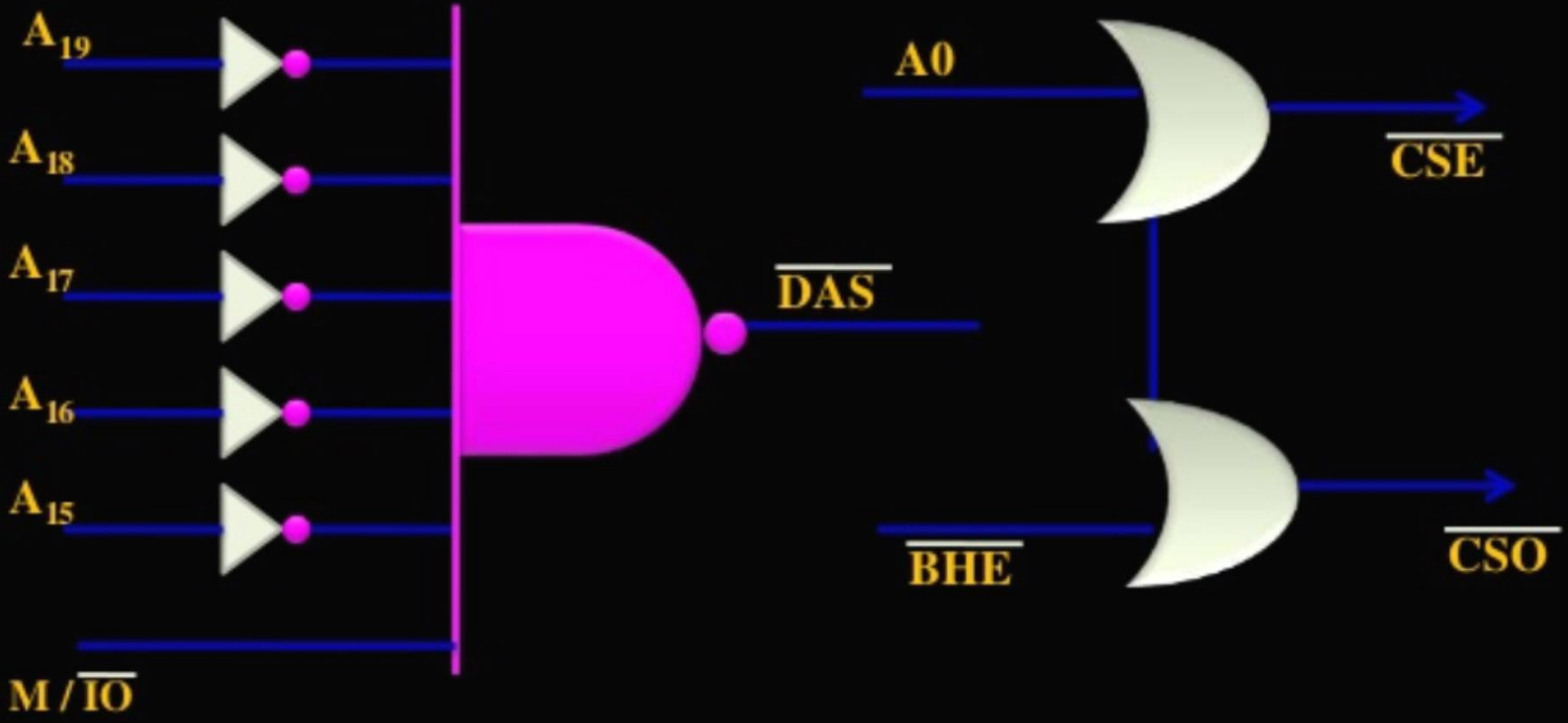
MEMORY IC	HEX ADDRESS	BINARY ADDRESS																			
		A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
16 K x 8 RAM-(1)	OOOOO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	07FFE	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

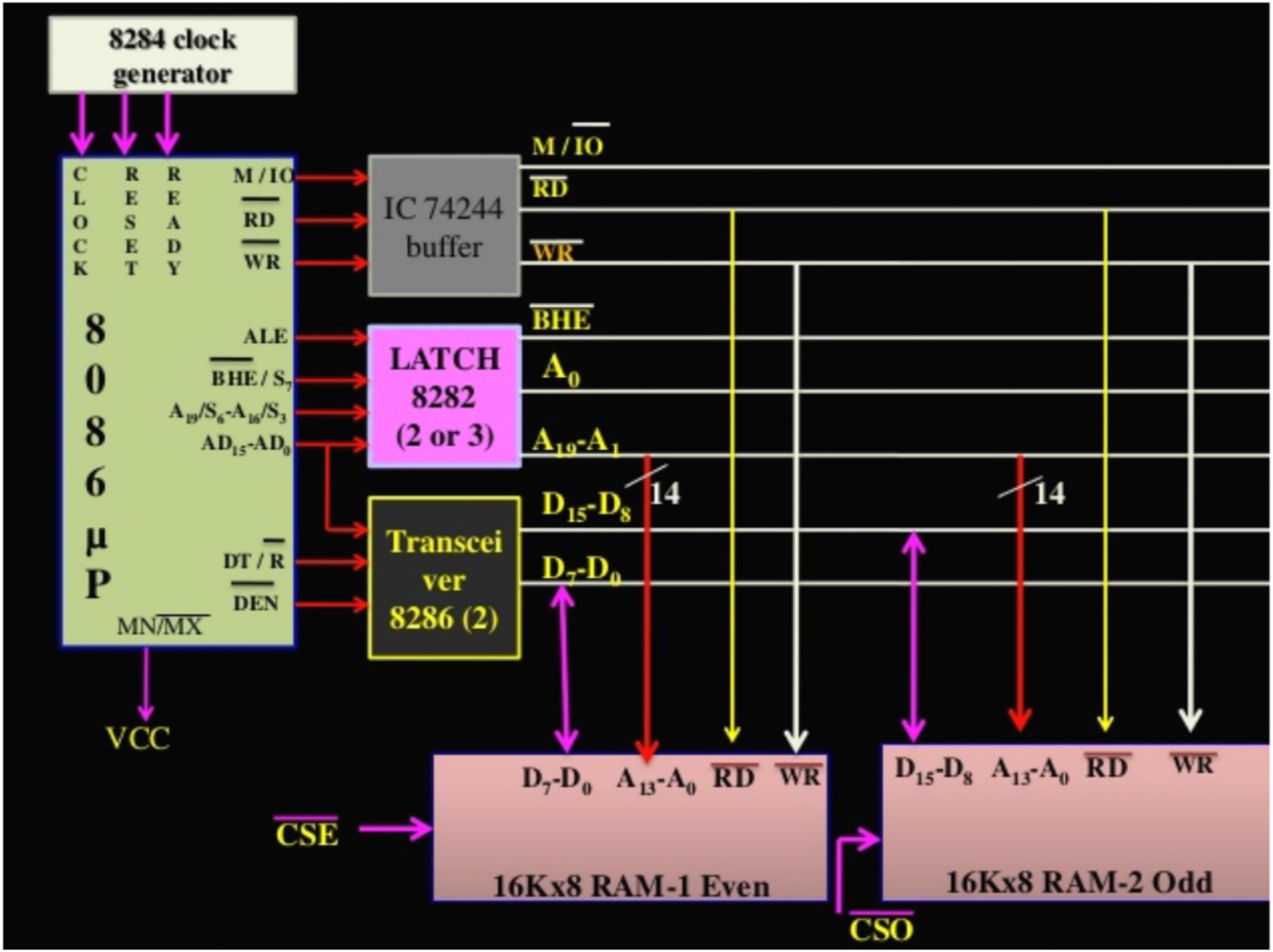


To decoder

To 16 K IC

Step_3: Generation of chip select logic




8284 clock generator
CLOCK
RESET
READY
M / IO
$\overline{RD}$
$\overline{WR}$
8086 μP
ALE
$\overline{BHE}$ / S_7
$A_{19}/S_6-A_{16}/S_3$
$AD_{15}-AD_0$
DT / $\overline{R}$
$\overline{DEN}$
MN/$\overline{MX}$
VCC
IC 74244 buffer
LATCH 8282 (2 or 3)
Transceiver 8286 (2)
M / $\overline{IO}$
$\overline{RD}$
$\overline{WR}$
$\overline{BHE}$
A_0
$A_{19}-A_1$
14
$D_{15}-D_8$
D_7-D_0
14
D_7-D_0
$A_{13}-A_0$
$\overline{RD}$
$\overline{WR}$
$\overline{CSE}$
16Kx8 RAM-1 Even
$D_{15}-D_8$
$A_{13}-A_0$
$\overline{RD}$
$\overline{WR}$
16Kx8 RAM-2 Odd
$\overline{CSO}$

Q. 2: Interface 32 K word of memory to the 8086 microprocessor system . Available memory chips are 16 K x 8 RAM. Use suitable decoder for generating chip select logic.

Step_1: Total memory = 32 K word = 32*2 K = 64 K

IC available = 16 K

hence,

number of RAM IC required = 64 K x 8/ 16 Kx8 = 4 ICs

so,

EVEV Bank = 2 ICs of 16 Kx8 RAM

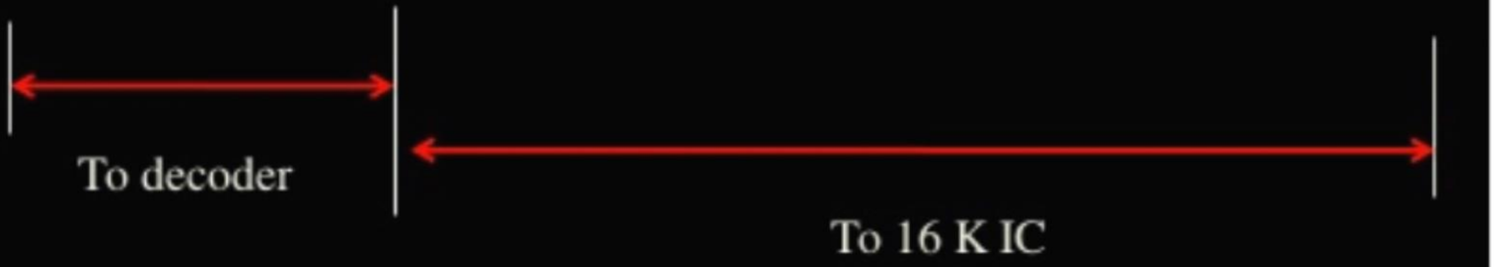
ODD Bank = 2 ICs of 16 Kx8 RAM

Even bank	Odd bank
RAM _1 (16K)	RAM _2 (16K)
RAM_ 3 (16K)	RAM _4 (16K)

Step_2: Number of address lines required = 15 address lines

Step_3: Address decoding table

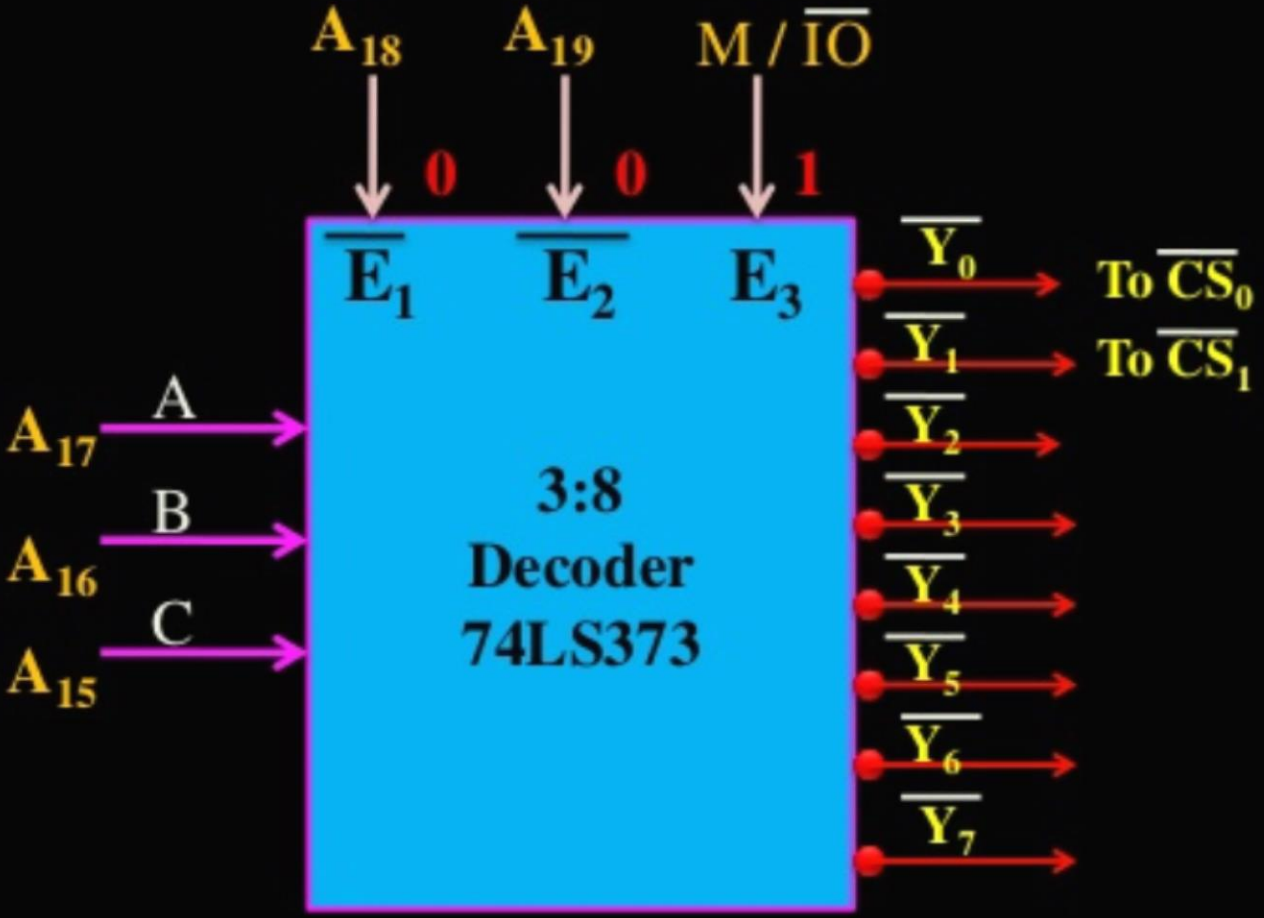
MEMORY IC	HEX ADDRESS	BINARY ADDRESS																			
		A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
16 K x 8 RAM-(1)	OOOOO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	07FFE	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
16 K x 8 RAM-(3)	8000	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0FFFE	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

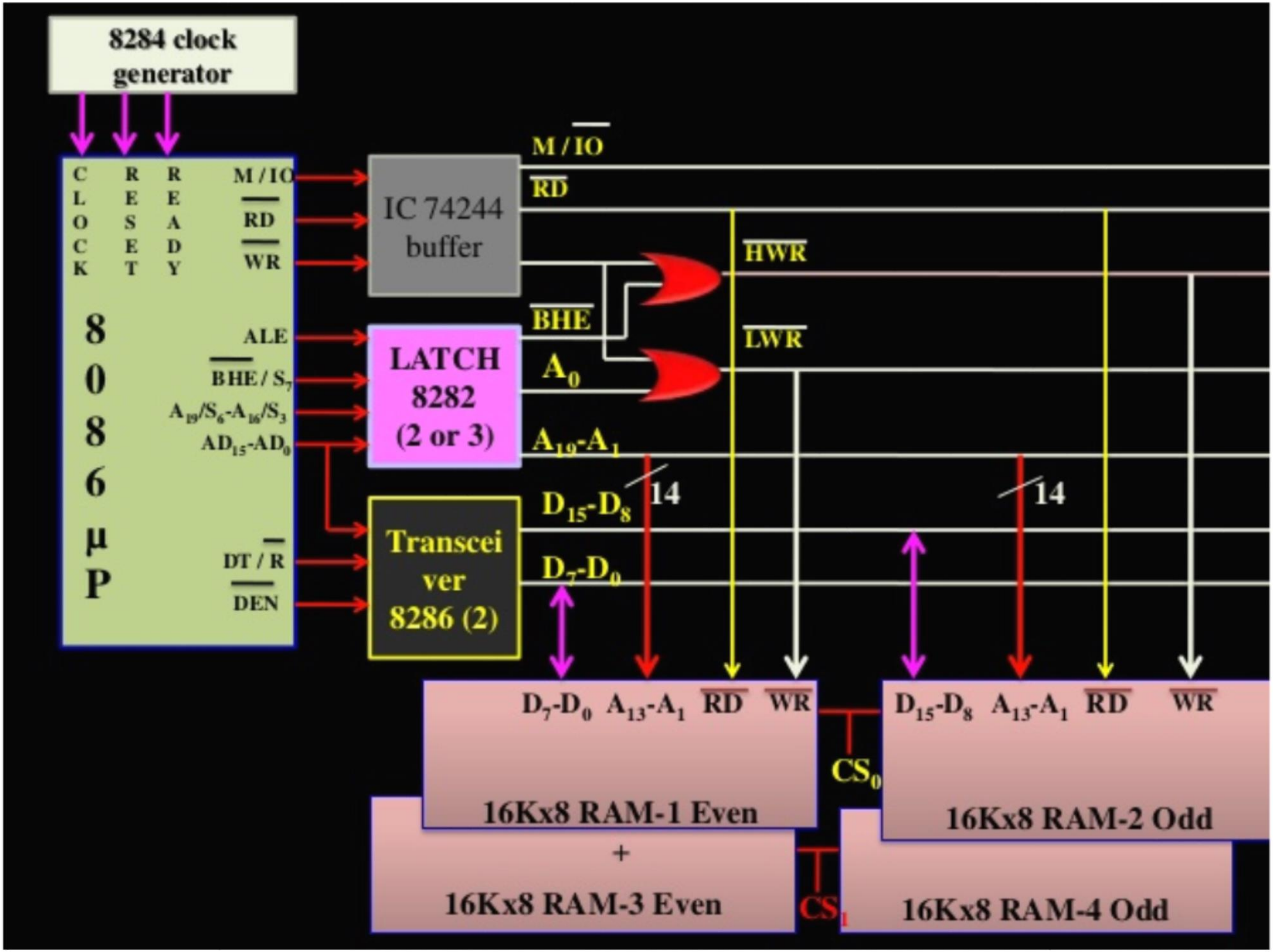


To decoder

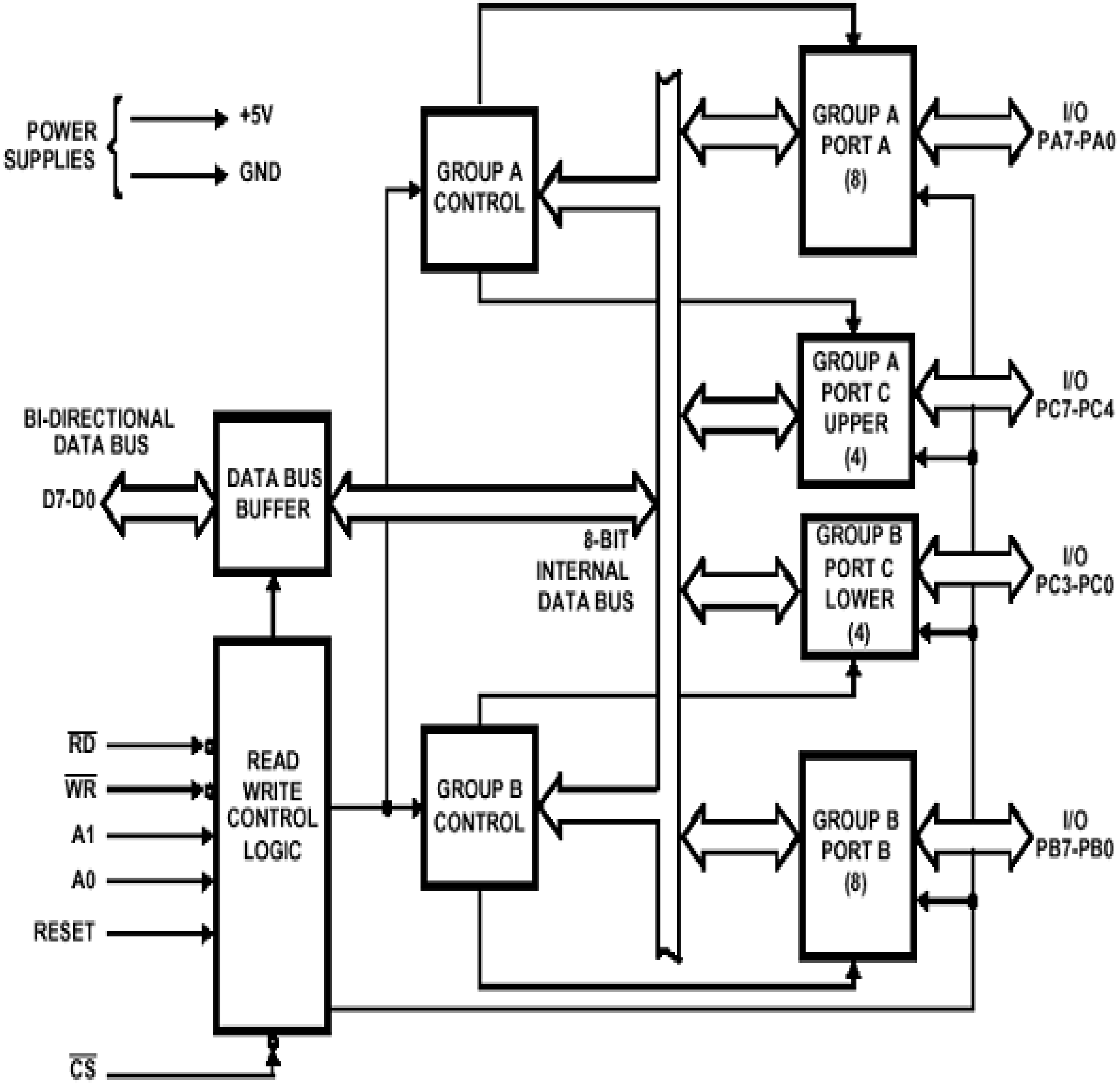
To 16 K IC

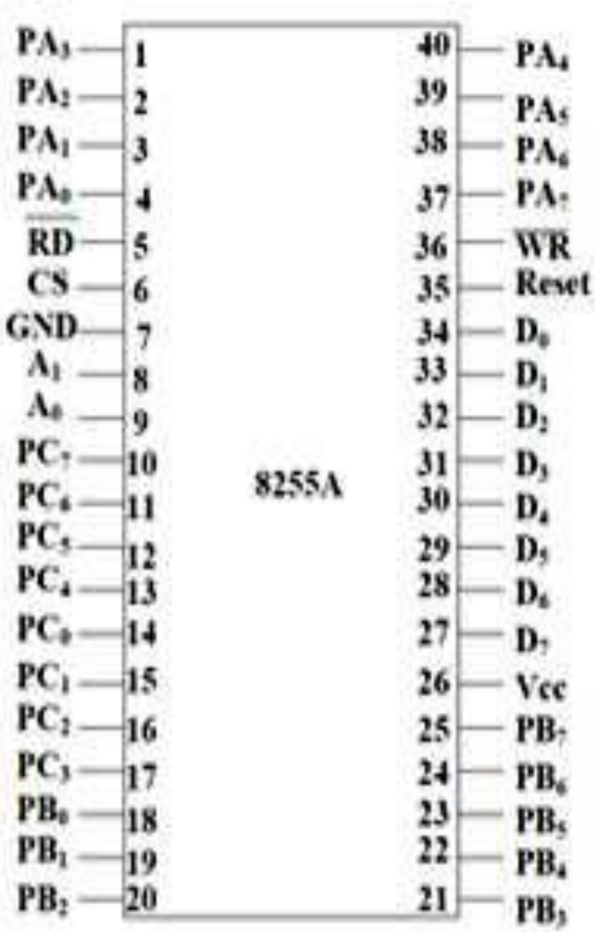
Step_3: Generation of chip select logic




8284 clock generator
CLOCK
RESET
READY
M / IO
$\overline{RD}$
$\overline{WR}$
ALE
$\overline{BHE}$ / S_7
A_{19}/S_6-A_{16}/S_3
AD_{15}-AD_0
DT / $\overline{R}$
$\overline{DEN}$
8086 μP
IC 74244 buffer
LATCH 8282 (2 or 3)
Transceiver 8286 (2)
M / $\overline{IO}$
$\overline{RD}$
$\overline{HWR}$
$\overline{BHE}$
$\overline{LWR}$
A_0
A_{19}-A_1
14
14
D_{15}-D_8
D_7-D_0
D_7-D_0
A_{13}-A_1
$\overline{RD}$
$\overline{WR}$
D_{15}-D_8
A_{13}-A_1
$\overline{RD}$
$\overline{WR}$
CS_0
16Kx8 RAM-1 Even
16Kx8 RAM-2 Odd
+
16Kx8 RAM-3 Even
CS_1
16Kx8 RAM-4 Odd

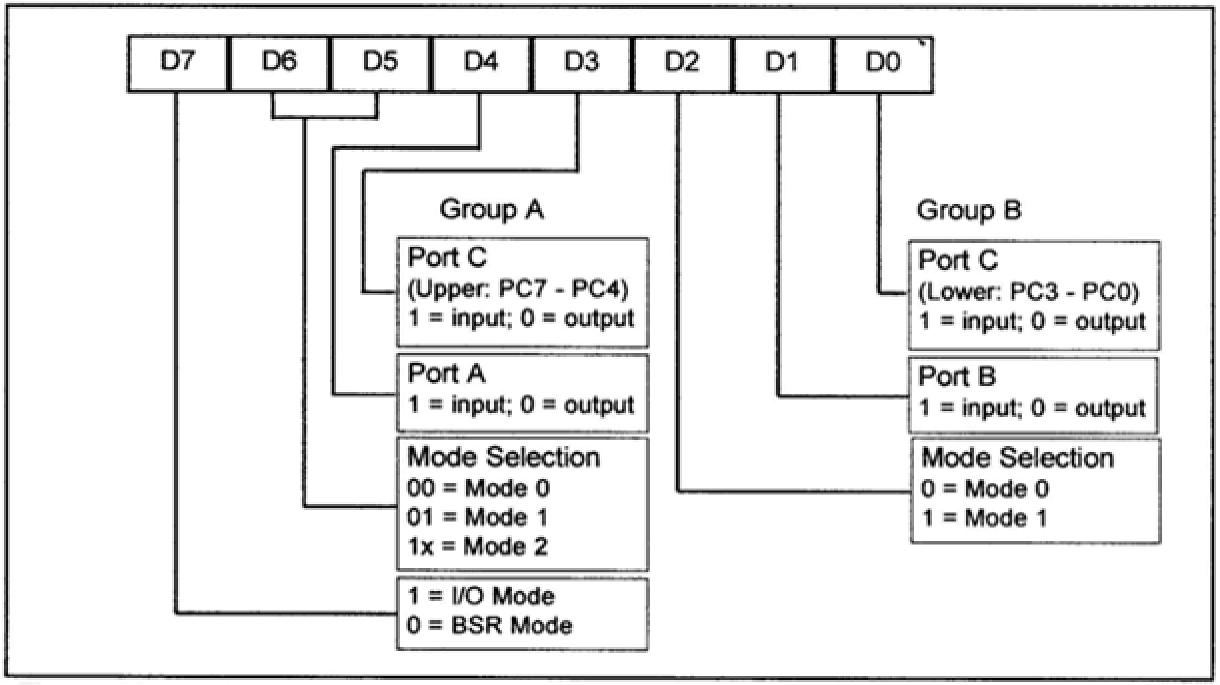
8255A - Programmable Peripheral Interface



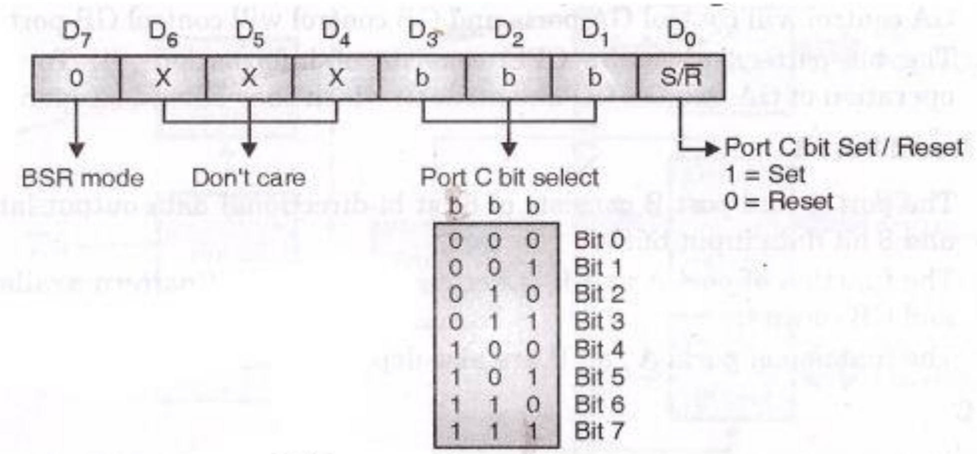


Read/Write Control Logic

CS	A1	A0	Result
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	CWR



Control Word Format 8255A

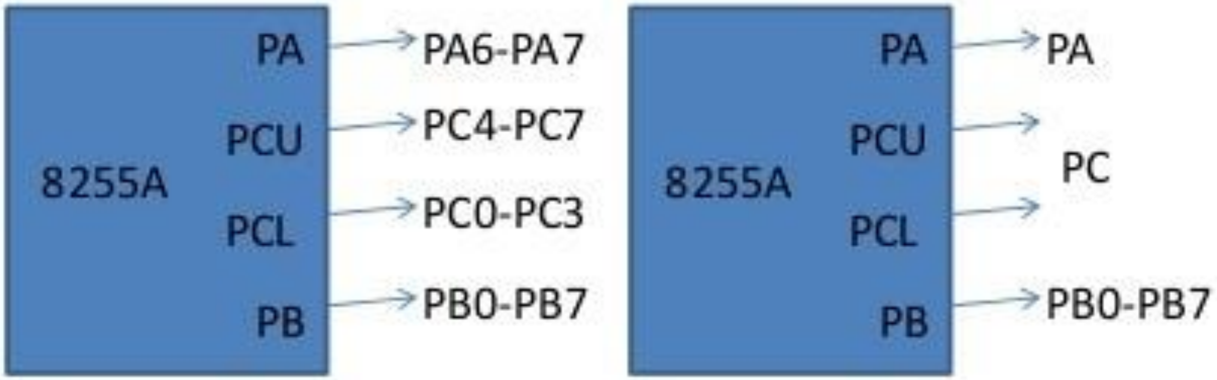


b	b	b	
0	0	0	Bit 0
0	0	1	Bit 1
0	1	0	Bit 2
0	1	1	Bit 3
1	0	0	Bit 4
1	0	1	Bit 5
1	1	0	Bit 6
1	1	1	Bit 7

BSR control word format

Mode 0 (BASIC I/O)

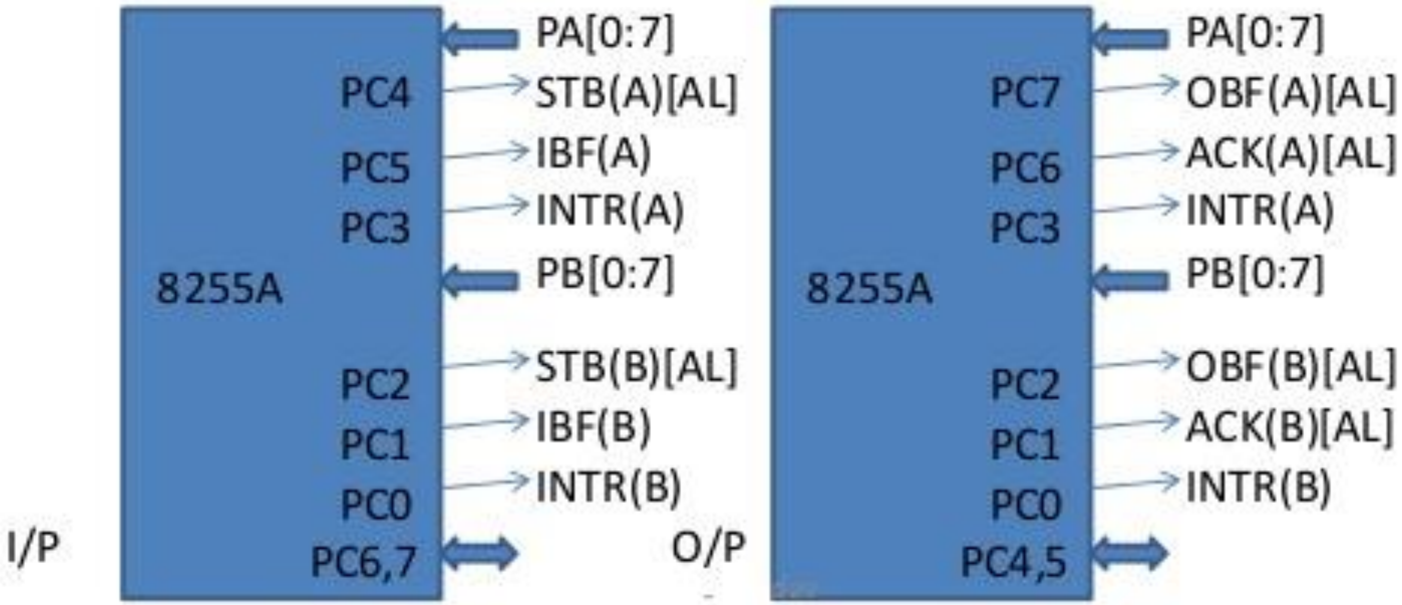
- Provides simple input and output capabilities using each of the three ports.
- Data can be simply read from and written to the input and output ports respectively, after appropriate initialization.
- No Handshaking is required.



Signals in MODE 0

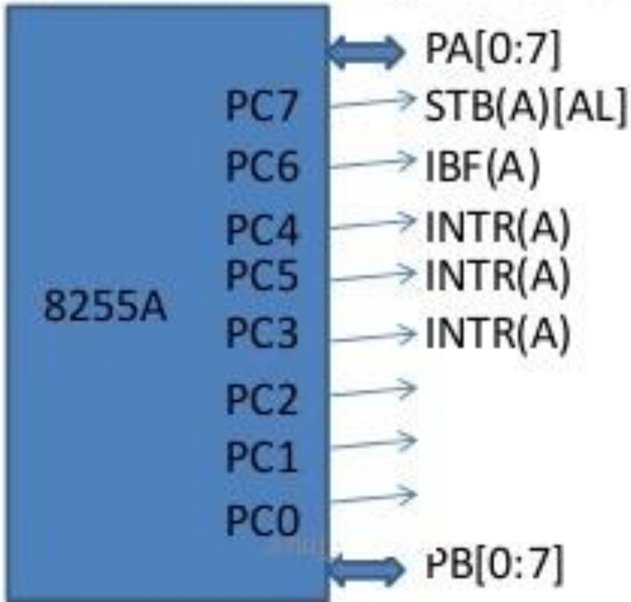
Mode 1 (STROBE I/O)

- This functional configuration provides a means for transferring I/O data to or from a specified port in conjunction with strobes or “handshaking” signals.
- In mode 1, Port A and Port B use the lines on Port C to generate or accept these “handshaking” signals.



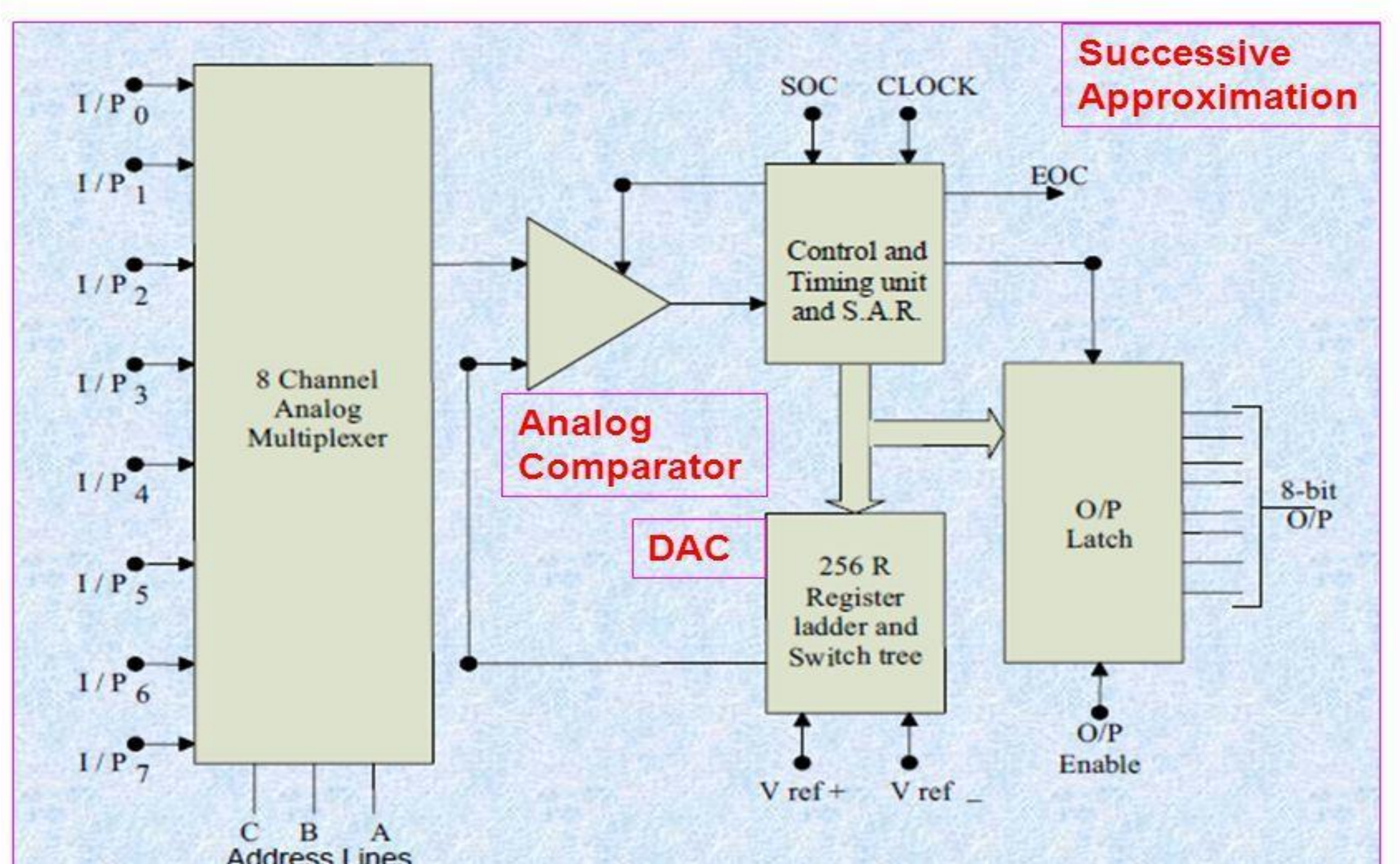
Mode 2
(STROBE BIDIRECTIONAL BUS I/O)

- This functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bidirectional bus I/O).
- “Handshaking” signals are provided to maintain proper bus flow discipline in a similar manner to MODE 1.
- Interrupt generation and enable/disable functions are also available.

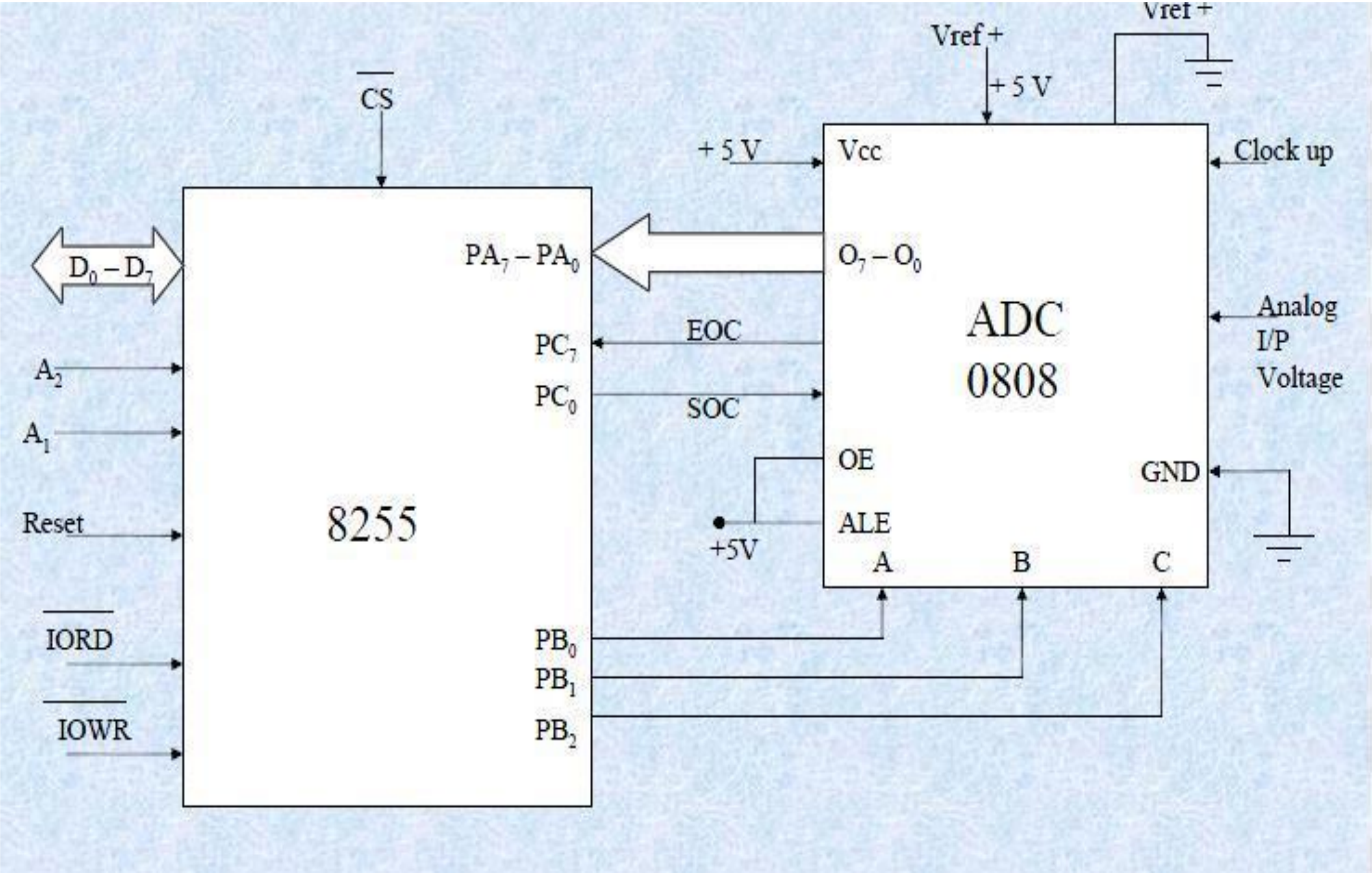


ADC/DAC INTERFACING WITH 8086

Block Diagram of ADC 0808 / 0809



ADC 0808/0809 WITH 8086 THROUGH 8255

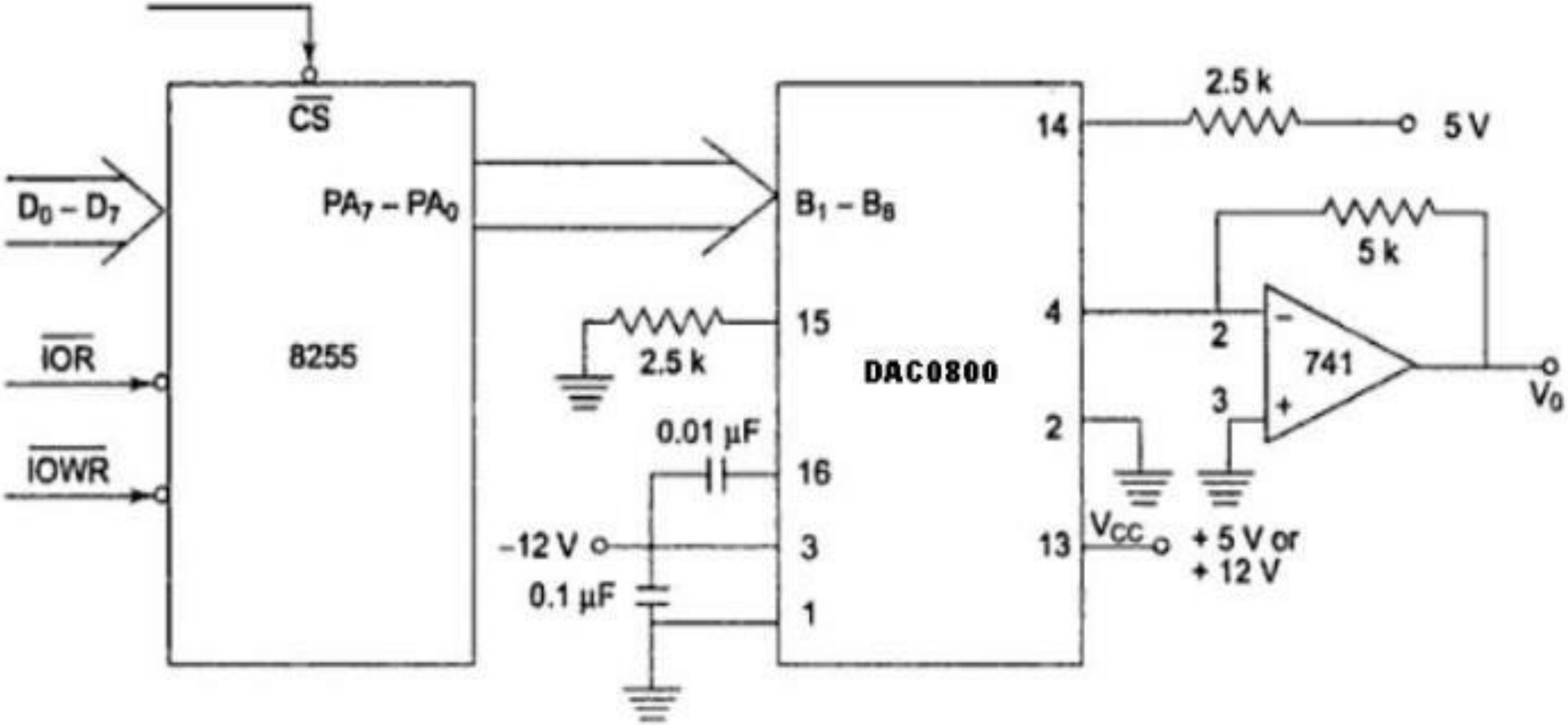


Assembly language program of ADC

```
    MOV AL,98H
    OUT CWR,AL
    MOV AL,02H
    OUT PORTB,AL
    MOV AL, 00H
    OUT PORTCL,AL
    MOV AL, 01H
    OUT PORTCL,AL
    MOV AL,00H
    OUT PORTCL, AL
    IN AL,PORTCU
    CLC
L1:RCL AL, 01H
    JNC : L1
    IN AL, PORTA
    HLT
```

Ex: Interface ADC0808 with 8086 using 8255 ports. Use portA of 8255 foe transferring Digital O/P of ADC to the CPU and portC for control signals. Assume that an analog I/P is present at I/P2 of the ADC and a clock input of suitable frequency is available for ADC. Draw the schematic and write the required ALP.

DAC interfacing



Assembly language program of DAC

EX: 8 bit DAC is connected with 8086 through port 90H write an assembly language program to generate a triangular wave at DAC o/p .

```
      MOV AL,00H
 L1: OUT 90H,AL
      INC AL
      CMP AL,FFH
      JNZ :L1
 L2:  DEC AL
      OUT 90H,AL
      CMP AL,00H
      JNZ : L2
      JMP : L1
      HLT
```

EX: Write an ALP to generate a square wave of 3V O/P in a DAC with 8-bit binary I/P and a maximum of 5V output, Assume that the addresses 80H, 82H, 84H, 86H are assigned to PORTA, B,C And CWR

```
        MOV AL,80H
        OUT 86H,AL
   L1: MOV AL,00H
        OUT 80H,AL
        CALL : DELAY
        MOV AL,99H
        OUT 80H,AL
        CALL: DELAY
        JMP: L1
```

```
DELAY: MOV CX,COUNT
     L2:  NOP
          NOP
          NOP
         LOOP:L1
          RET
```

KEY BAORD INTERFACING

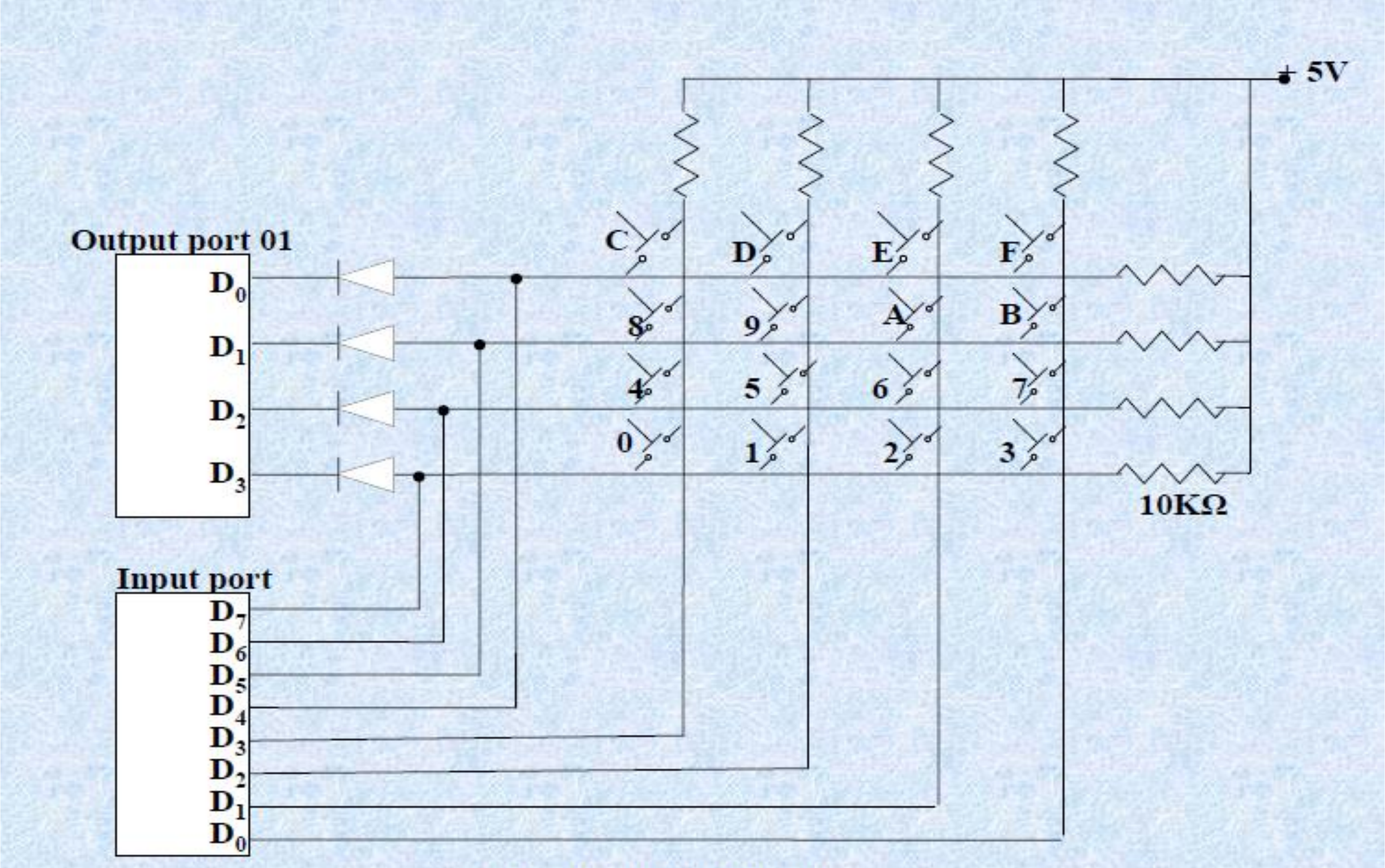
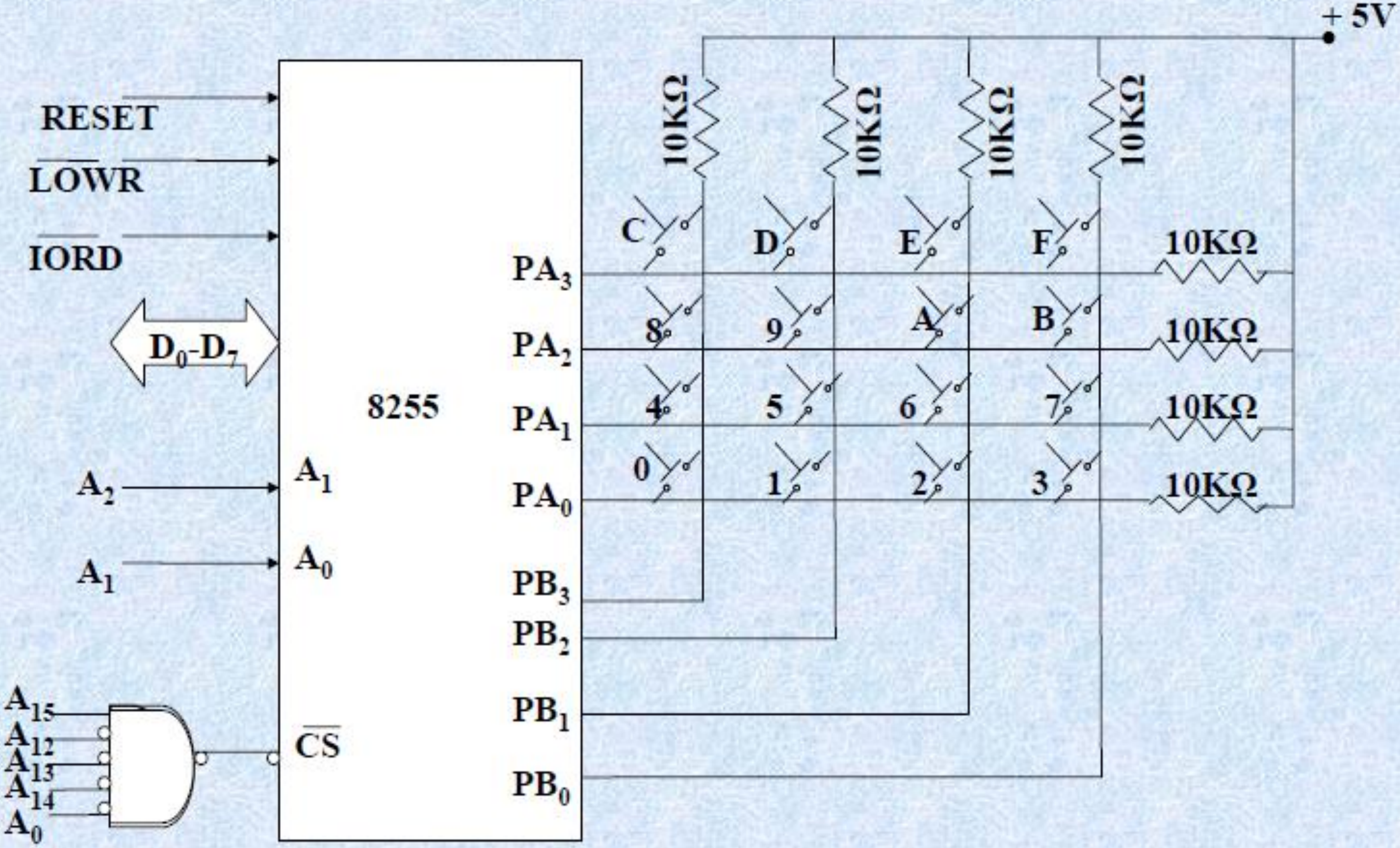


Fig: (a) Port connections

Example

- Interface a 4 * 4 keyboard with 8086 using 8255 an write an ALP for detecting a key closure and return the key code in AL. The debounce period for a key is 10ms. Use software debouncing technique. DEBOUNCE is an available 10ms delay routine.
- **Solution**: Port A is used as output port for selecting a row of keys while Port B is used as an input port for sensing a closed key. Thus the keyboard lines are selected one by one through port A and the port B lines are polled continuously till a key closure is sensed. The routine DEBOUNCE is called for key debouncing. The key code is depending upon the selected row and a low sensed column.



Interfacing 4 * 4 Keyboard

```
          MOV AL,82H
          OUT 86H, AL
START:    MOV AL,00H }
          MOV 80H,AL } clear all rows by sending 00H to PORTA
NEXT:     IN     AL,82H } obtain status of columns in AL by reading PORTB
          AND AL,0FH } mask upper nibble in AL
          CMP AL,0FH } compare Al with 0F to identify whether any key is pressed
          JZ  NEXT      } If ZF=1,then no key is pressed; go to location back ,else  go to
                          next step to find the key number which is pressed
           CALL : DELAY
           MOV BL,00H ; store first key number in row0 ( 0th key)
           MOV AL,FEH ; ground row0 alone by sending FEH to PORTA
           OUT 80H,AL
            IN AL,82H
            AND AL,0FH
            CMP AL,0FH
            JNZ : FIND
            MOV BL,04H
            MOV AL,FDH
            OUT 80H,AL
             IN AL,82H
```

```
     AND AL,0FH
     CMP AL,0FH
     JNZ : FIND
     MOV AL,08H
     MOV AL,FBH
     OUT 80H,AL
     IN AL,82H
     AND AL,0FH
     CMP AL,0FH
     JNZ :FIND
     MOV BL,0CH
     MOV AL,F7H
     OUT 80H,AL
     IN AL,0FH
     AND AL,0FH
     CMP AL,0FH
     JNZ: FIND
     JMP: START
     MOV CX,COUNT
L1   NOP
     NOP
     LOOP :L1
      RET
```

```
FIND : RCR AL,01H
       JNC GOT_KEY
       INC BL
       JMP : FIND
GOT :  RET
```

DATA TRANSFER SCHEMES

- There will be several IO and memory devices connected to transfer data between memory and mp

- No problem for transferring data between MP and memory since same technology is used in the memory and MP. Speed of both is compatible.

- Data transfer between the MP and IO devices is problematic because the Speed of the IO devices and the speed of MP or memory is mismatch.

- To overcome the speed problem we have different Modes of data Transfer.

NEEDS OF DATA TRANSFER SCHEME

- A wide variety of IO devices having wide range of speed and other different characteristics are available .

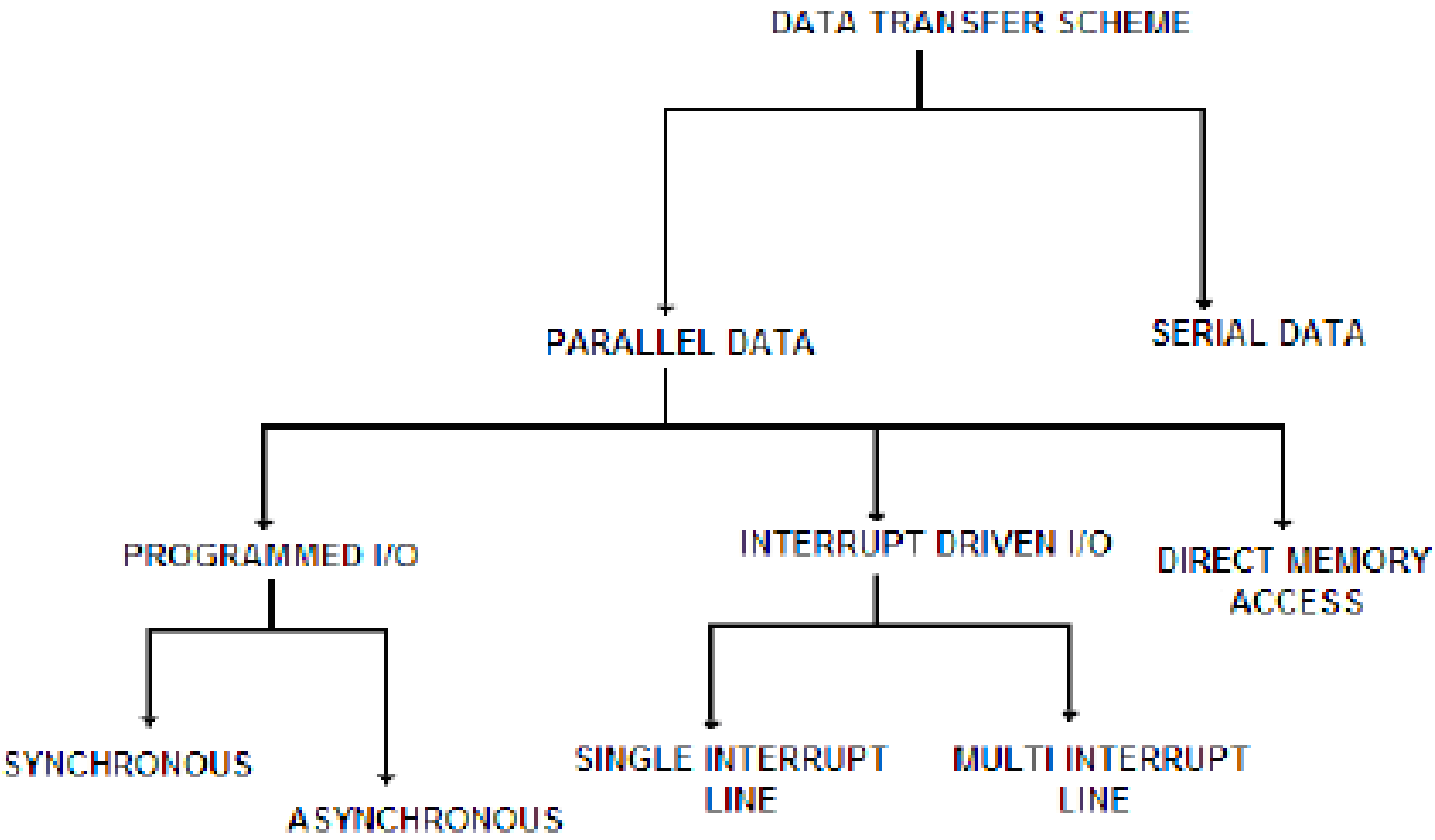
- A slow responding IO device cannot transfer data when microprocessor issues instruction for it as it takes some time to get ready.

- Data codes and formats in peripheral differ from the word format of in the central processing unit and memory.

- Transfers rates of peripherals is usually slower than the transfer rates of central processing unit.

- Operating modes of peripheral are different from each other and each must be controlled so as not to disturb the operation of each other peripherals connected to central processing unit .

Modes of data transfer schemes



8251 USART

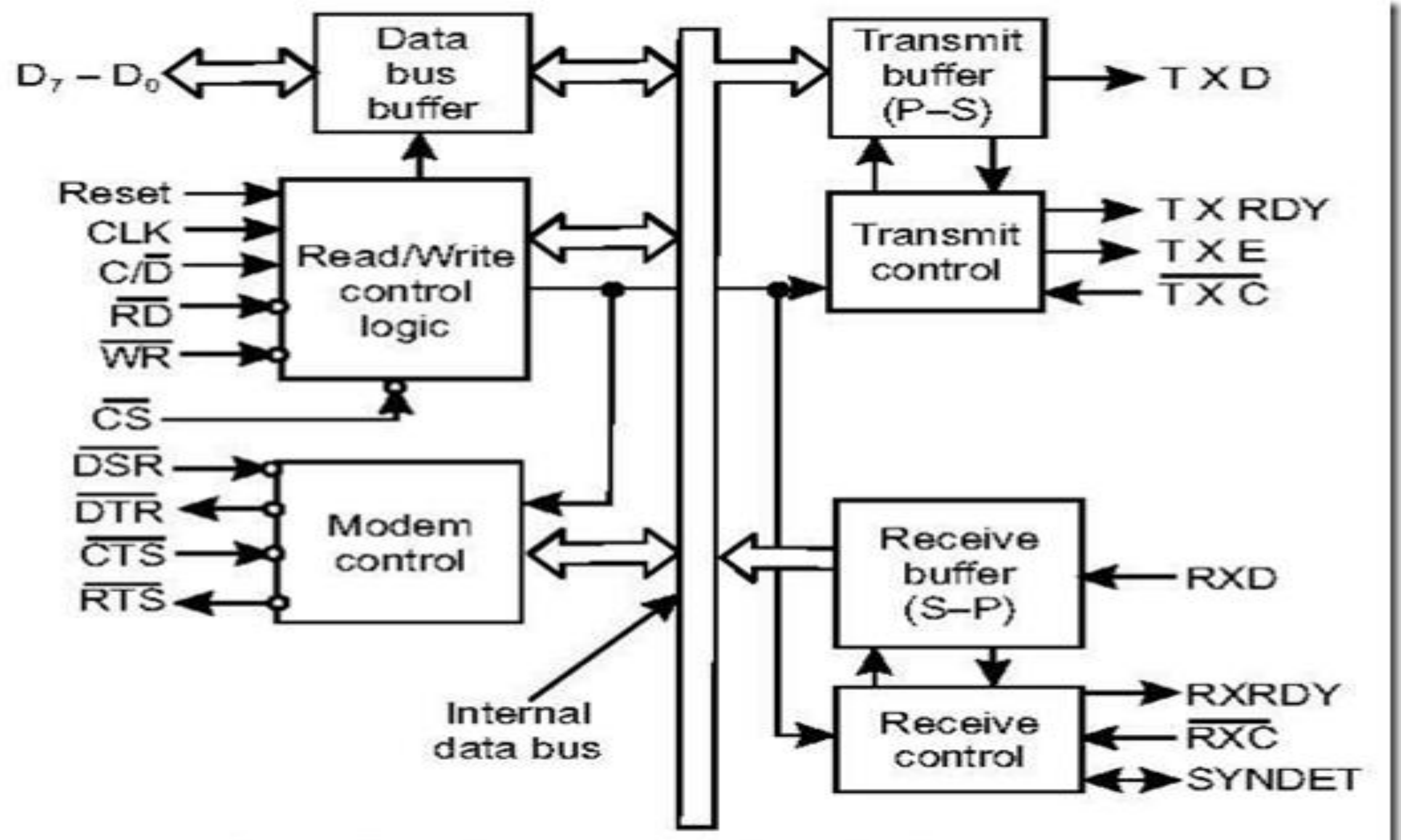


Fig. 9i.2: Functional block diagram of 8251

CONTROL WORDS

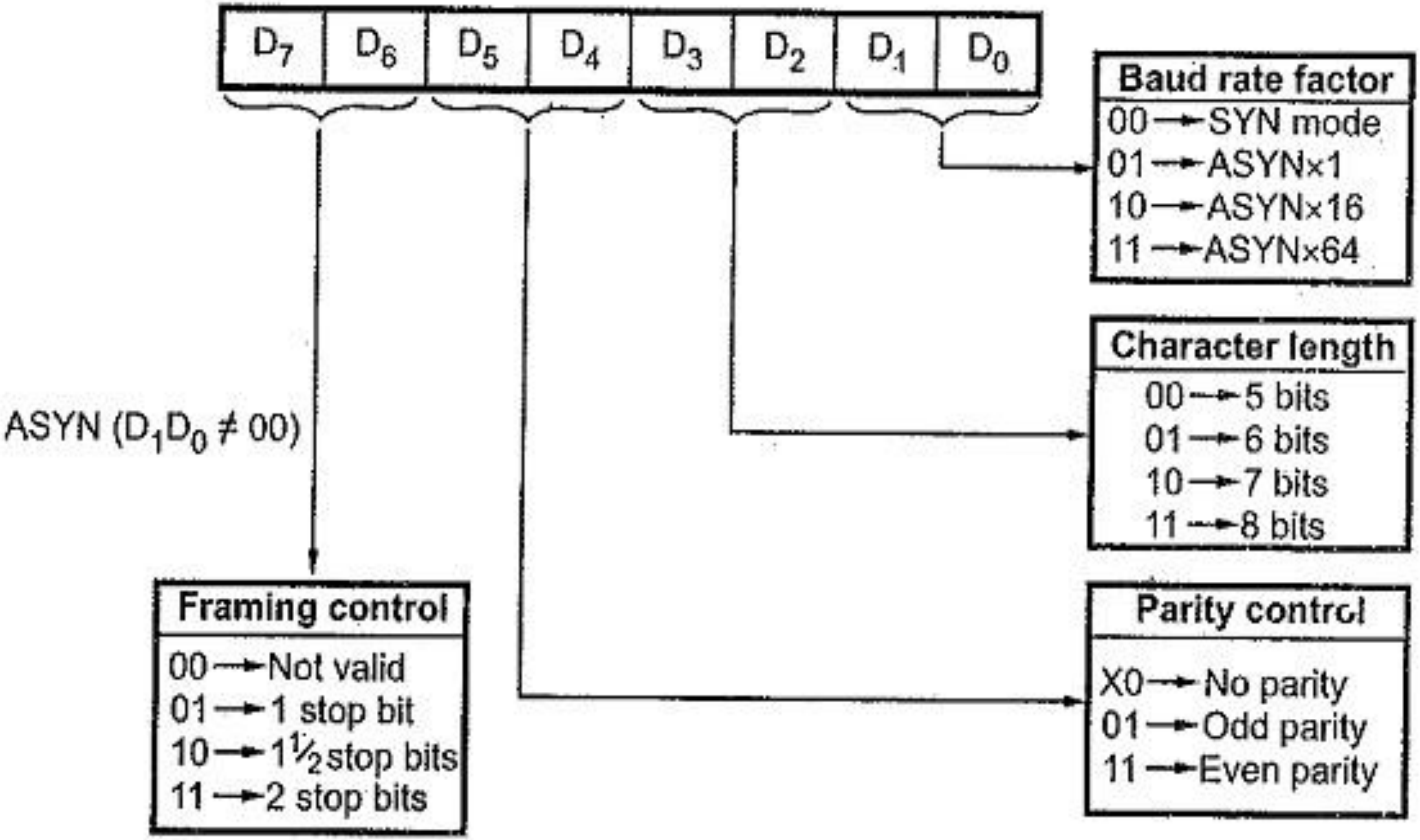
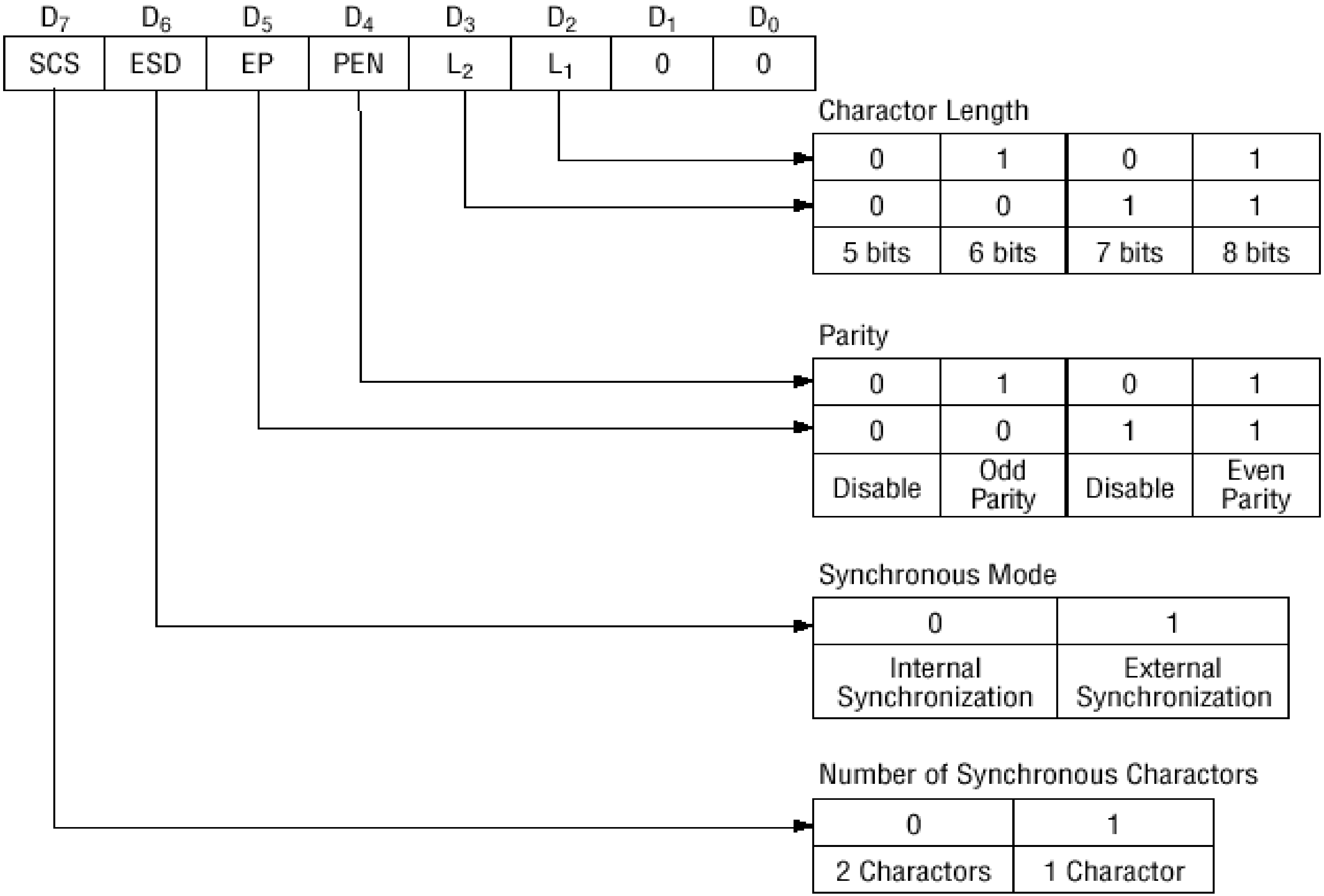


Fig. 14.38 Mode instruction format



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SCS	ESD	EP	PEN	L_2	L_1	0	0

Charactor Length

0	1	0	1
0	0	1	1
5 bits	6 bits	7 bits	8 bits

Parity

0	1	0	1
0	0	1	1
Disable	Odd Parity	Disable	Even Parity

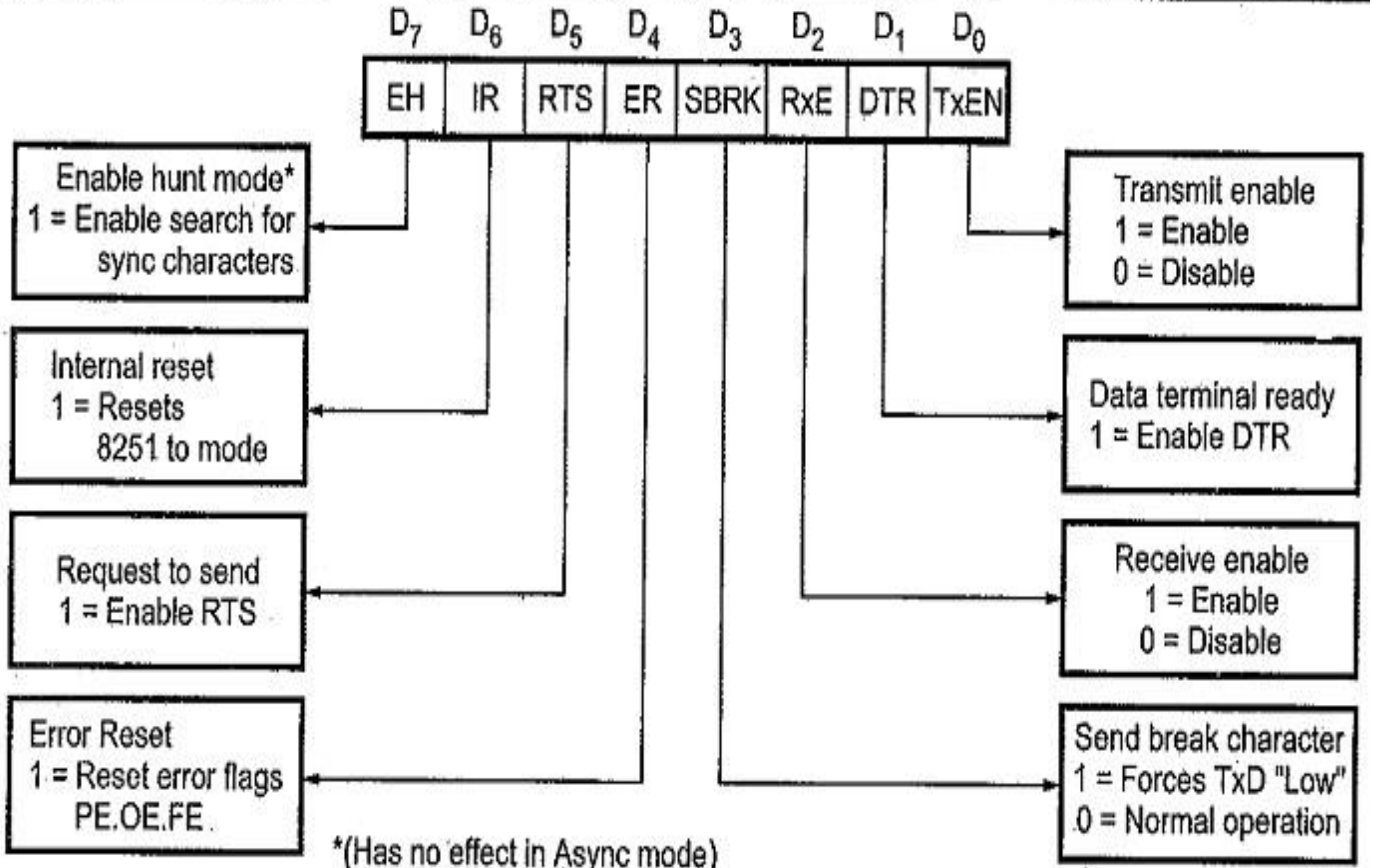
Synchronous Mode

0	1
Internal Synchronization	External Synchronization

Number of Synchronous Charactors

0	1
2 Charactors	1 Charactor

Fig. 3 Bit Configuration of Mode Instruction (Synchronous)

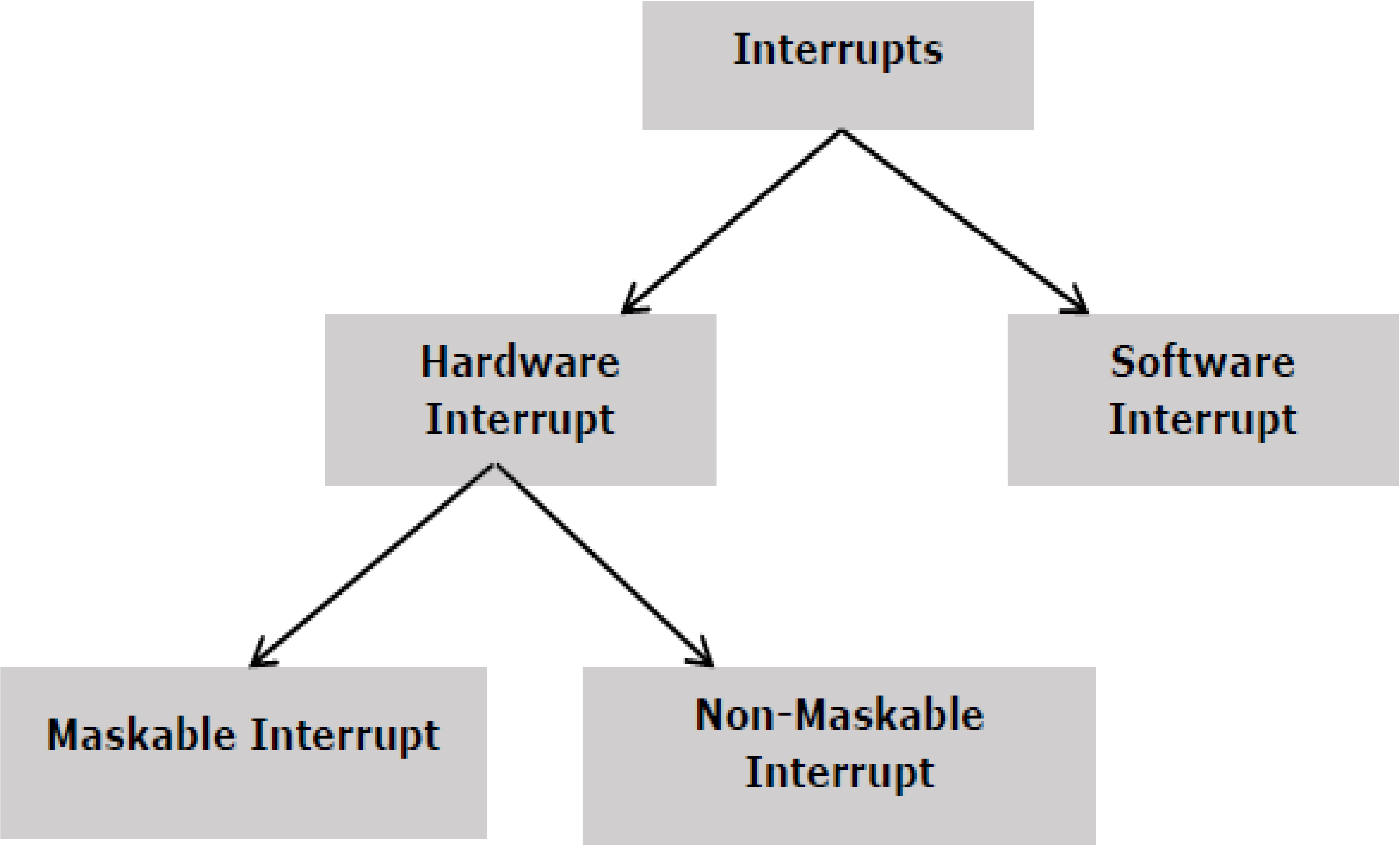


*(Has no effect in Async mode)

Note : Error reset must be performed whenever RX enable and enter hunt are programmed

Fig. 14.39 Command instruction format

INTERRUPT STRUCTURE OF 8086



When the interrupt is activated, these actions take place

- Completes the current instruction that is in progress.
- Pushes the Flag register values on to the stack.
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.
- IP is loaded from the contents of the word location 00008H.
- CS is loaded from the contents of the next word location 0000AH.

Interrupt flag and trap flag are reset to 0.

Hardware interrupt

NMI

INTR

Software interrupts (INT N)

256 types of software interrupt

TYPE 0 (divide by zero interrupt)

TYPE 1 (single step execution)

TYPE 2 (non-maskable interrupt)

TYPE 3 (break point interrupt)

TYPE 4 (over flow interrupt)

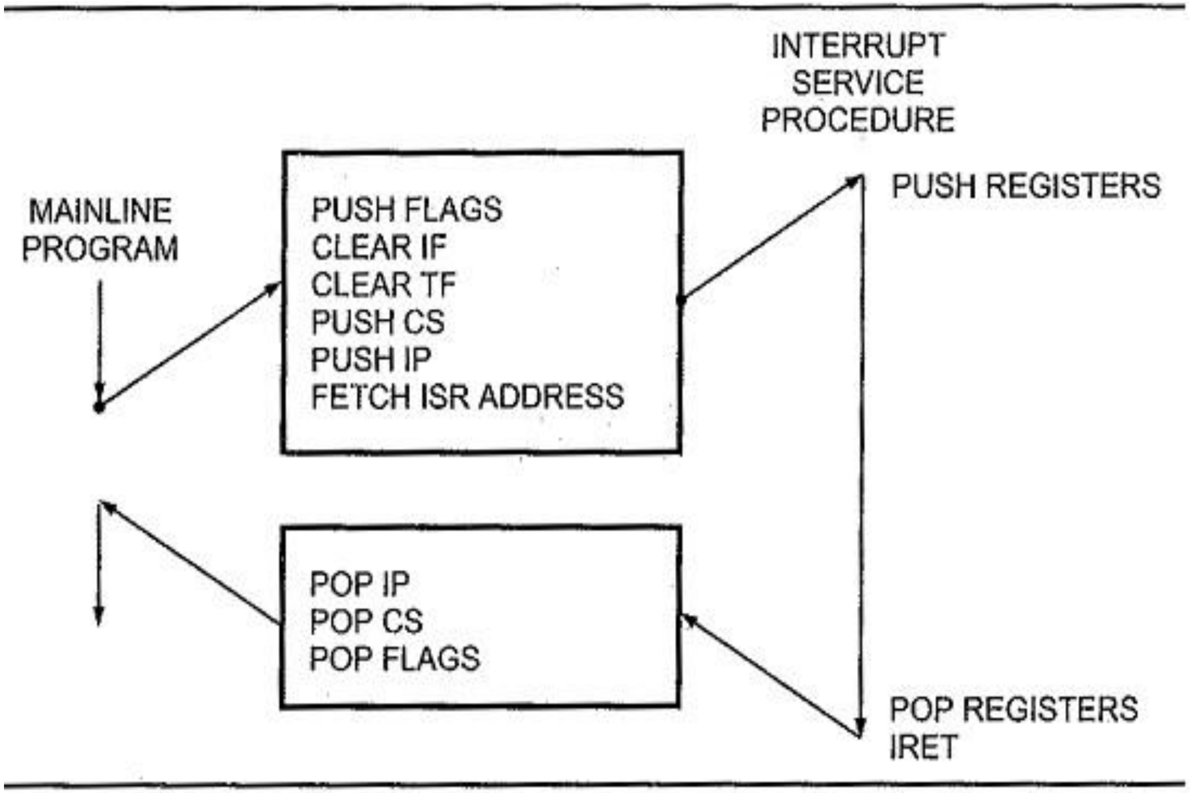
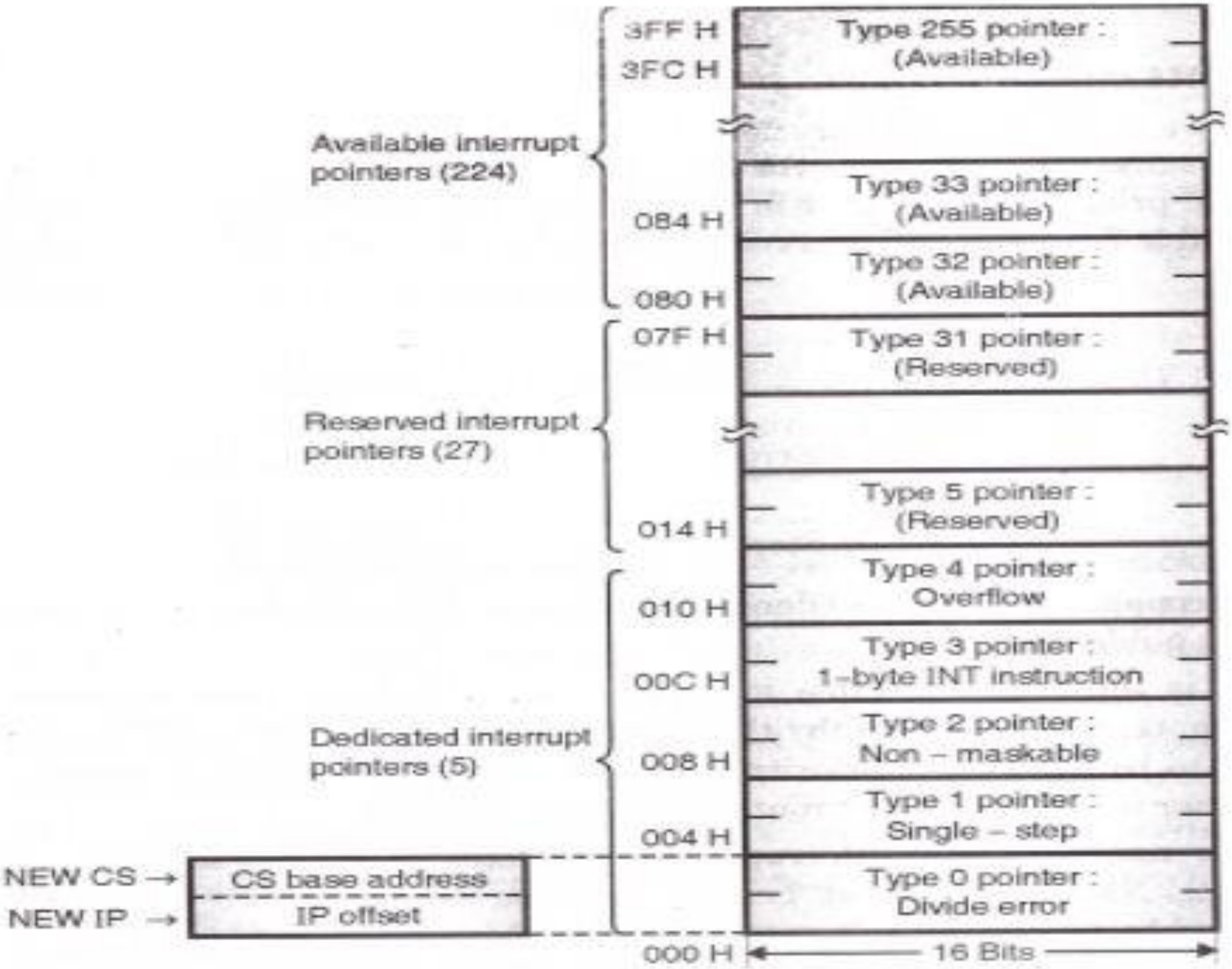


Fig. 9.1 8086 interrupt response

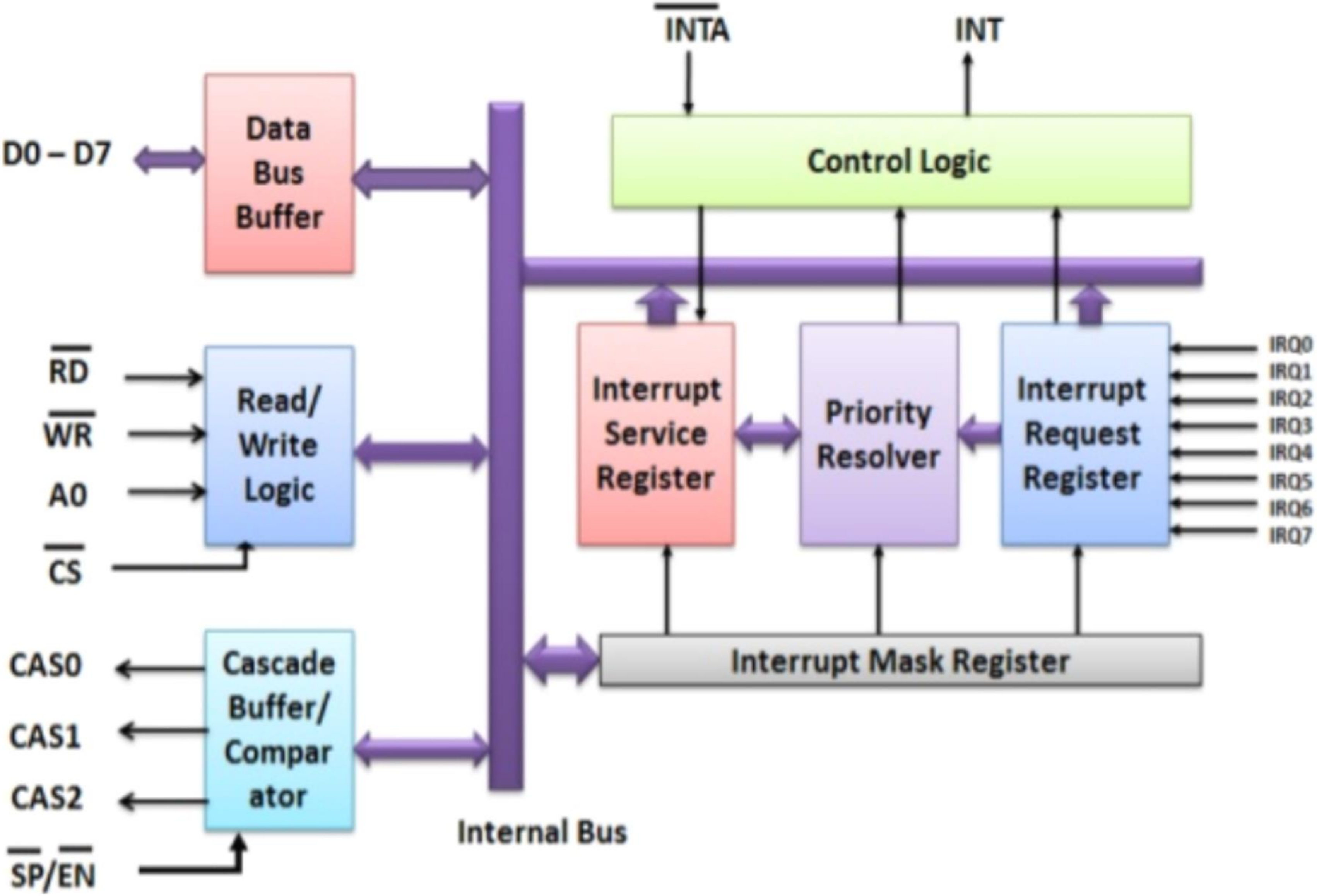
Interrupt	Priority
Divide Error, INT(n),INTO	Highest
NMI	↓
INTR	↓
Single Step	Lowest

INTERRUPT VECTOR TABLE

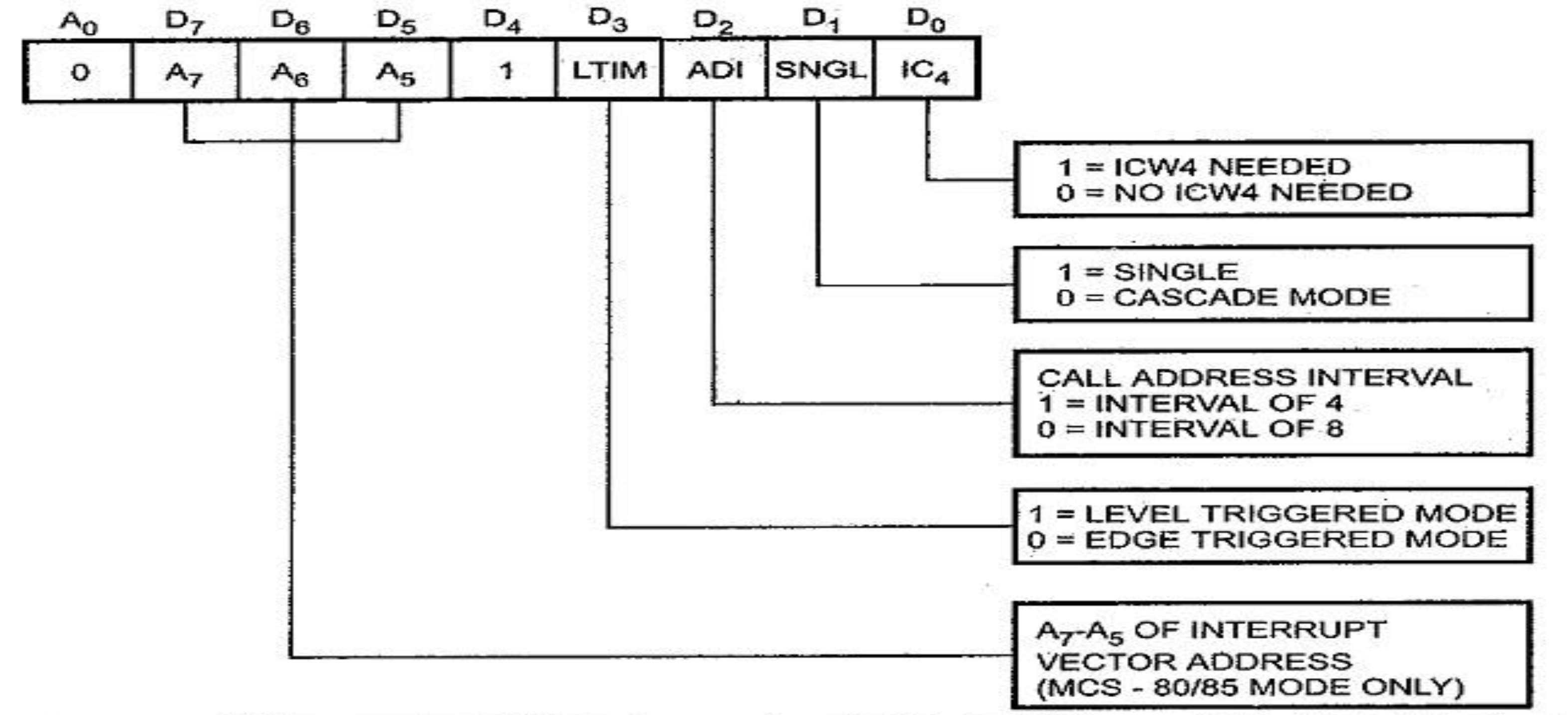

3FF H
3FC H
Type 255 pointer : (Available)
Available interrupt pointers (224)
Type 33 pointer : (Available)
084 H
Type 32 pointer : (Available)
080 H
07F H
Type 31 pointer : (Reserved)
Reserved interrupt pointers (27)
Type 5 pointer : (Reserved)
014 H
Type 4 pointer : Overflow
010 H
Type 3 pointer : 1-byte INT instruction
00C H
Dedicated interrupt pointers (5)
Type 2 pointer : Non – maskable
008 H
Type 1 pointer : Single – step
004 H
NEW CS →
CS base address
NEW IP →
IP offset
Type 0 pointer : Divide error
000 H
16 Bits

Interrupt vector table

Block Diagram Architecture of 8259



ICWS AND OCWS OF 8059



A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC_4

1 = ICW4 NEEDED
0 = NO ICW4 NEEDED

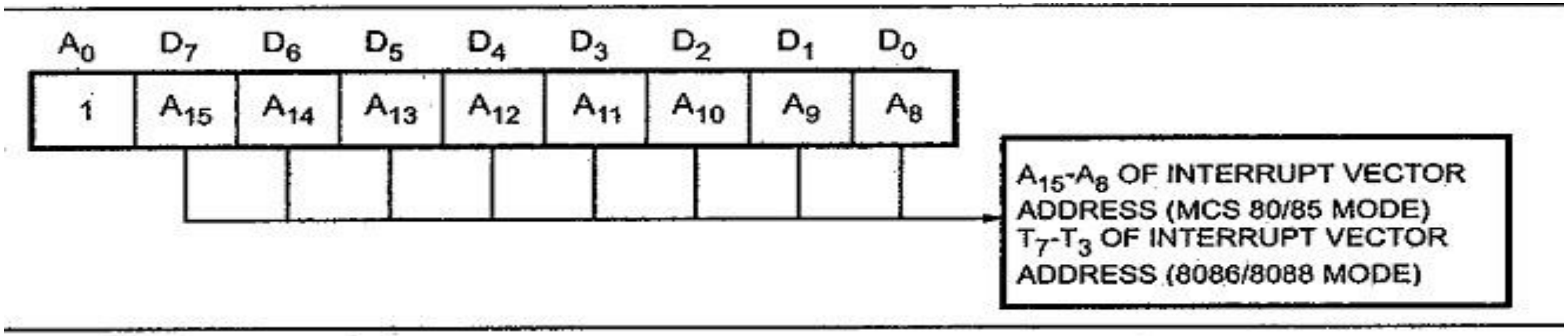
1 = SINGLE
0 = CASCADE MODE

CALL ADDRESS INTERVAL
1 = INTERVAL OF 4
0 = INTERVAL OF 8

1 = LEVEL TRIGGERED MODE
0 = EDGE TRIGGERED MODE

A_7-A_5 OF INTERRUPT
VECTOR ADDRESS
(MCS - 80/85 MODE ONLY)

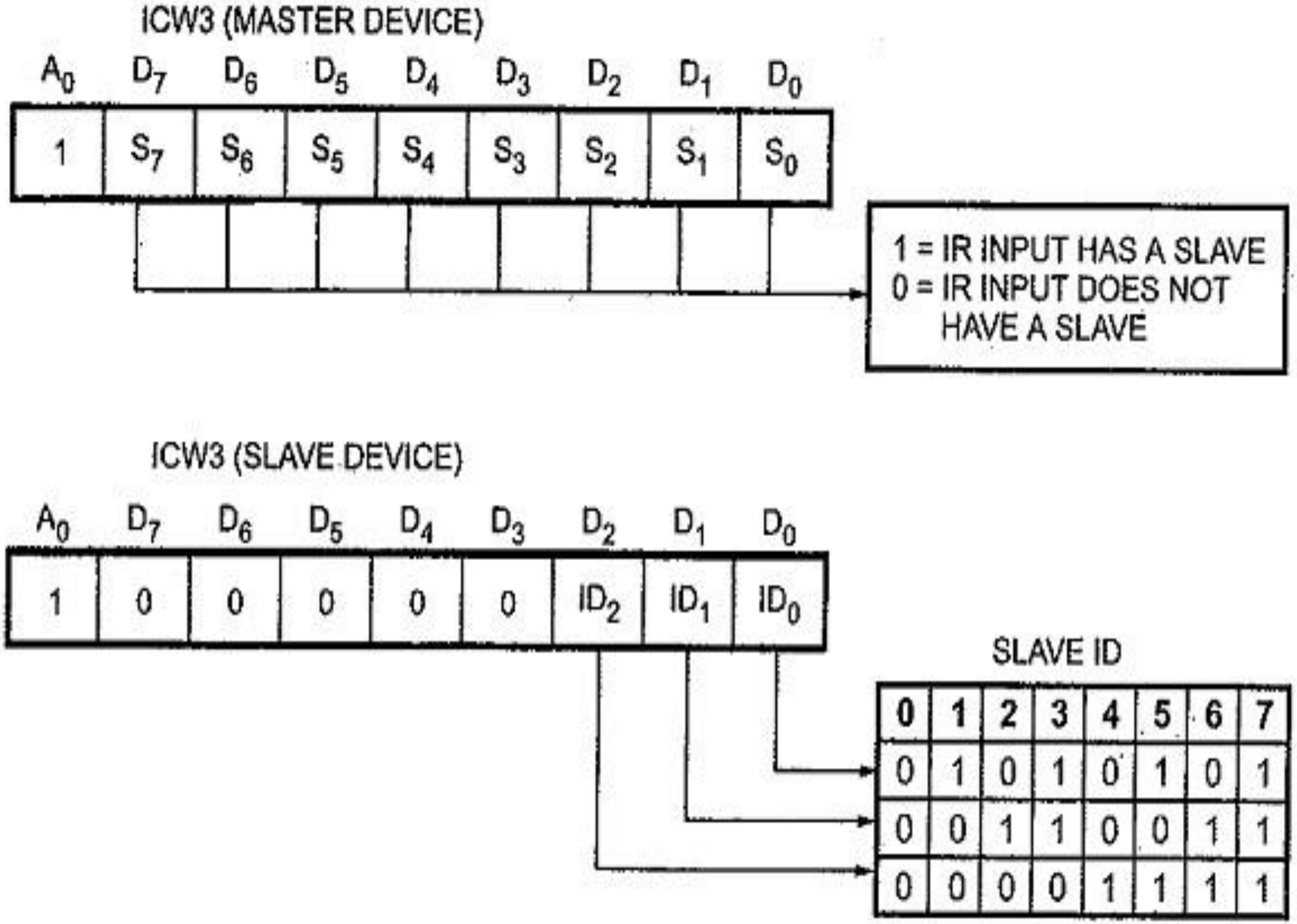
Fig. 14.76 Initialization command word 1 (ICW1)



A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8

A_{15}-A_8 OF INTERRUPT VECTOR
ADDRESS (MCS 80/85 MODE)
T_7-T_3 OF INTERRUPT VECTOR
ADDRESS (8086/8088 MODE)

Fig. 14.77 Initialization command word 2 (ICW2)



ICW3 (MASTER DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

1 = IR INPUT HAS A SLAVE
0 = IR INPUT DOES NOT
HAVE A SLAVE

ICW3 (SLAVE DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	0	0	ID_2	ID_1	ID_0

SLAVE ID

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

Fig. 14.78 Initialization command word 3 (ICW3)

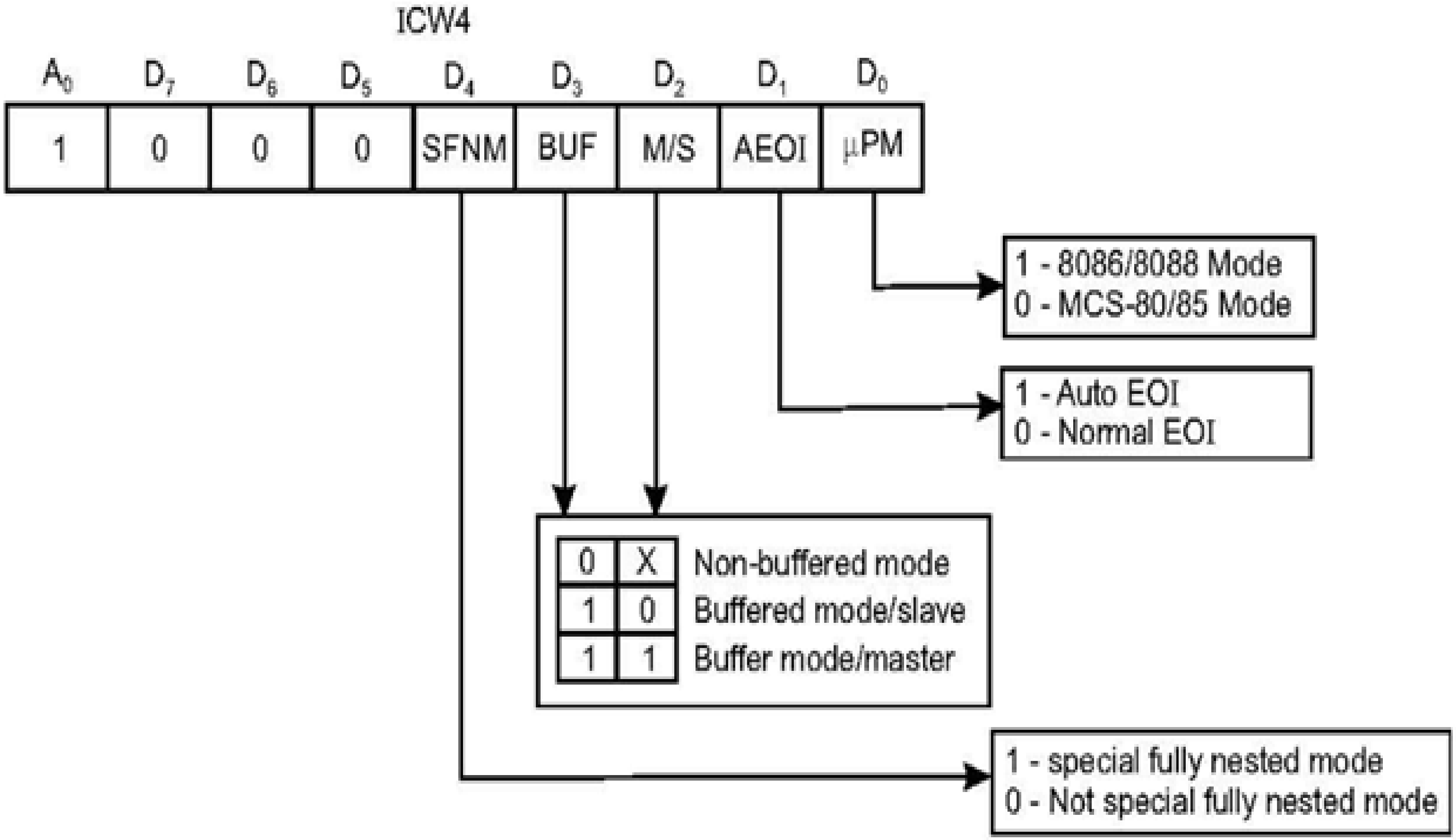
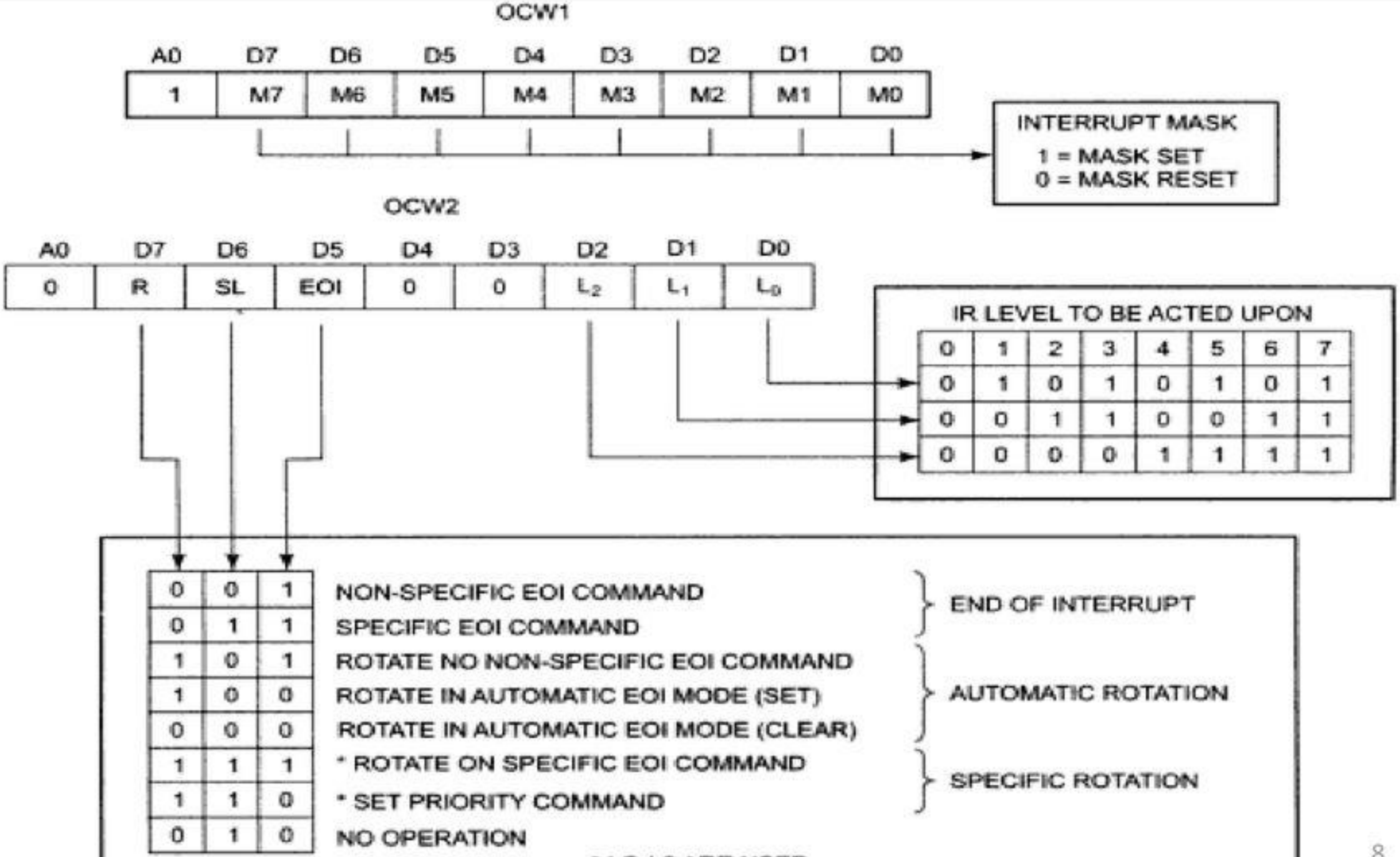


Fig. 9e.11: Initialisation command word 4
(*Source:* Intel Corporation)

Operational command word-ocw 1 and ocw 2



OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

INTERRUPT MASK
1 = MASK SET
0 = MASK RESET

OCW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	R	SL	EOI	0	0	L_2	L_1	L_0

IR LEVEL TO BE ACTED UPON

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

0	0	1	NON-SPECIFIC EOI COMMAND	END OF INTERRUPT
0	1	1	SPECIFIC EOI COMMAND	
1	0	1	ROTATE NO NON-SPECIFIC EOI COMMAND	AUTOMATIC ROTATION
1	0	0	ROTATE IN AUTOMATIC EOI MODE (SET)	
0	0	0	ROTATE IN AUTOMATIC EOI MODE (CLEAR)	
1	1	1	* ROTATE ON SPECIFIC EOI COMMAND	SPECIFIC ROTATION
1	1	0	* SET PRIORITY COMMAND	
0	1	0	NO OPERATION	

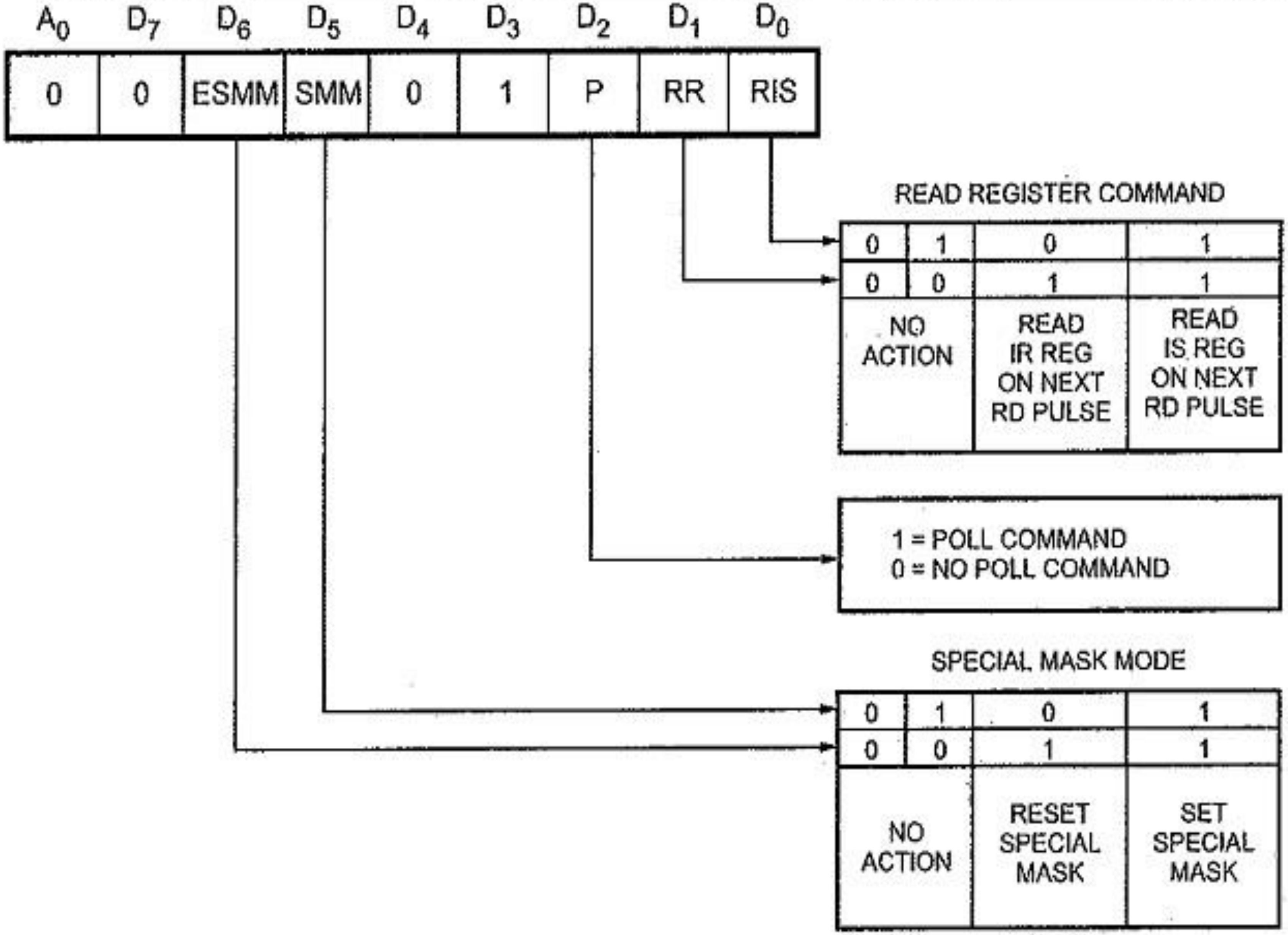


Fig. 14.82 Operation Command word 3 (OCW3)

INTRODUCTION TO 8051 MICROCONTROLLER UNIT- IV

DIFFERENCE BETWEEN MICROPROCESSOR AND MICROCONTROLLER

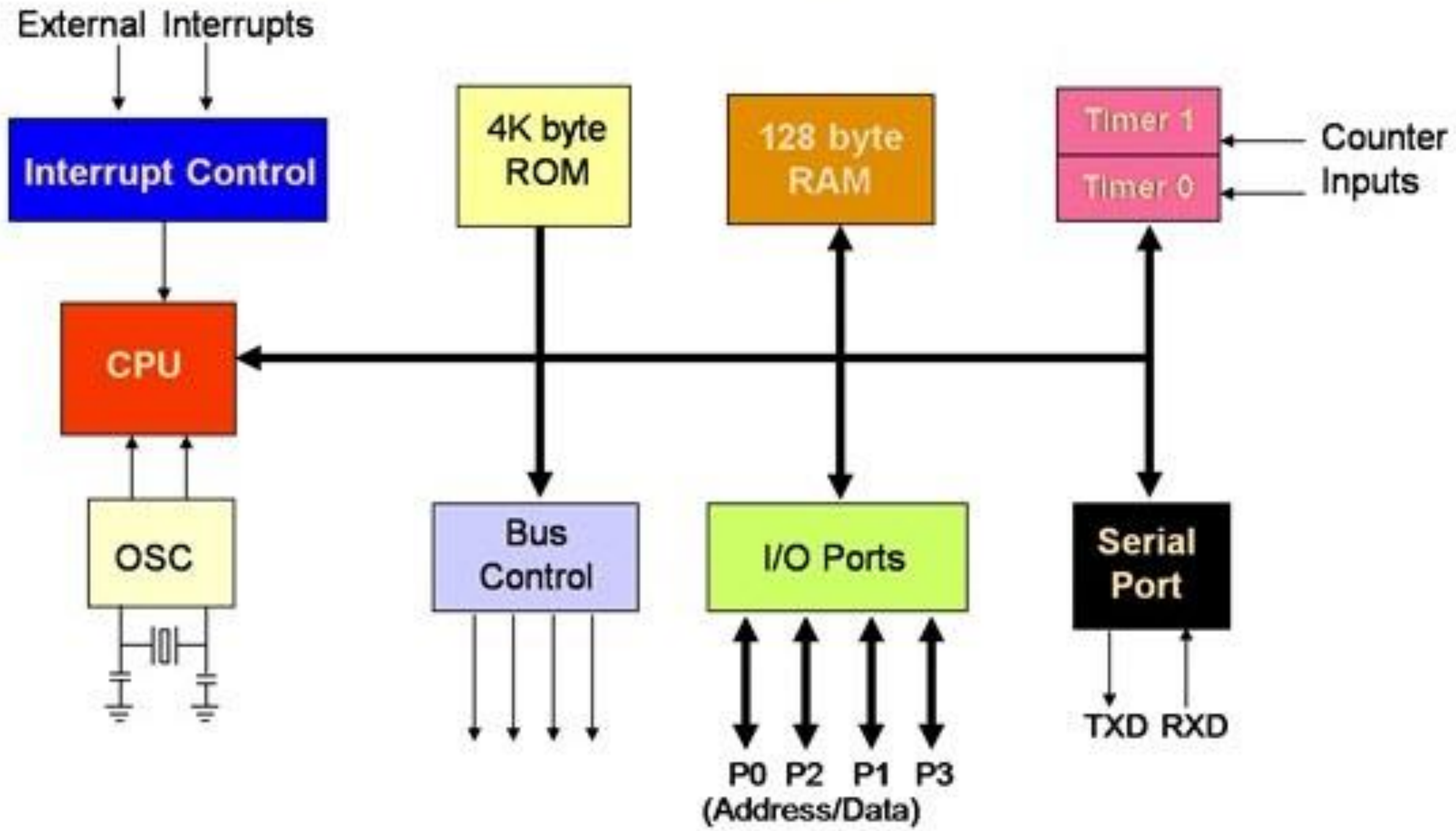
Microcontroller	Microprocessor
Microcontrollers are used to execute a single task within an application.	Microprocessors are used for big applications.
Its designing and hardware cost is low.	Its designing and hardware cost is high.
Easy to replace.	Not so easy to replace.
It is built with CMOS technology, which requires less power to operate	Its power consumption is high because it has to control the entire system
It consists of CPU, RAM, ROM, I/O ports.	It doesn’t consist of RAM, ROM, I/O ports. It uses its pins to interface to peripheral devices.

COMMON MICROCONTROLLERS

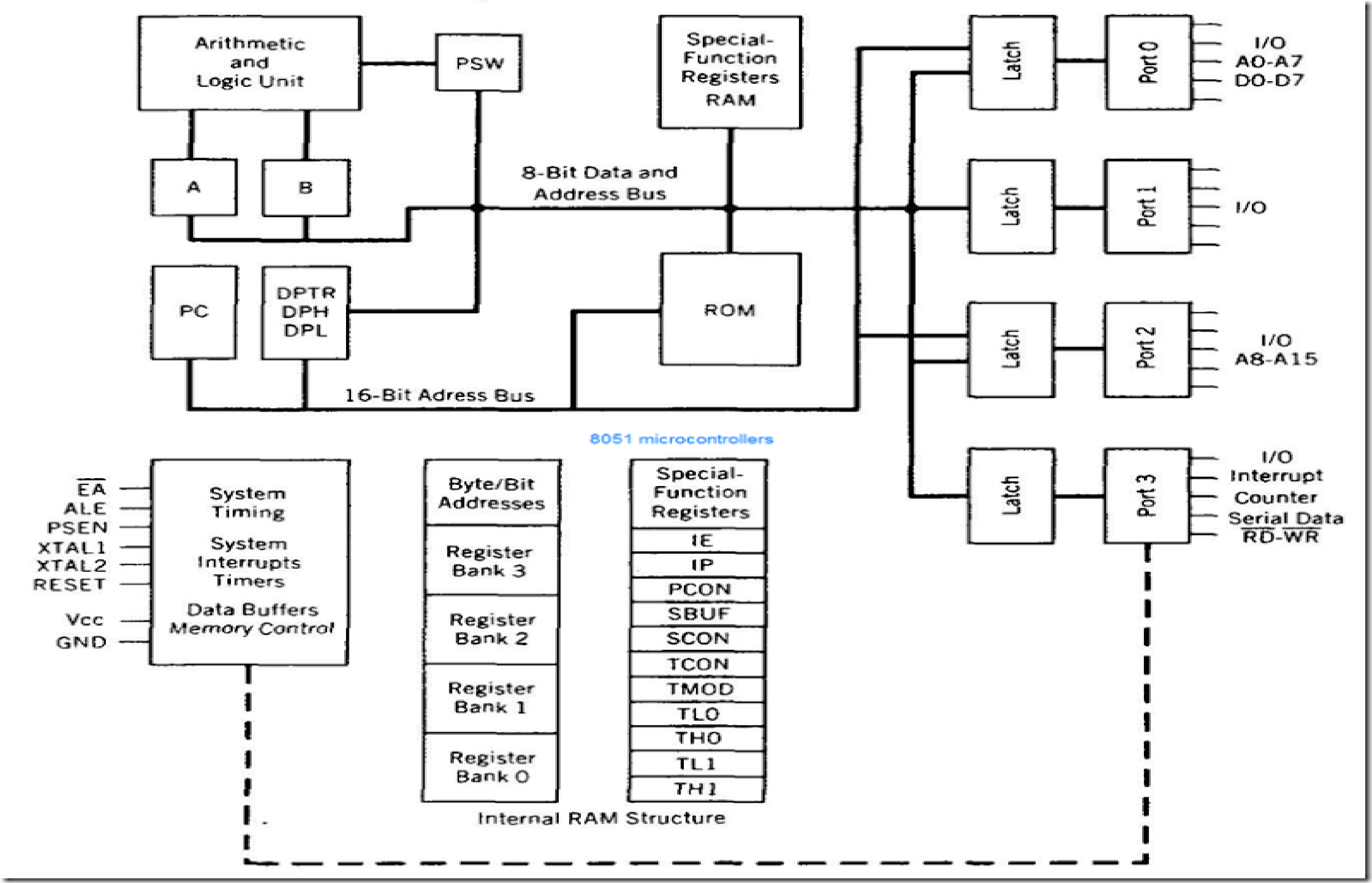
- Atmel
- ARM
- Intel
  - 8-bit
    - 8XC42
    - MCS48
    - **MCS51**
    - 8xC251
  - 16-bit
    - MCS96
    - MXS296
- National Semiconductor
  - COP8
- Microchip
  - 12-bit instruction PIC
  - 14-bit instruction PIC
    - PIC16F84
  - 16-bit instruction PIC
- NEC

- Motorola
  - 8-bit
    - 68HC05
    - 68HC08
    - 68HC11
  - 16-bit
    - 68HC12
    - 68HC16
  - 32-bit
    - 683xx
- Texas Instruments
  - TMS370, 16/32 bit
  - MSP430 , 16 bit
- Zilog
  - Z8
  - Z86E02

BLOCK DIAGRAM OF 8051

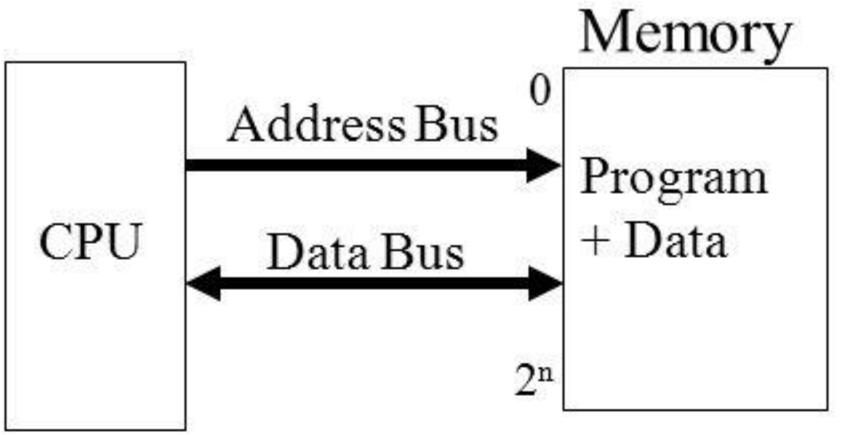


ARCHITECTURE OF 8051

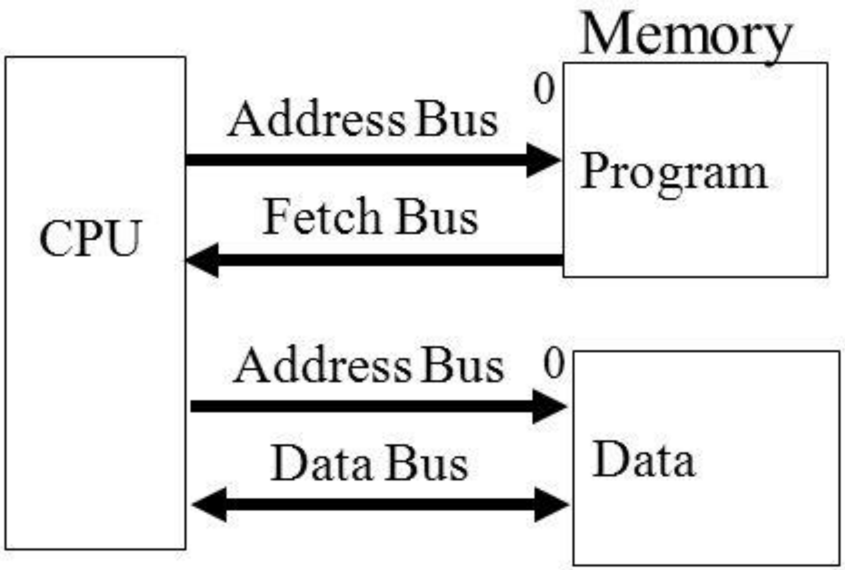


Internal RAM Structure

Microcontroller Architectures

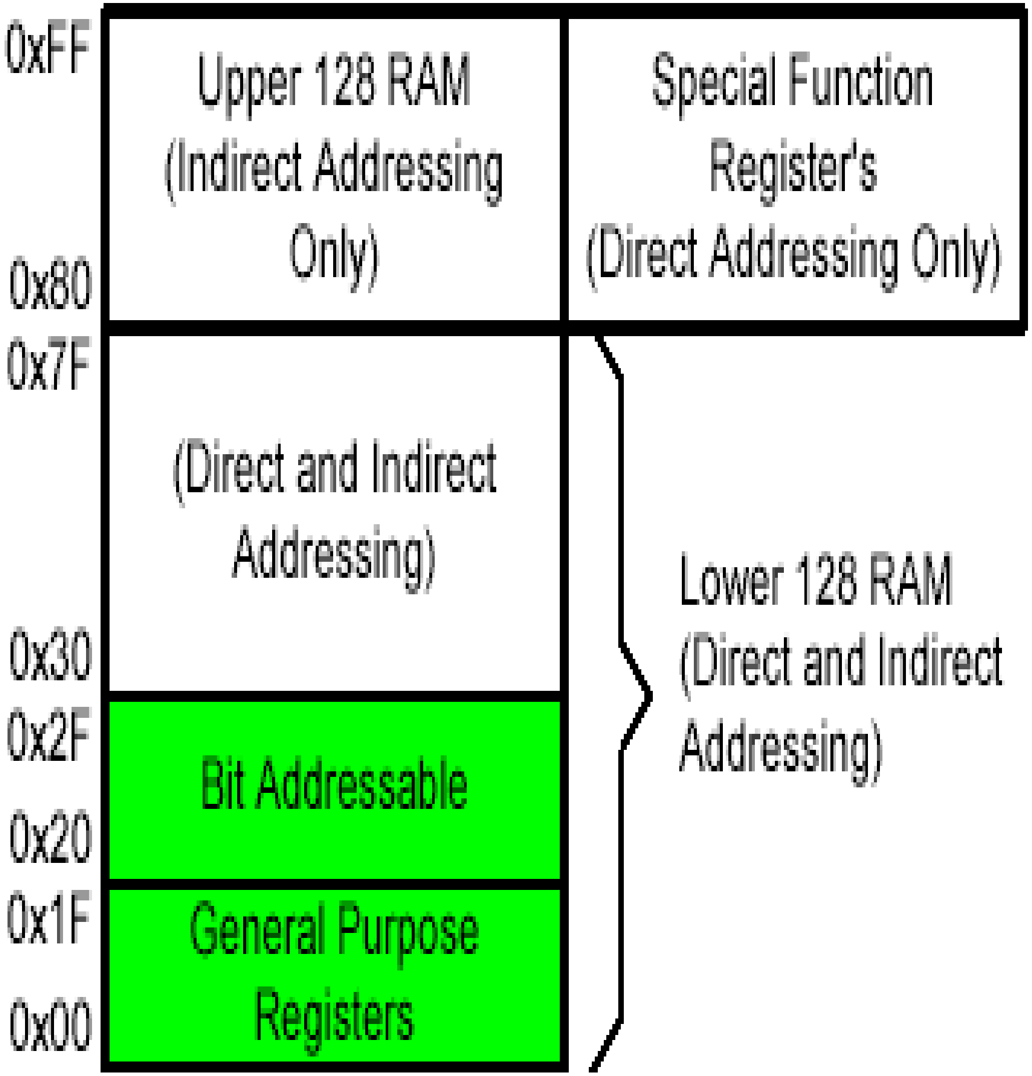


Von Neumann Architecture



Harvard Architecture

ON-CHIP DATA MEMORY: RAM



0xFF

Upper 128 RAM (Indirect Addressing Only)

Special Function Register's (Direct Addressing Only)

0x80

0x7F

(Direct and Indirect Addressing)

Lower 128 RAM (Direct and Indirect Addressing)

0x30

0x2F

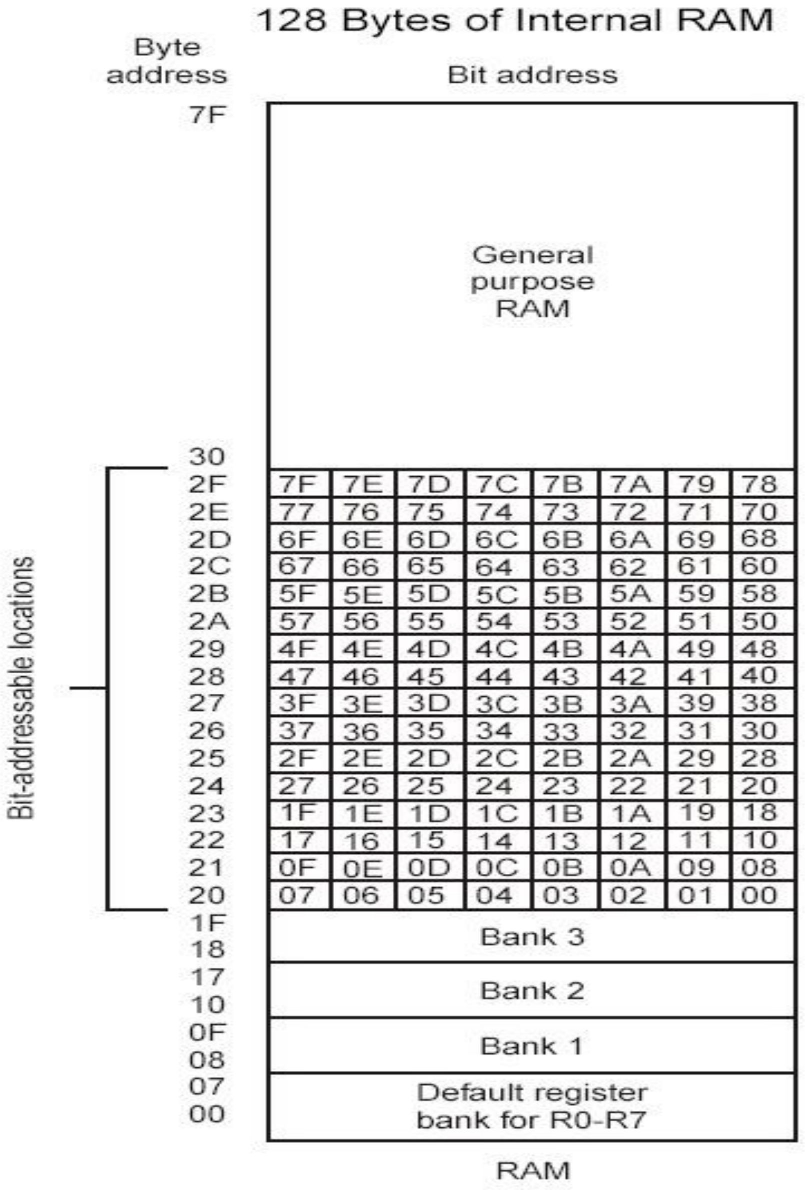
Bit Addressable

0x20

0x1F

General Purpose Registers

0x00



128 Bytes of Internal RAM

Byte address

7F

General purpose RAM

30

Bit-addressable locations

Byte address	Bit address							
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00

1F
Bank 3
18

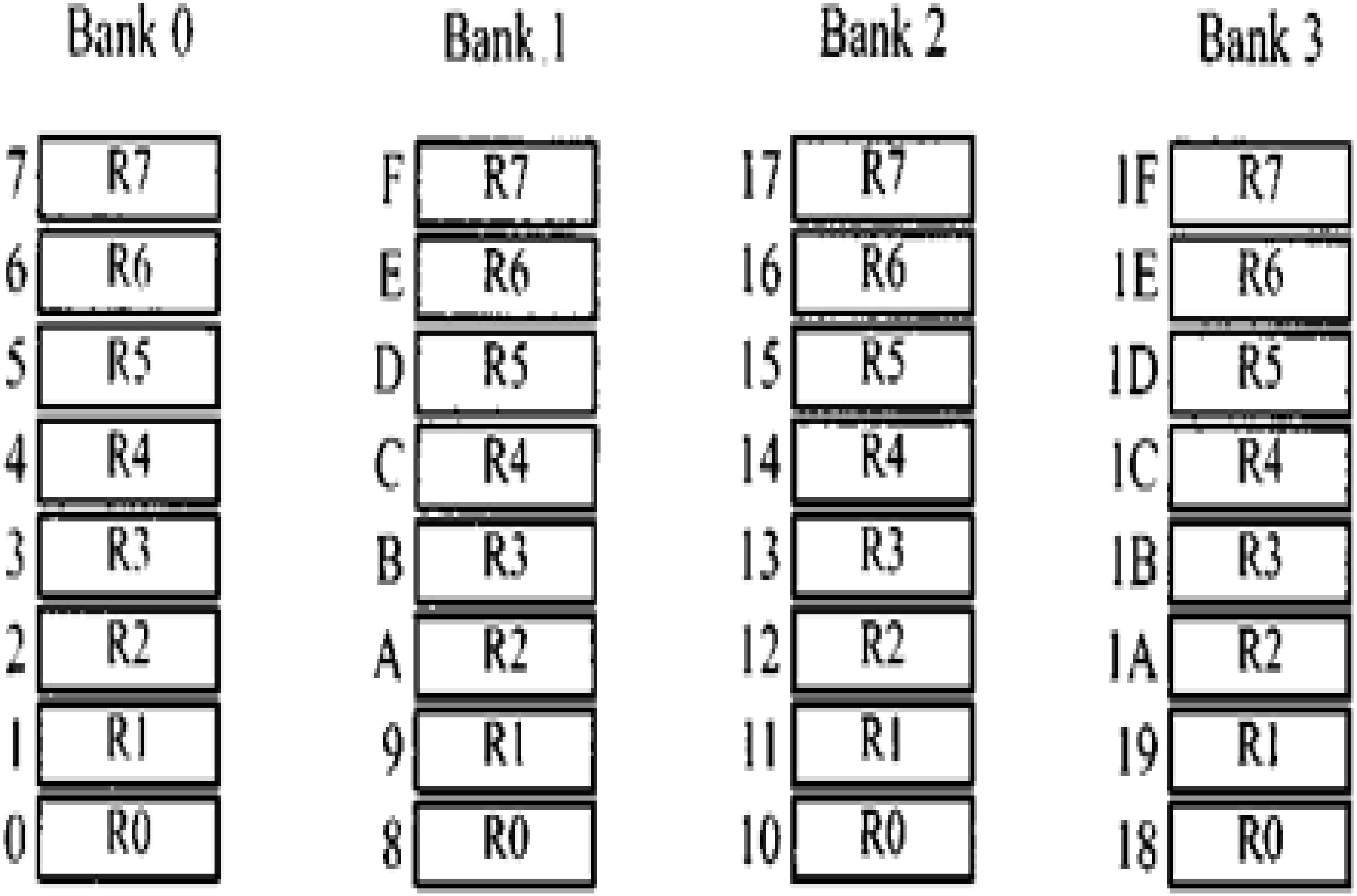
17
Bank 2
10

0F
Bank 1
08

07
Default register bank for R0-R7
00

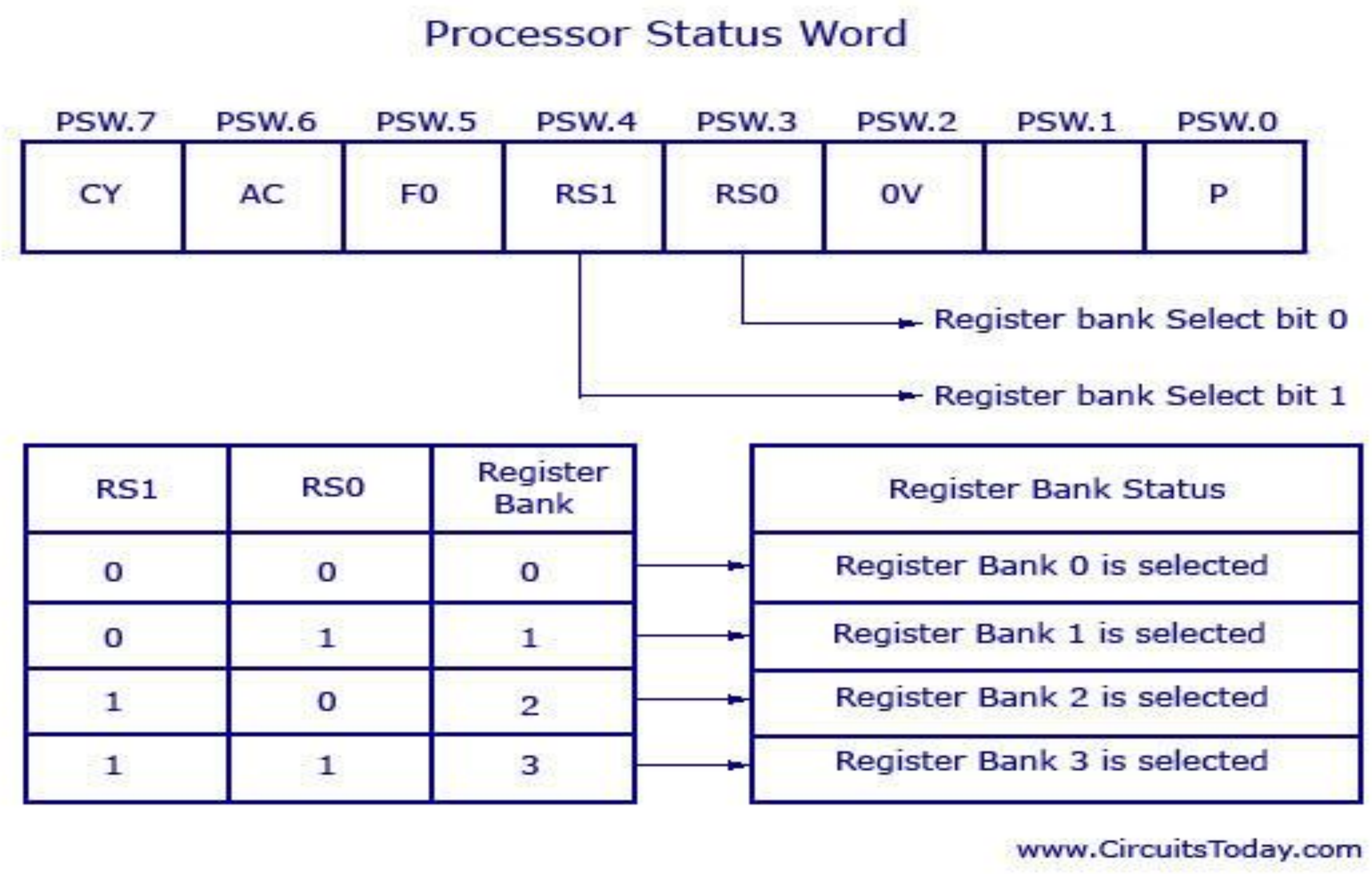
RAM

REGISTER BANKS



	Bank 0		Bank 1		Bank 2		Bank 3
7	R7	F	R7	17	R7	1F	R7
6	R6	E	R6	16	R6	1E	R6
5	R5	D	R5	15	R5	1D	R5
4	R4	C	R4	14	R4	1C	R4
3	R3	B	R3	13	R3	1B	R3
2	R2	A	R2	12	R2	1A	R2
1	R1	9	R1	11	R1	19	R1
0	R0	8	R0	10	R0	18	R0

PROGRAM STATUS WORD OF 8051



Processor Status Word

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
CY	AC	F0	RS1	RS0	OV		P

Register bank Select bit 0

Register bank Select bit 1

RS1	RS0	Register Bank
0	0	0
0	1	1
1	0	2
1	1	3

Register Bank Status
Register Bank 0 is selected
Register Bank 1 is selected
Register Bank 2 is selected
Register Bank 3 is selected

www.CircuitsToday.com

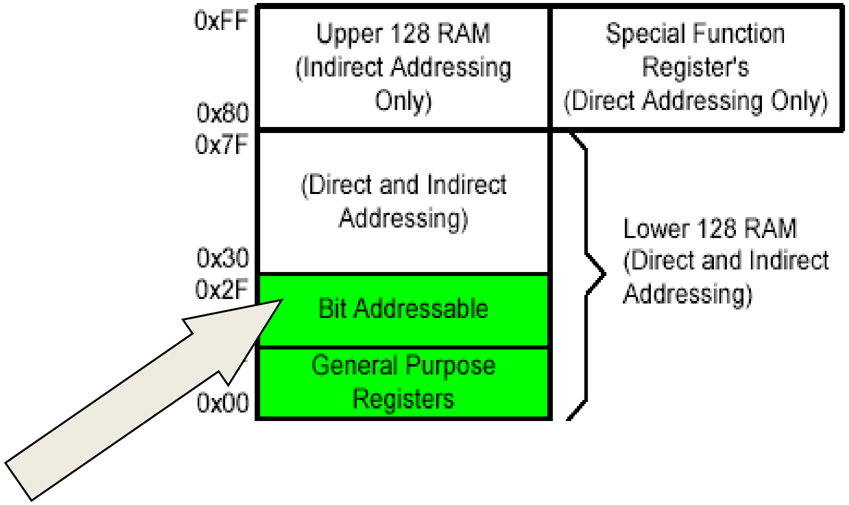
BIT ADDRESSABLE MEMORY

2F	7F							78
2E								
2D								
2C								
2B								
2A								
29								
28								
27								
26								
25								
24								
23						1A		
22								10
21	0F							08
20	07	06	05	04	03	02	01	00

20h – 2Fh (16 locations X 8-bits = 128 bits)

Bit addressing:

mov C, 1Ah
or
mov C, 23h.2



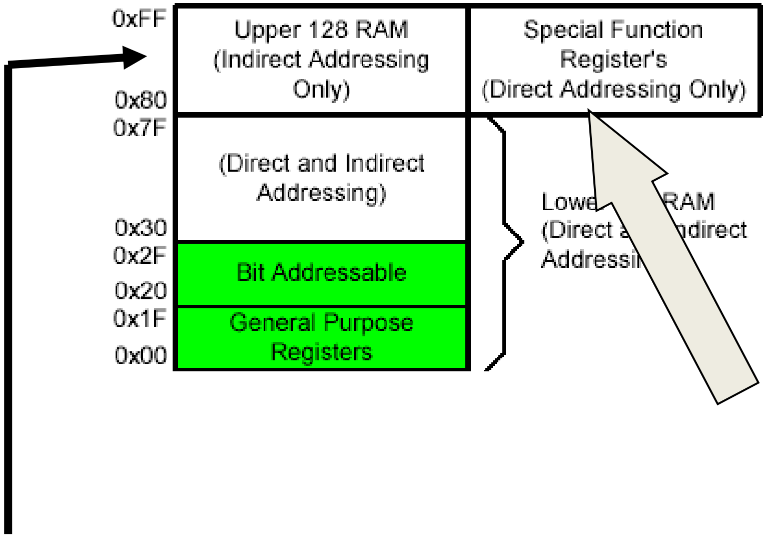
0xFF
0x80
0x7F
0x30
0x2F
0x00
Upper 128 RAM (Indirect Addressing Only)
Special Function Register's (Direct Addressing Only)
(Direct and Indirect Addressing)
Bit Addressable
General Purpose Registers
Lower 128 RAM (Direct and Indirect Addressing)

SPECIAL FUNCTION REGISTERS

DATA registers

CONTROL registers

- Timers
- Serial ports
- Interrupt system
- Analog to Digital converter
- Digital to Analog converter
- Etc.

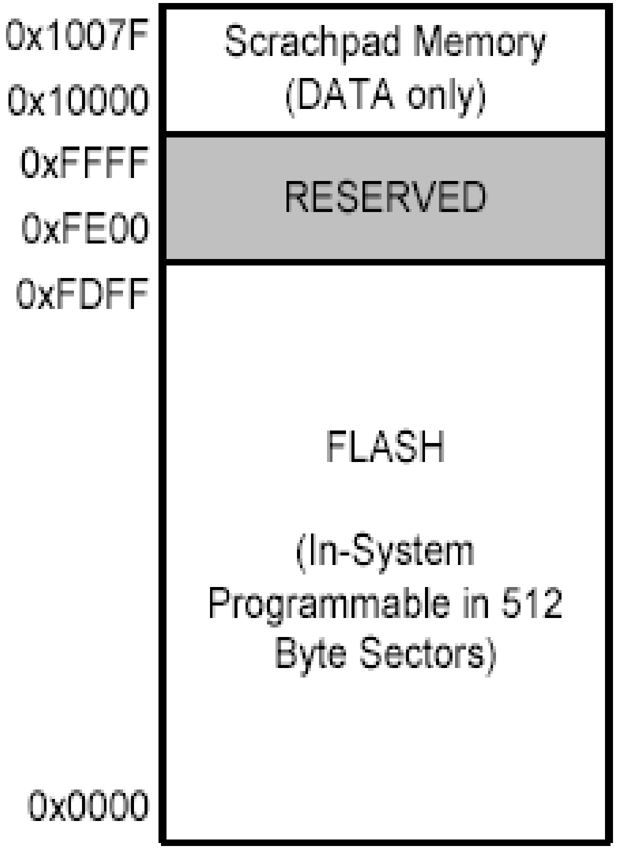

0xFF
0x80
0x7F
0x30
0x2F
0x20
0x1F
0x00
Upper 128 RAM (Indirect Addressing Only)
Special Function Register's (Direct Addressing Only)
(Direct and Indirect Addressing)
Bit Addressable
General Purpose Registers
Lowe RAM (Direct a ndirect Addressi

Addresses 80h – FFh

Direct Addressing used to access SPRs

ON-CHIP MEMORY: PROGRAM/DATA

PROGRAM/DATA MEMORY (FLASH)



0x1007F
0x10000
Scratchpad Memory (DATA only)

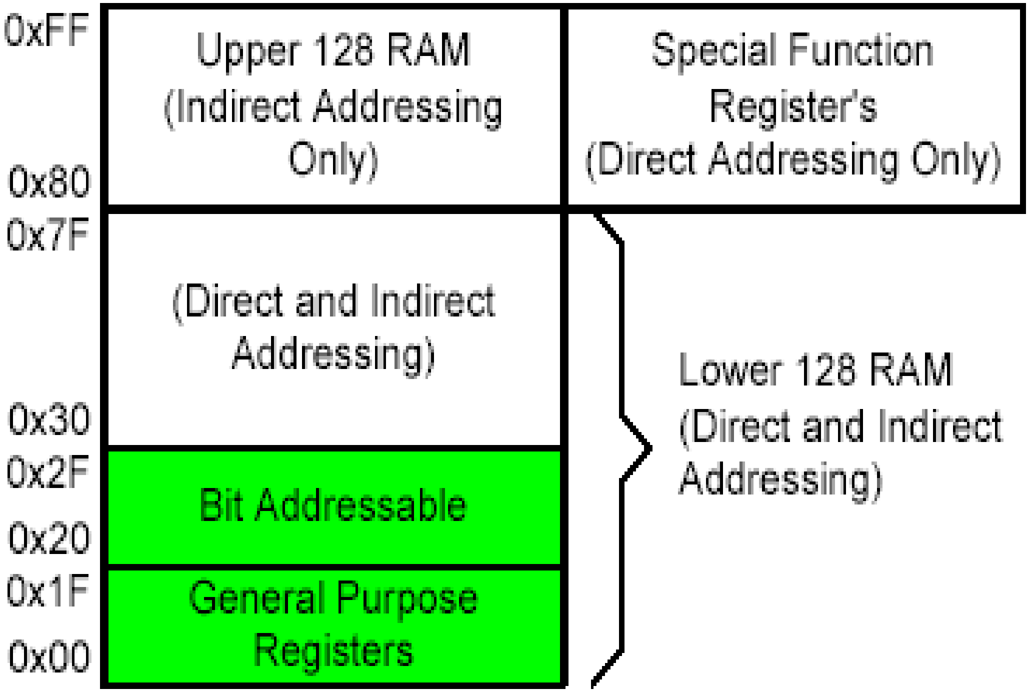
0xFFFF
0xFE00
RESERVED

0xFDFF
0x0000
FLASH

(In-System Programmable in 512 Byte Sectors)

DATA MEMORY (RAM)

INTERNAL DATA ADDRESS SPACE



0xFF
0x80
Upper 128 RAM (Indirect Addressing Only)

Special Function Register's (Direct Addressing Only)

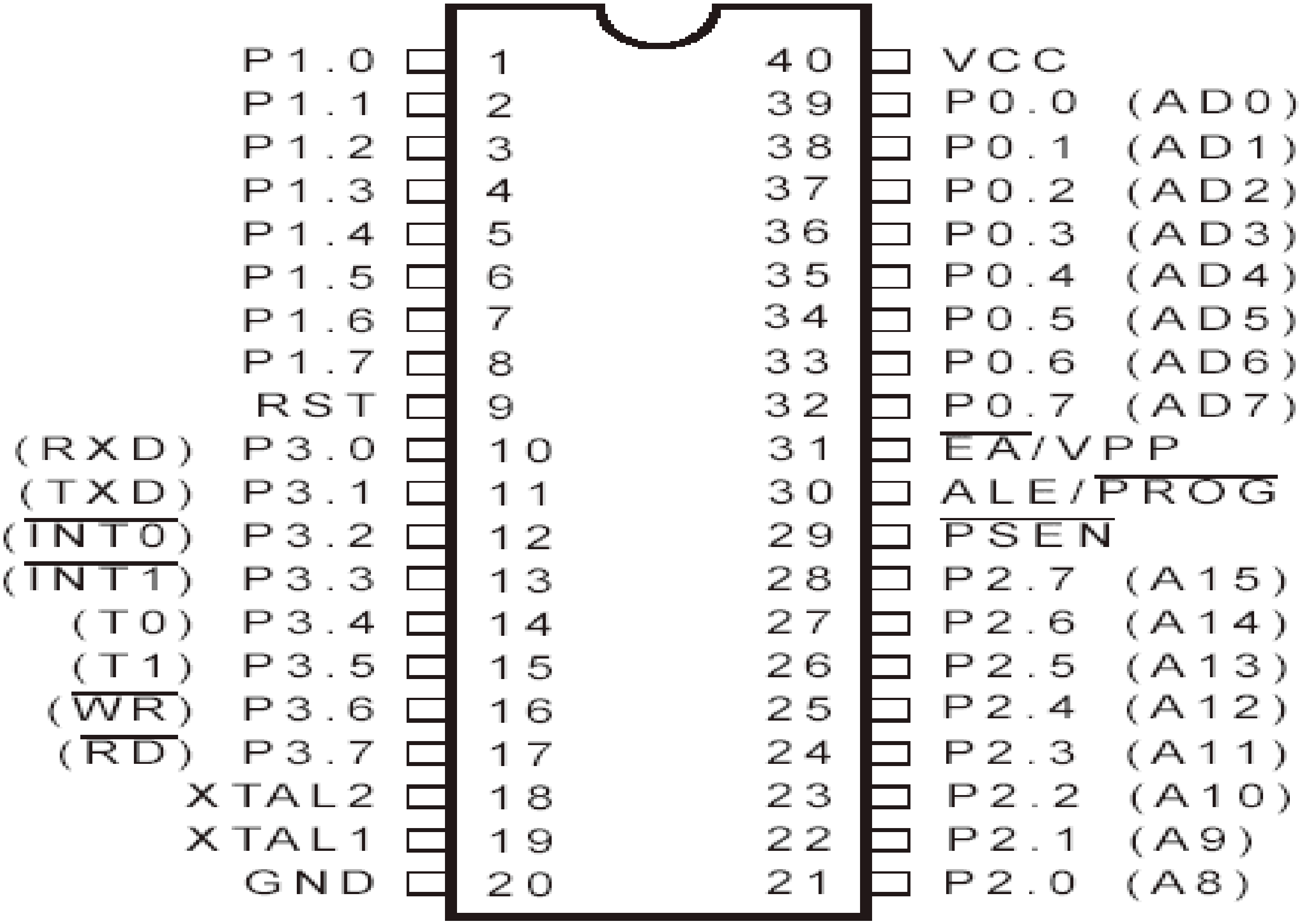
0x7F
0x30
(Direct and Indirect Addressing)

0x2F
0x20
Bit Addressable

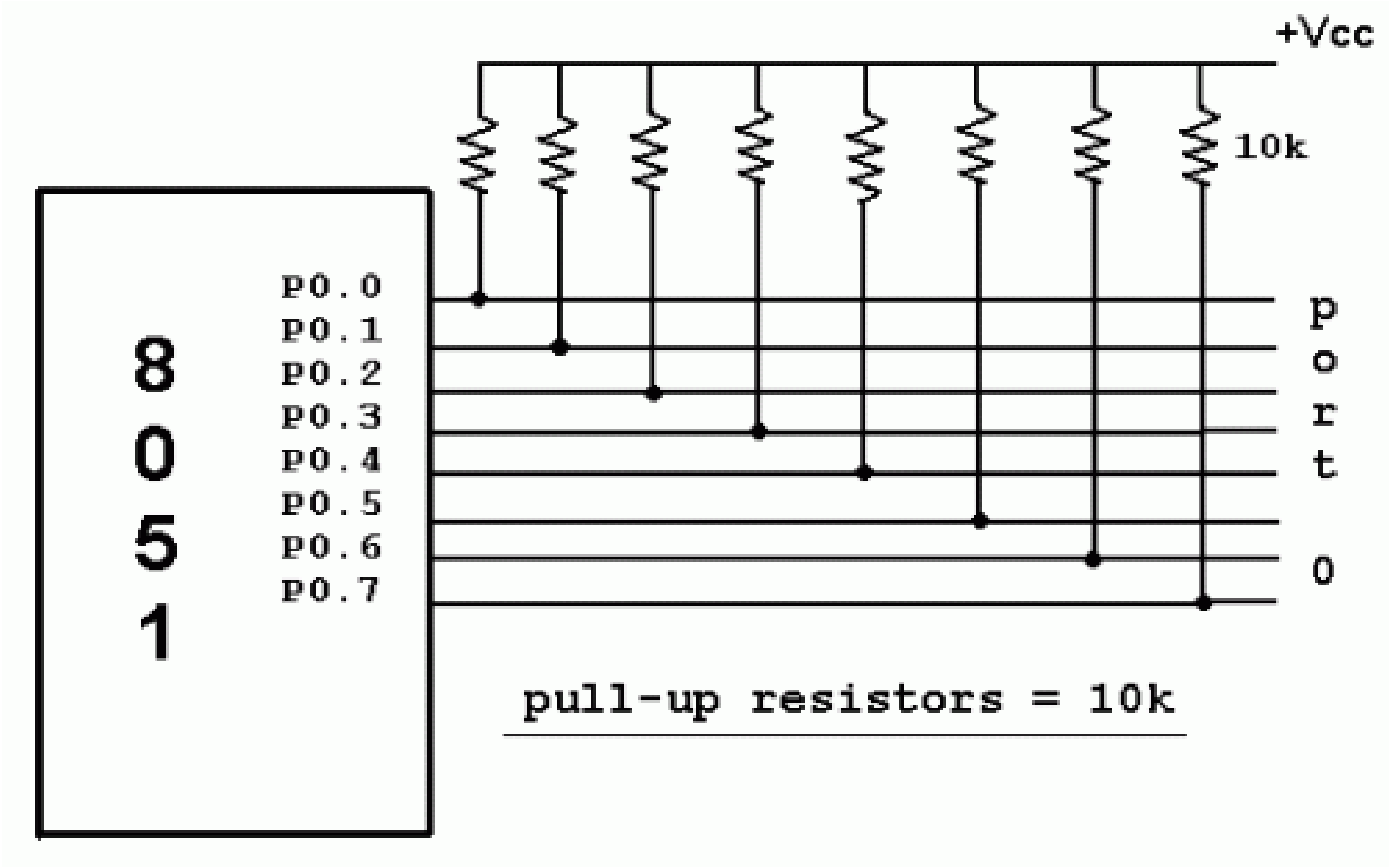
0x1F
0x00
General Purpose Registers

Lower 128 RAM (Direct and Indirect Addressing)

PIN DIAGRAM OF 8051



PORT STRUCTURE

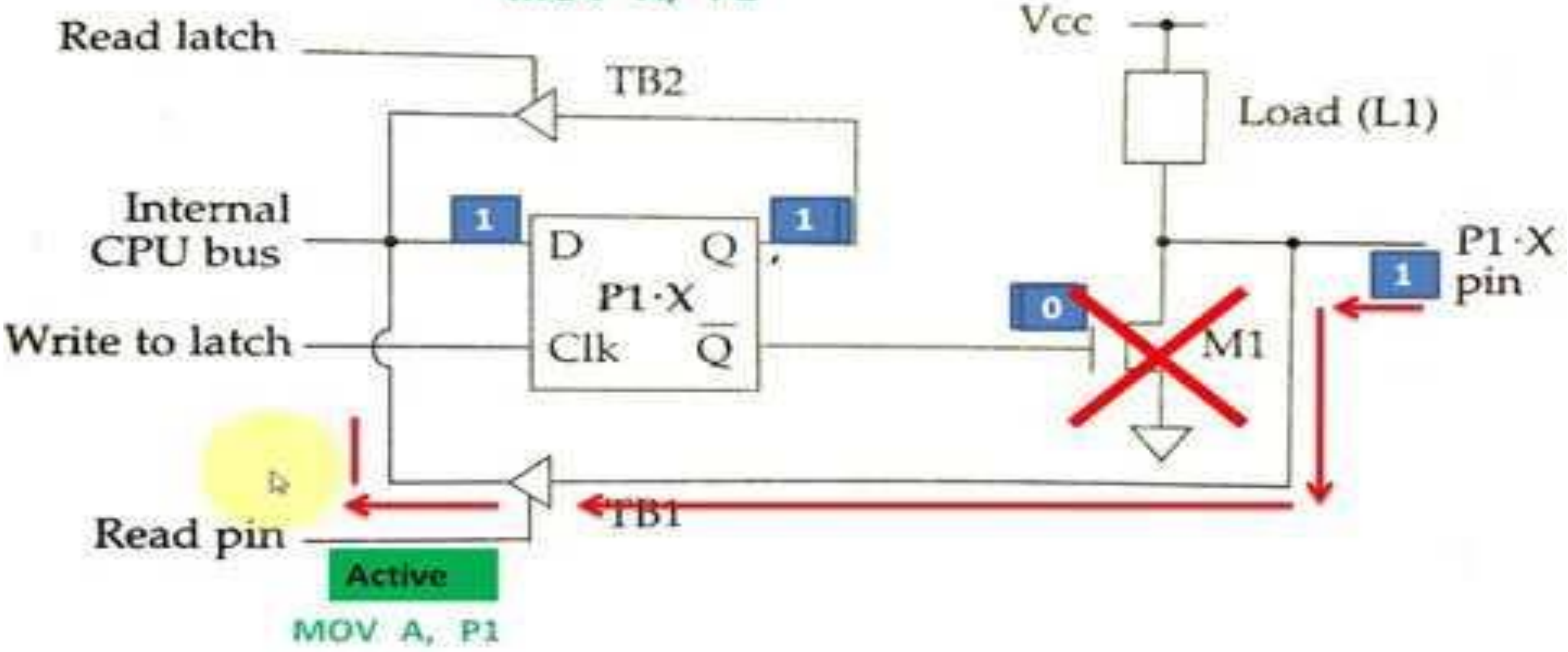


PORT STRUCTURE

How to read port pin?

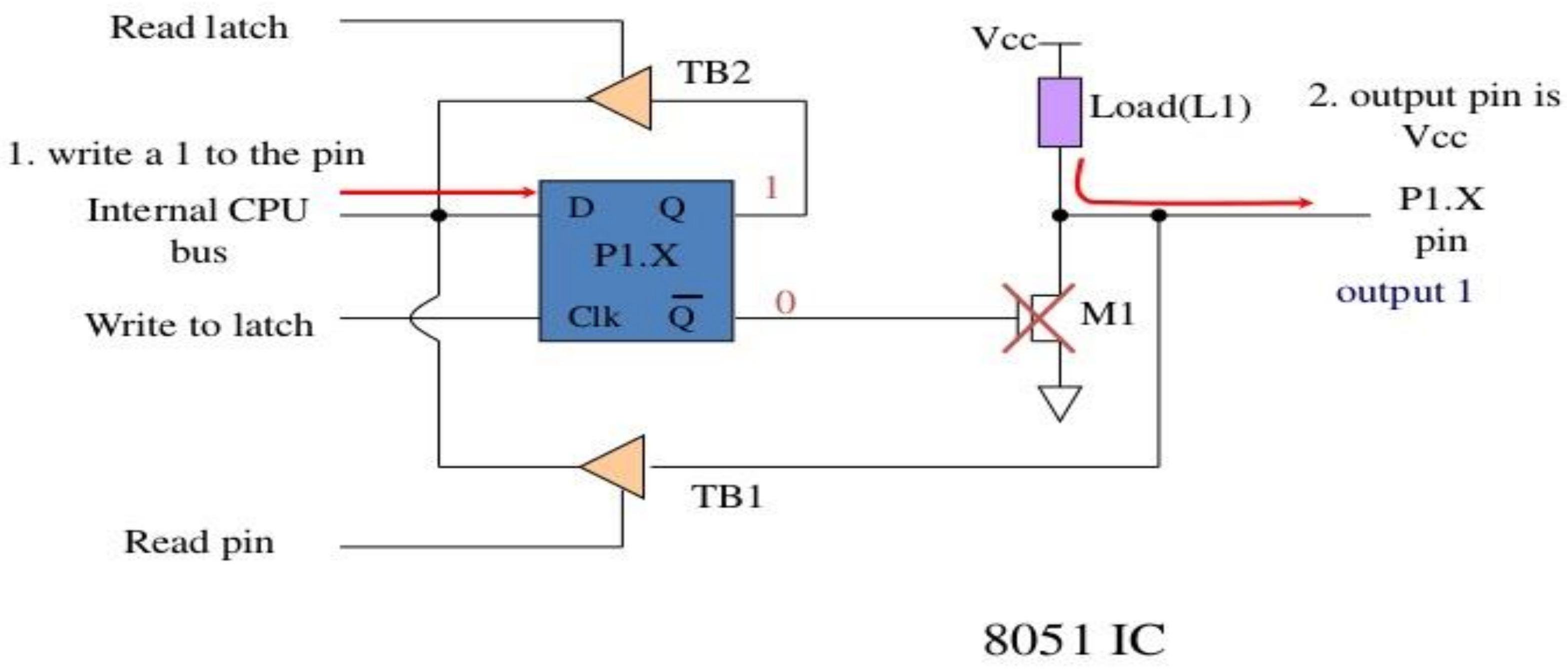
MOV P1, #0FFH; to configured port as input port, we must write 1(logic high) to that ports

MOV A, P1



PORT STRUCTURE

Writing “1” to Output Pin P1.X



ADDRESSING MODES OF 8051

1)Immediate addressing mode

```
MOVA, #0AFH;
MOVR3, #45H;
MOVDPTR, #FE00H;
```

2)Register addressing mode

```
MOVA, R5;
MOVR2, #45H;
MOVR0, A;
```

3) Direct addressing mod

```
MOV80H, R6;
MOVR2, 45H;
MOVR0, 05H;
```

4) Indirect addressing mode

```
MOV0E5H, @R0;
MOV@R1, 80H;
MOVXA, @R1;
MOV@DPTR, A;
```

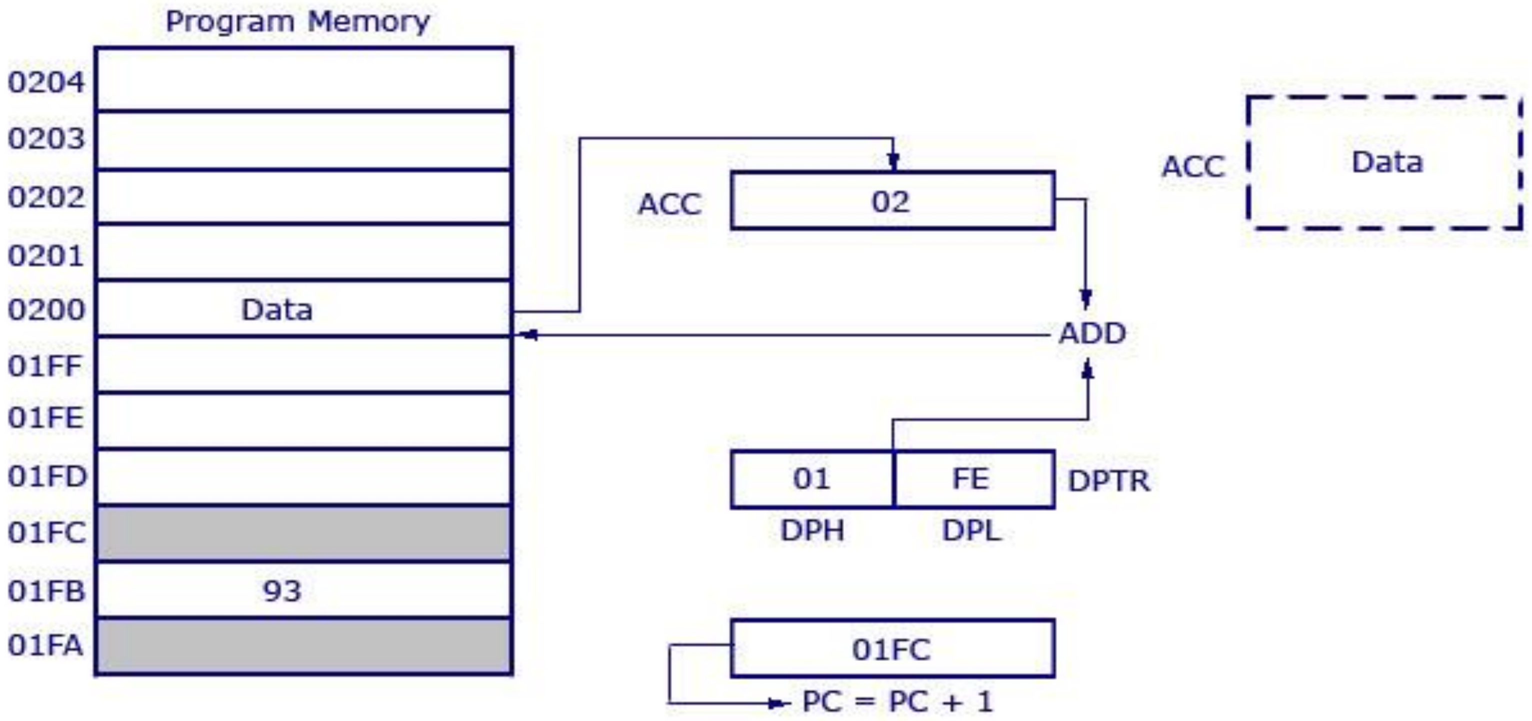
5) Indexed addressing mode

MOVCA, @A+PC;

MOVCA, @A+DPTR;

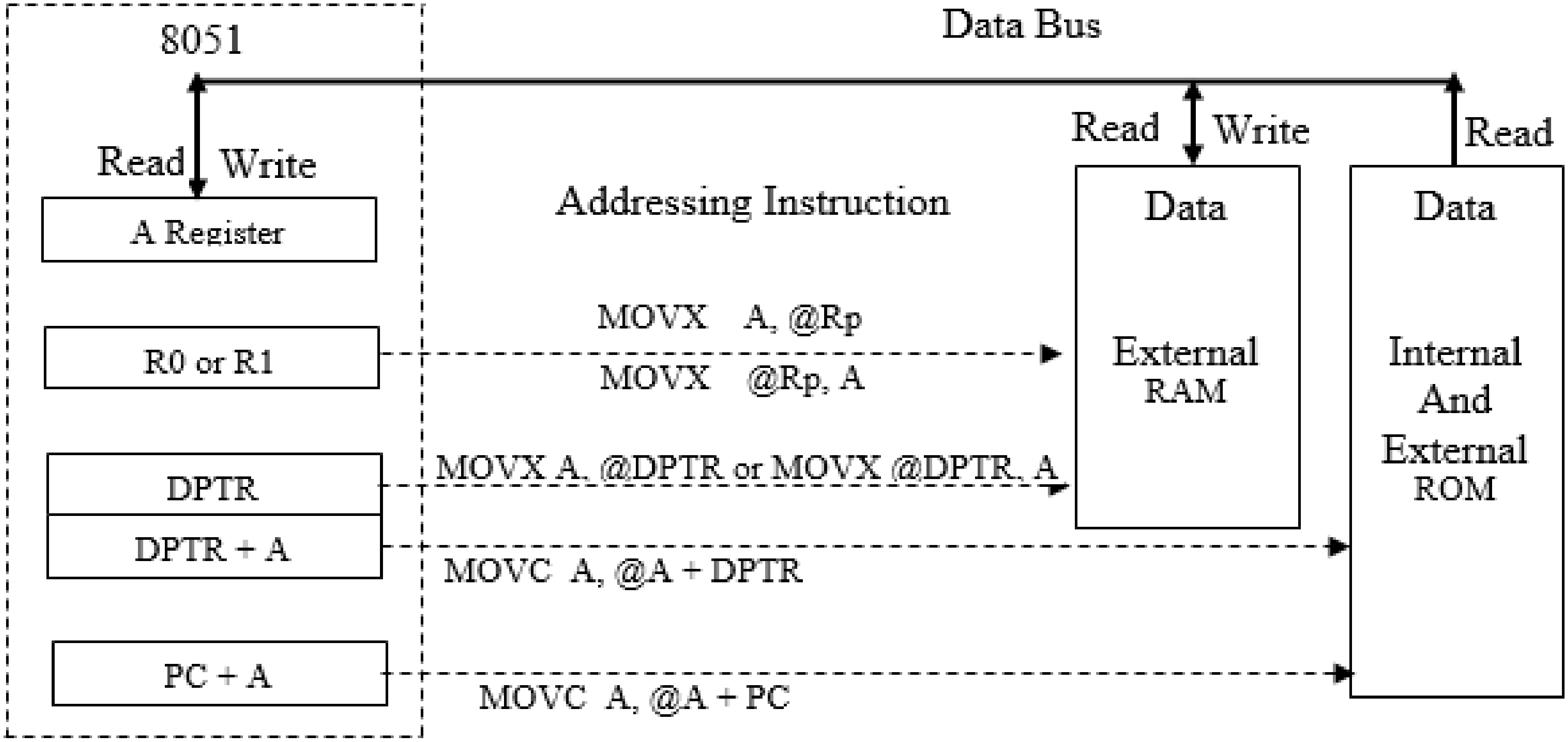
Indexed Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOVC A,@A +DPTR	93H	1	2

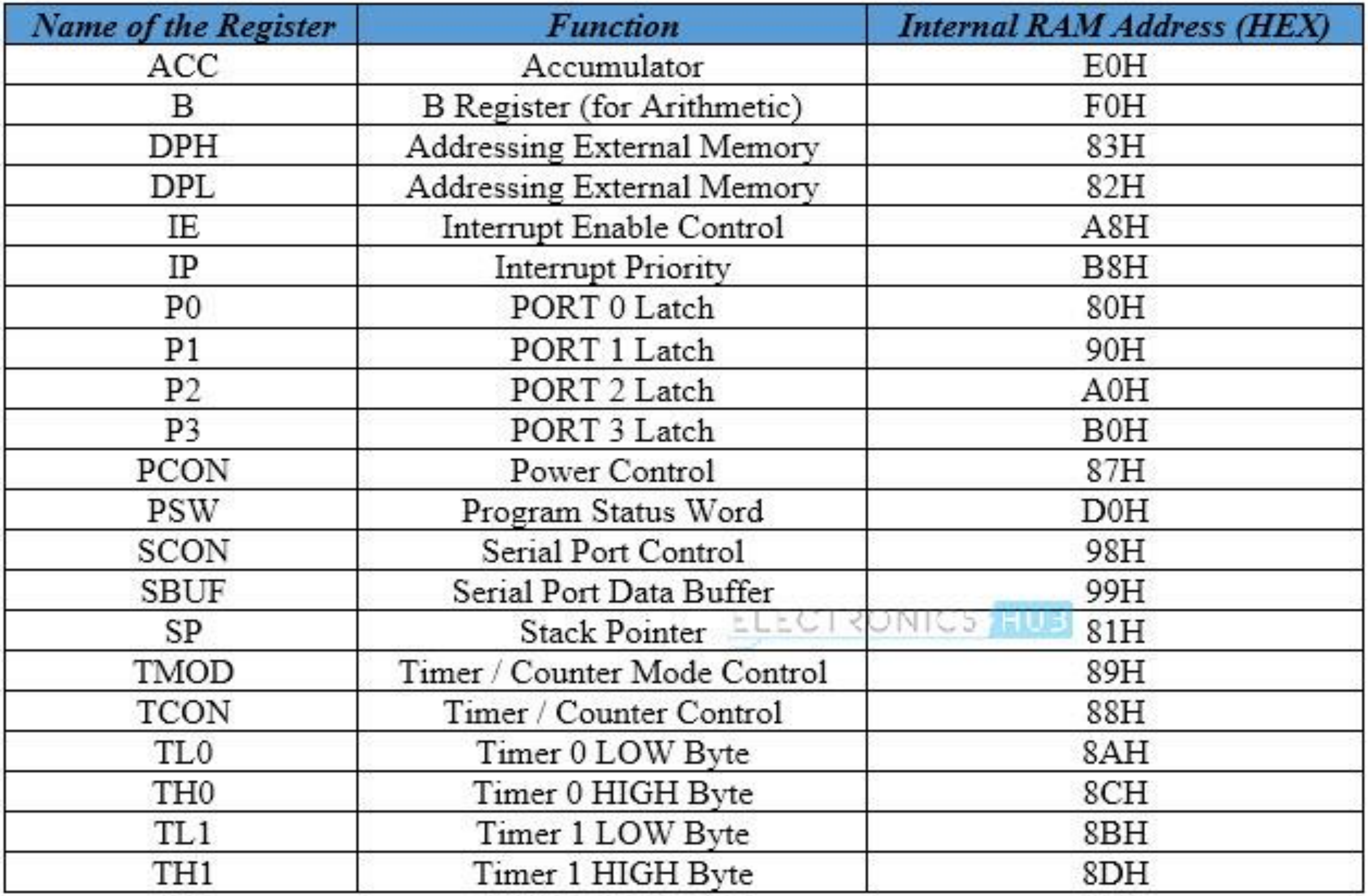


INSTRUCTION SET OF 8051

DATA TRANSFER	***ARITHMETIC***	***LOGICAL***	***BOOLEAN***	***PROGRAM BRANCHING***
MOV	ADD	ANL	CLR	LJMP
MOVC	ADDC	ORL	SETB	AJMP
MOVX	SUBB	XRL	MOV	SJMP
PUSH	INC	CLR	JC	JZ
POP	DEC	CPL	JNC	JNZ
XCH	MUL	RL	JB	CJNE
XCHD	DIV	RLC	JNB	DJNZ
	DA A	RR	JBC	NOP
		RRC	ANL	LCALL
		SWAP	ORL	ACALL
			CPL	RET
				RETI
				JMP


8051
Data Bus
Read
Write
A Register
Addressing Instruction
Read
Write
Read
Data
External
RAM
Data
Internal
And
External
ROM
R0 or R1
MOVX A, @Rp
MOVX @Rp, A
DPTR
MOVX A, @DPTR or MOVX @DPTR, A
DPTR + A
MOVC A, @A + DPTR
PC + A
MOVC A, @A + PC

SPECIAL FUNCTION REGISTERS OF 8051



Name of the Register	*Function*	*Internal RAM Address (HEX)*
ACC	Accumulator	E0H
B	B Register (for Arithmetic)	F0H
DPH	Addressing External Memory	83H
DPL	Addressing External Memory	82H
IE	Interrupt Enable Control	A8H
IP	Interrupt Priority	B8H
P0	PORT 0 Latch	80H
P1	PORT 1 Latch	90H
P2	PORT 2 Latch	A0H
P3	PORT 3 Latch	B0H
PCON	Power Control	87H
PSW	Program Status Word	D0H
SCON	Serial Port Control	98H
SBUF	Serial Port Data Buffer	99H
SP	Stack Pointer	81H
TMOD	Timer / Counter Mode Control	89H
TCON	Timer / Counter Control	88H
TL0	Timer 0 LOW Byte	8AH
TH0	Timer 0 HIGH Byte	8CH
TL1	Timer 1 LOW Byte	8BH
TH1	Timer 1 HIGH Byte	8DH

8051 REAL TIME CONTROL

UNIT- V

A **timer** is a specialized type of clock which is used to measure time intervals. A timer that counts from zero upwards for measuring time elapsed is often called a **stopwatch**. It is a device that counts down from a specified time interval and used to generate a time delay, for example, an hourglass is a timer.
A **counter** is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal. It is used to count the events happening outside the microcontroller. In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.

DIFFERENCES BETWEEN TIMERS/COUNTERS

Timer	Counter
The register incremented for every machine cycle.	The register is incremented considering 1 to 0 transition at its corresponding to an external input pin (T0, T1).
Maximum count rate is 1/12 of the oscillator frequency.	Maximum count rate is 1/24 of the oscillator frequency.
A timer uses the frequency of the internal clock, and generates delay.	A counter uses an external signal to count pulses.

TMOD : Timer/Counter Mode Control Register (Not Bit Addressable)

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
TIMER 1				TIMER 0			

GATE When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).

C/$\overline{T}$ Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).

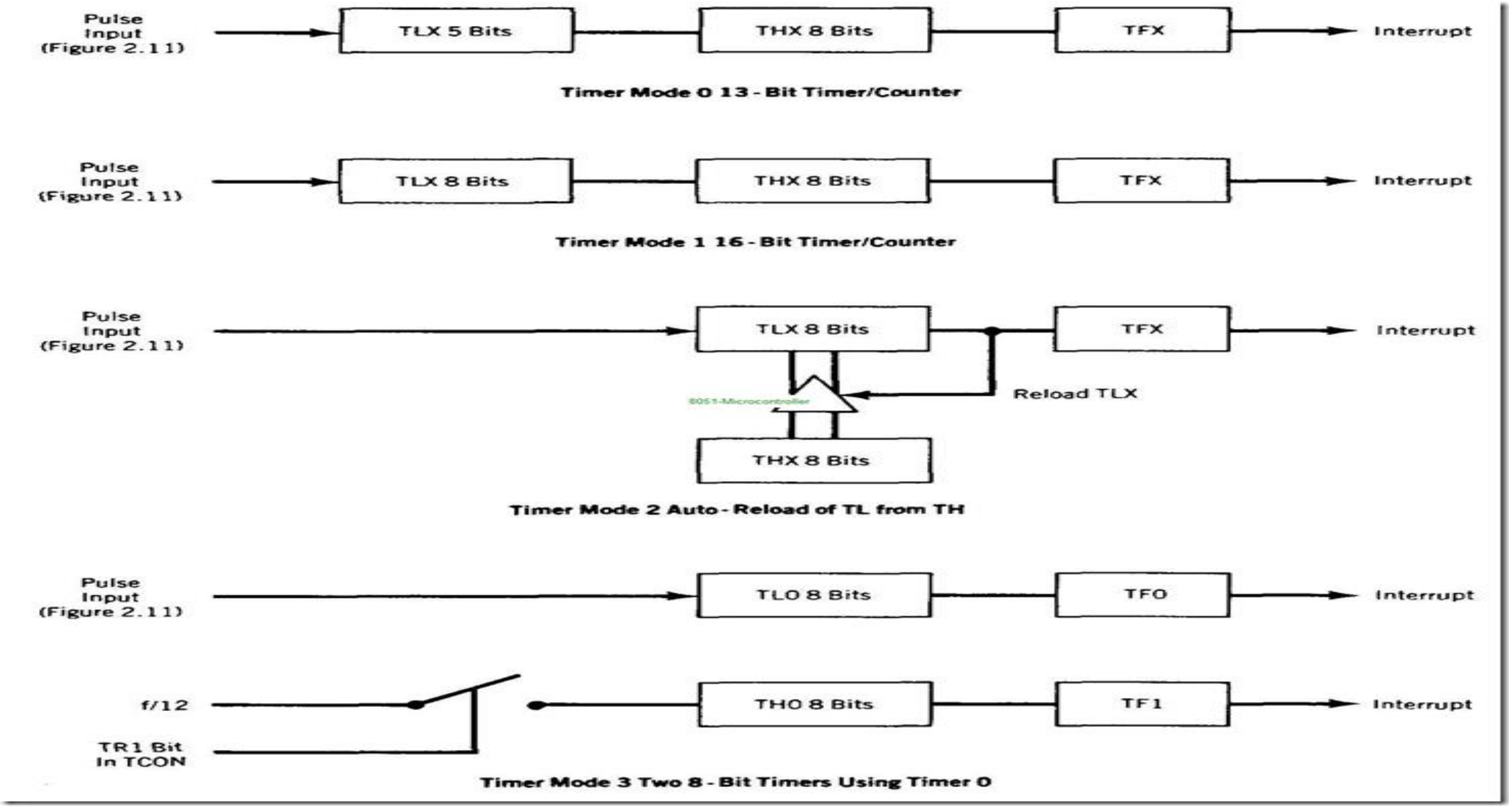
M1 Mode selector bit (NOTE 1).

M0 Mode selector bit (NOTE 1).

Note 1 :

M1	**M0**	**OPERATING MODE**	
0	0	0	13–bit Timer
0	1	1	16–bit Timer/Counter
1	0	2	8–bit Auto–Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8–bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8–bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

MODES OF TIMERS/COUNTERS


Timer Mode 0 13 - Bit Timer/Counter

Timer Mode 1 16 - Bit Timer/Counter

Timer Mode 2 Auto - Reload of TL from TH

Timer Mode 3 Two 8 - Bit Timers Using Timer 0

TCON : Timer/Counter Control Register (Bit Addressable)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1 TCON.7 Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.

TR1 TCON.6 Timer 1 run control bit. Set/cleared by software to turn Timer/Counter ON/OFF.

TF0 TCON.5 Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the service routine.

TR0 TCON.4 Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.

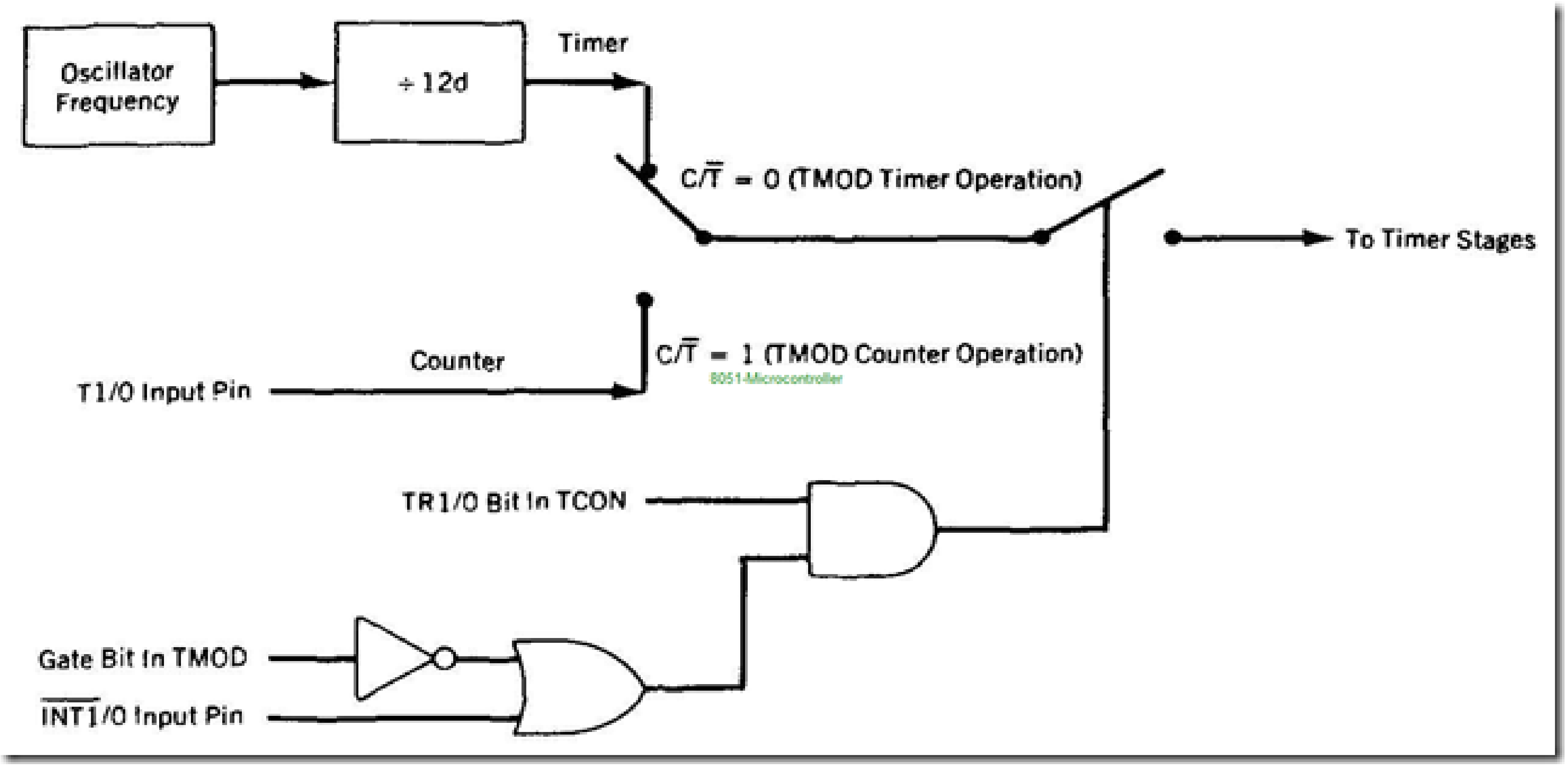
IE1 TCON.3 External Interrupt 1 edge flag. Set by hardware when External interrupt edge is detected. Cleared by hardware when interrupt is processed.

IT1 TCON.2 Interrupt 1 type control bit. Set/cleared by software to specify falling edge/flow level triggered External Interrupt.

IE0 TCON.1 External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.

IT0 TCON.0 Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

INTERNAL STRUCTURE OF TIMER/COUNTER


Oscillator Frequency
÷ 12d
Timer
C/$\overline{T}$ = 0 (TMOD Timer Operation)
To Timer Stages
Counter
T1/0 Input Pin
C/$\overline{T}$ = 1 (TMOD Counter Operation)
8051-Microcontroller
TR1/0 Bit In TCON
Gate Bit In TMOD
$\overline{\text{INT1}}$/0 Input Pin

INTERRUPT STRUCTURE OF 8051

External interrupts

1)INT0

2)INT1

Internal interrupts

1)Timer0 (TF0)

2)Timer1(TF1)

Serial communication interrupts

Transmit interrupt (TI)

Receive interrupt (RI)

IE &IP Registers are used to enable and disable these interrupts.

IE Register – Interrupt Enable Register

7	6	5	4	3	2	1	0
EA	X	X	ES	ET1	EX1	ET0	EX0

- EX0- Enable/Disable – External Interrupt 0
- ET0- Enable/Disable – Timer 0 overflow Interrupt
- EX1- Enable/Disable – External Interrupt 1
- ET1- Enable/Disable – Timer 1 overflow Interrupt
- ES -Enable/Disable – Serial port Interrupt
- EA - Enable/Disable – All interrupts

Examples –

-Disable all interrupts

CLR IE.7

Interrupt Priority Register

	7	6	5	4	3	2	1	0
IP -	X	X	X	PS	PT1	PX1	PT0	PX0

PX0 – Priority – External Interrupt 0

PT0 – Priority – Timer 0 overflow

PX1 – Priority – External Interrupt 1

PT1 – Priority – Timer 1 overflow

PS - Priority – Serial port Interrupt

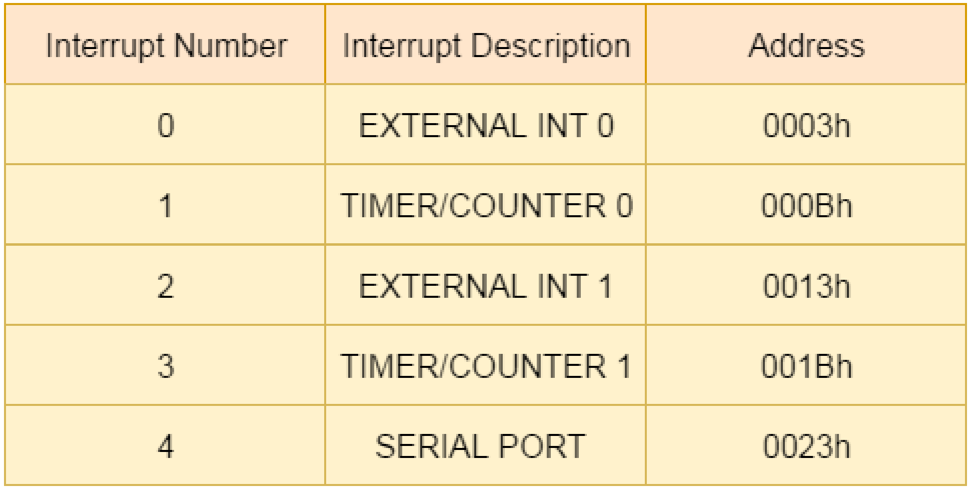
Example – To place Timer 0, external interrupt 1 at high priority and other interrupts at low priority.

IP = 0000 0110 = 06H
MOV IP, # 06H

or

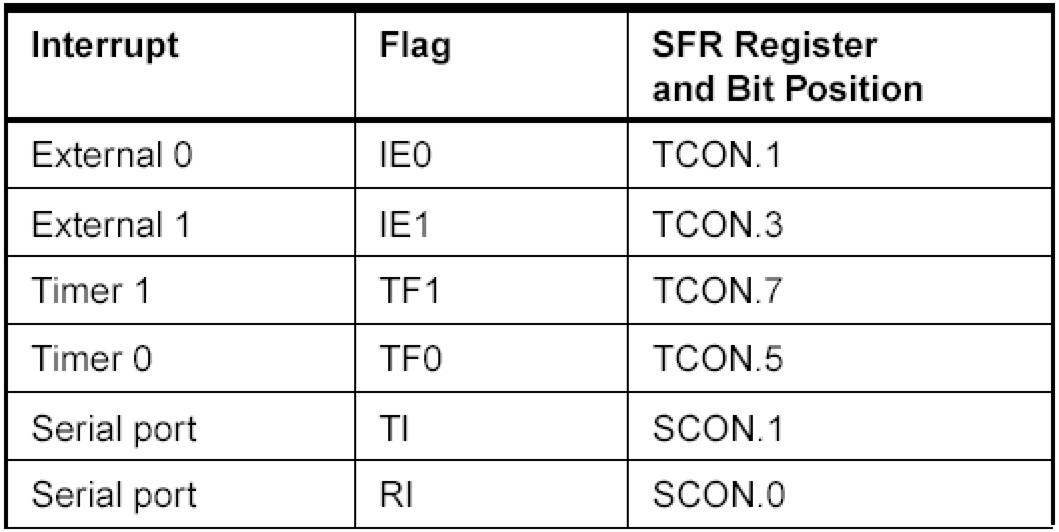
SETB IP.1
SETB IP.2

INTERRUPT PRIORITIES AND VECTOR ADDRESSES



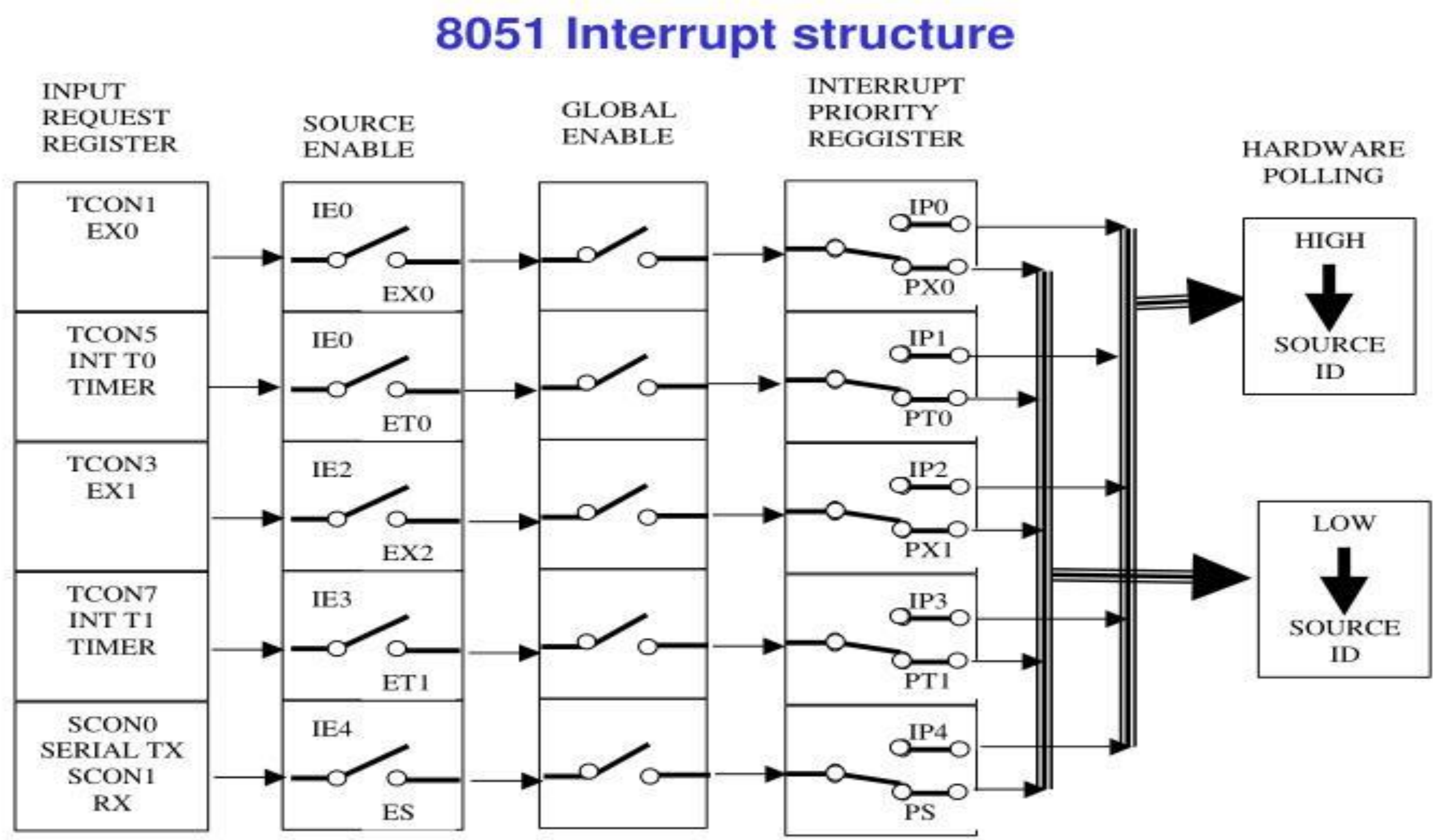
Interrupt Number	Interrupt Description	Address
0	EXTERNAL INT 0	0003h
1	TIMER/COUNTER 0	000Bh
2	EXTERNAL INT 1	0013h
3	TIMER/COUNTER 1	001Bh
4	SERIAL PORT	0023h

Interrupt Flag Bits

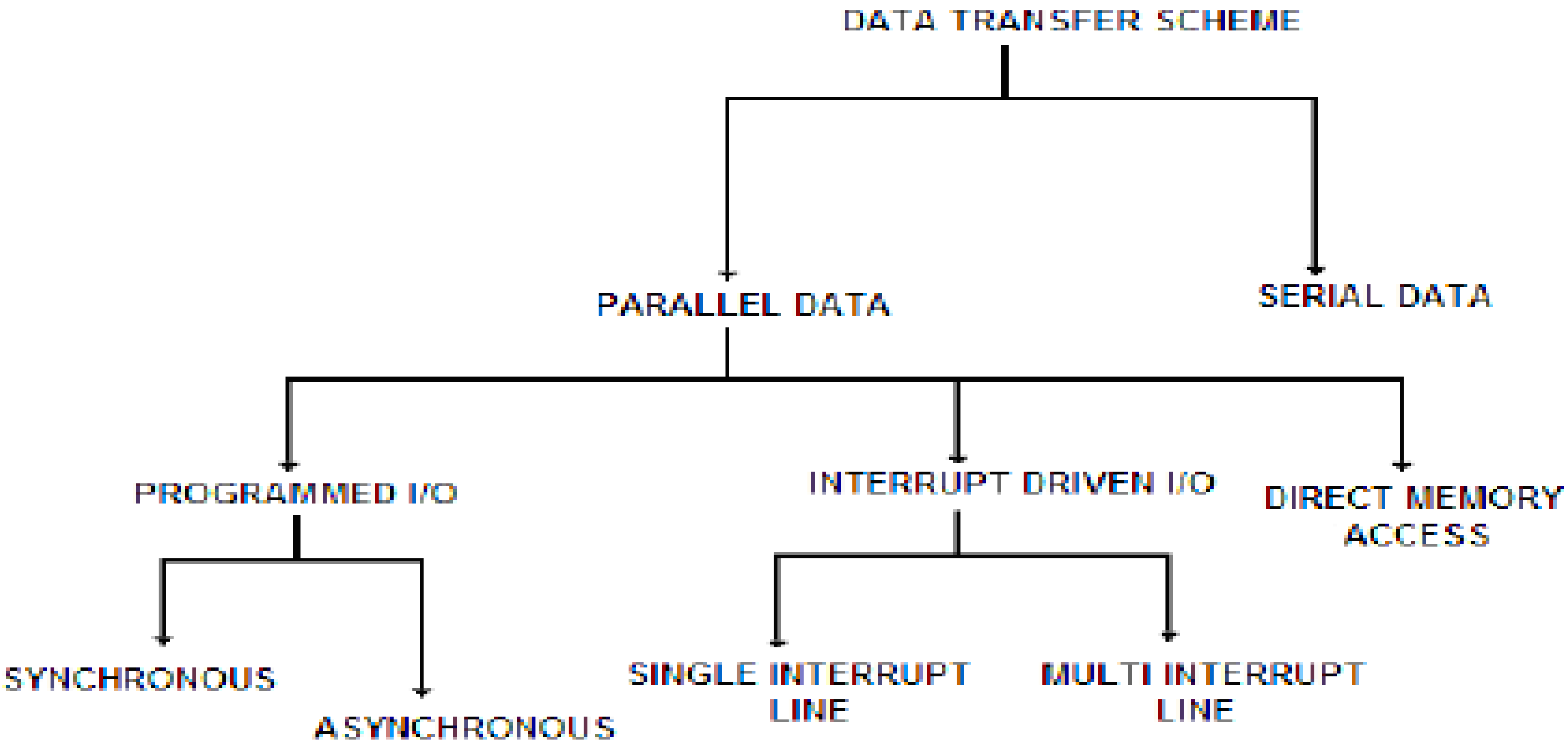


Interrupt	Flag	SFR Register and Bit Position
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 1	TF1	TCON.7
Timer 0	TF0	TCON.5
Serial port	TI	SCON.1
Serial port	RI	SCON.0

8051 INTERRUPT STRUCTURE



MODES OF DATA TRANSFER SCHEMES


DATA TRANSFER SCHEME

PARALLEL DATA

SERIAL DATA

PROGRAMMED I/O

INTERRUPT DRIVEN I/O

DIRECT MEMORY ACCESS

SYNCHRONOUS

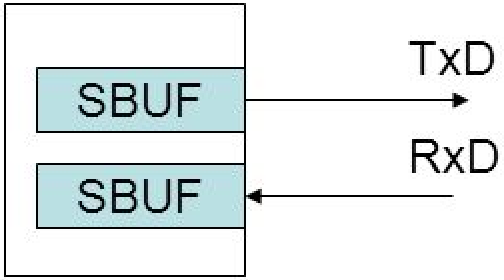
ASYNCHRONOUS

SINGLE INTERRUPT LINE

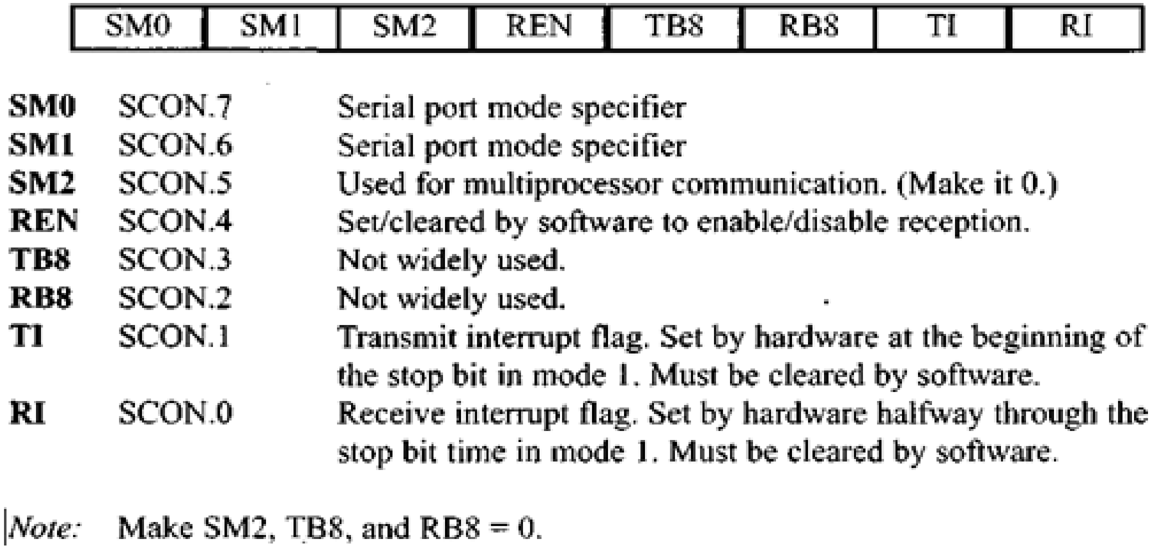
MULTI INTERRUPT LINE

SBUF (Serial Control, Addresses 99h):

- used to send and receive data via the on-board serial port.
- These are two physical registers – one as Write only and the other is Read only
- When SBUF is written with data Transmission starts
- To receive a data byte, 8051 has to be enabled explicitly for "Receive" operation



SCON REGISTER OF 8051



SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode specifier
SM1	SCON.6	Serial port mode specifier
SM2	SCON.5	Used for multiprocessor communication. (Make it 0.)
REN	SCON.4	Set/cleared by software to enable/disable reception.
TB8	SCON.3	Not widely used.
RB8	SCON.2	Not widely used.
TI	SCON.1	Transmit interrupt flag. Set by hardware at the beginning of the stop bit in mode 1. Must be cleared by software.
RI	SCON.0	Receive interrupt flag. Set by hardware halfway through the stop bit time in mode 1. Must be cleared by software.

Note: Make SM2, TB8, and RB8 = 0.

SM0	SM1	Operation	Description	Baud Rate Source
0	0	Mode 0	8-bit UART	1/12 the quartz frequency
0	1	Mode 1	8-bit UART	Determined by the timer 1
1	0	Mode 2	9-bit UART	1/32 the quartz frequency
1	1	Mode 0	9-bit UART	Determined by the timer 1

PCON Register

SMOD Double baud rate. If Timer 1 is used to generate baud and SMOD=1, the baud rate is doubled when the Serial Port is used in modes 1,2,3.

GF1,GF0 General purpose flag bit.

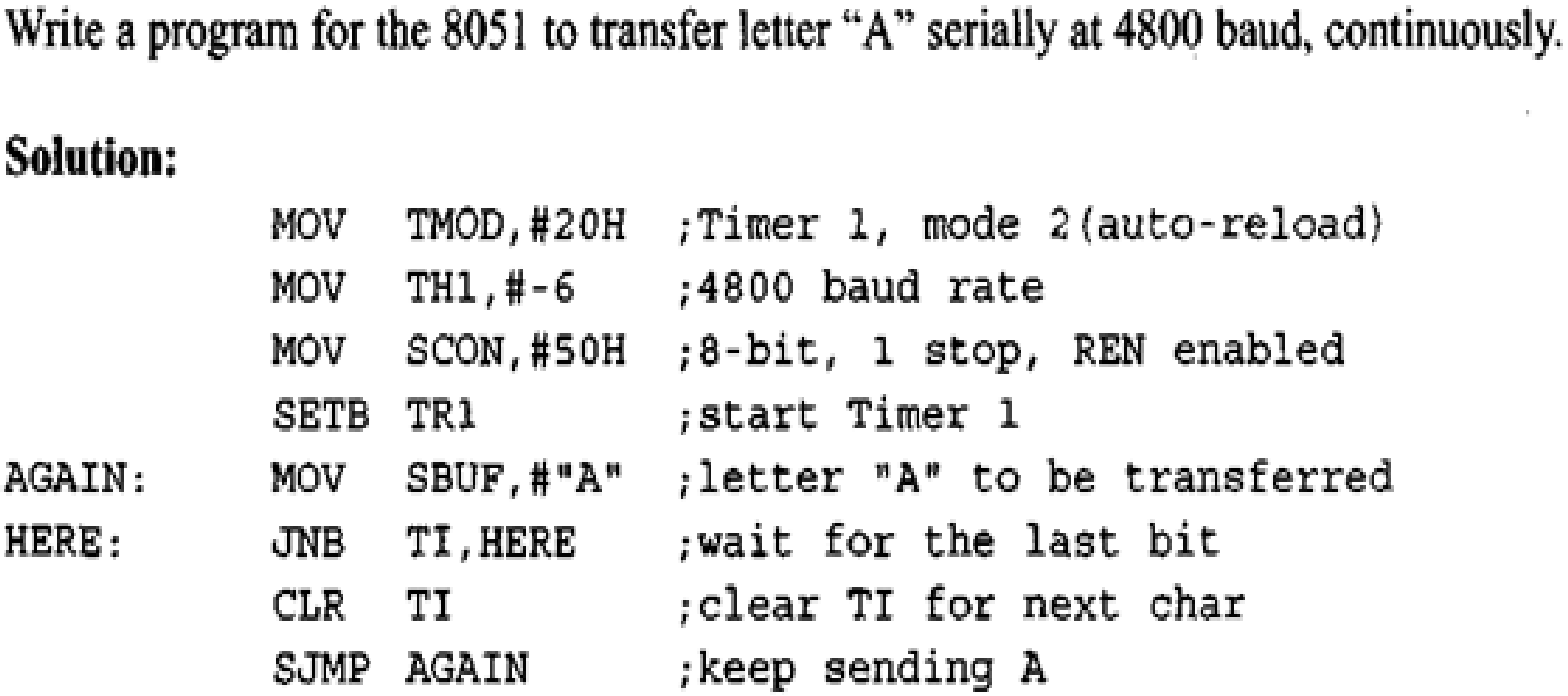
PD Power down bit. Setting this bit activates “Power Down” operation in the 80C51BH. (precedence)

IDL Idle Mode bit. Setting this bit activates “Idle Mode” operation in the 80C51BH.

(MSB)							(LSB)
SMOD	--	--	--	GF1	GF2	PD	IDL

* PCON is not bit-addressable. See Appendix H. p410

PROGRAMMING SERIEL COMMUNICATION INTERRUPTS

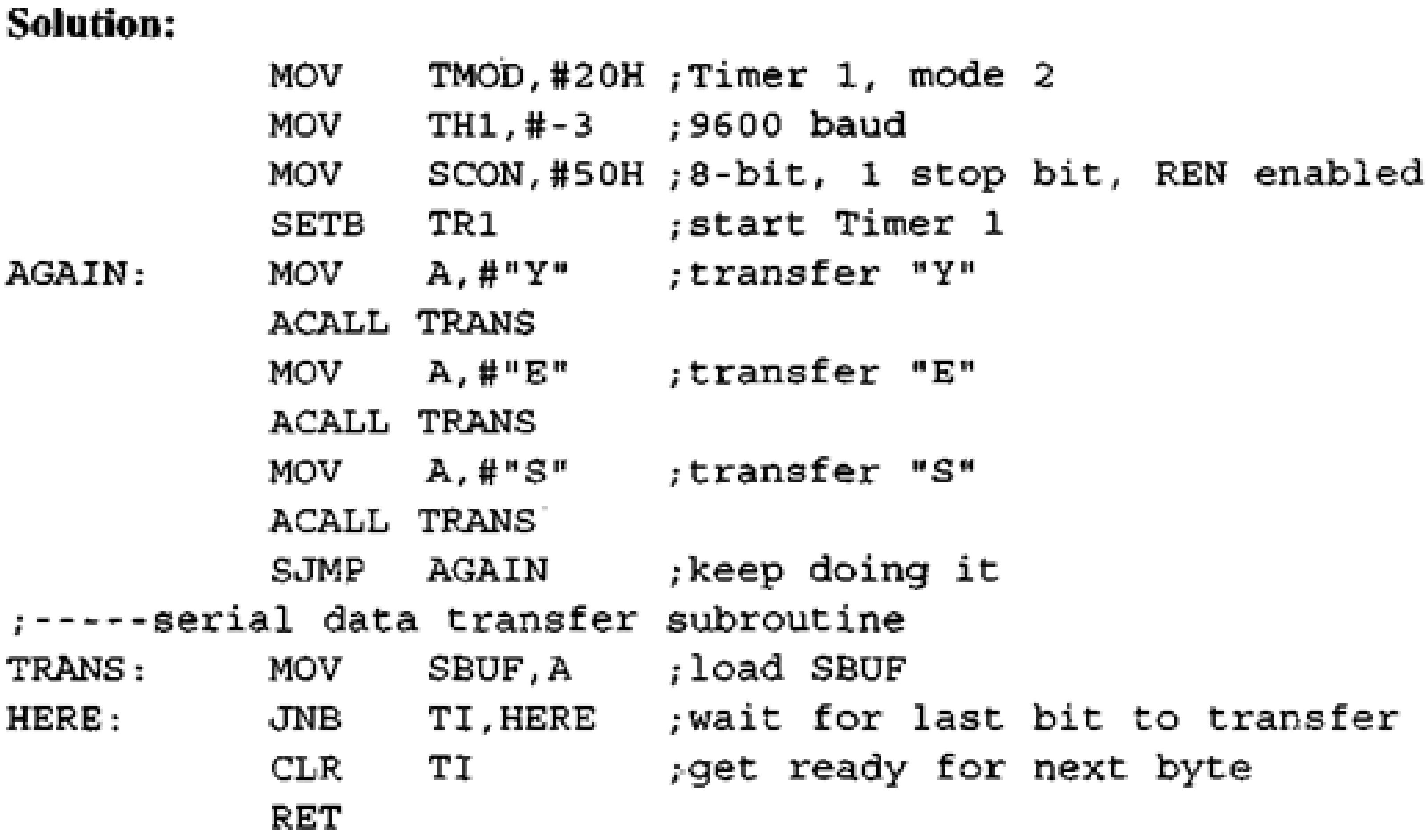


Write a program for the 8051 to transfer letter “A” serially at 4800 baud, continuously.

Solution:

```
        MOV   TMOD,#20H   ;Timer 1, mode 2(auto-reload)
        MOV   TH1,#-6     ;4800 baud rate
        MOV   SCON,#50H   ;8-bit, 1 stop, REN enabled
        SETB  TR1         ;start Timer 1
AGAIN:  MOV   SBUF,#"A"   ;letter "A" to be transferred
HERE:   JNB   TI,HERE     ;wait for the last bit
        CLR   TI          ;clear TI for next char
        SJMP  AGAIN       ;keep sending A
```

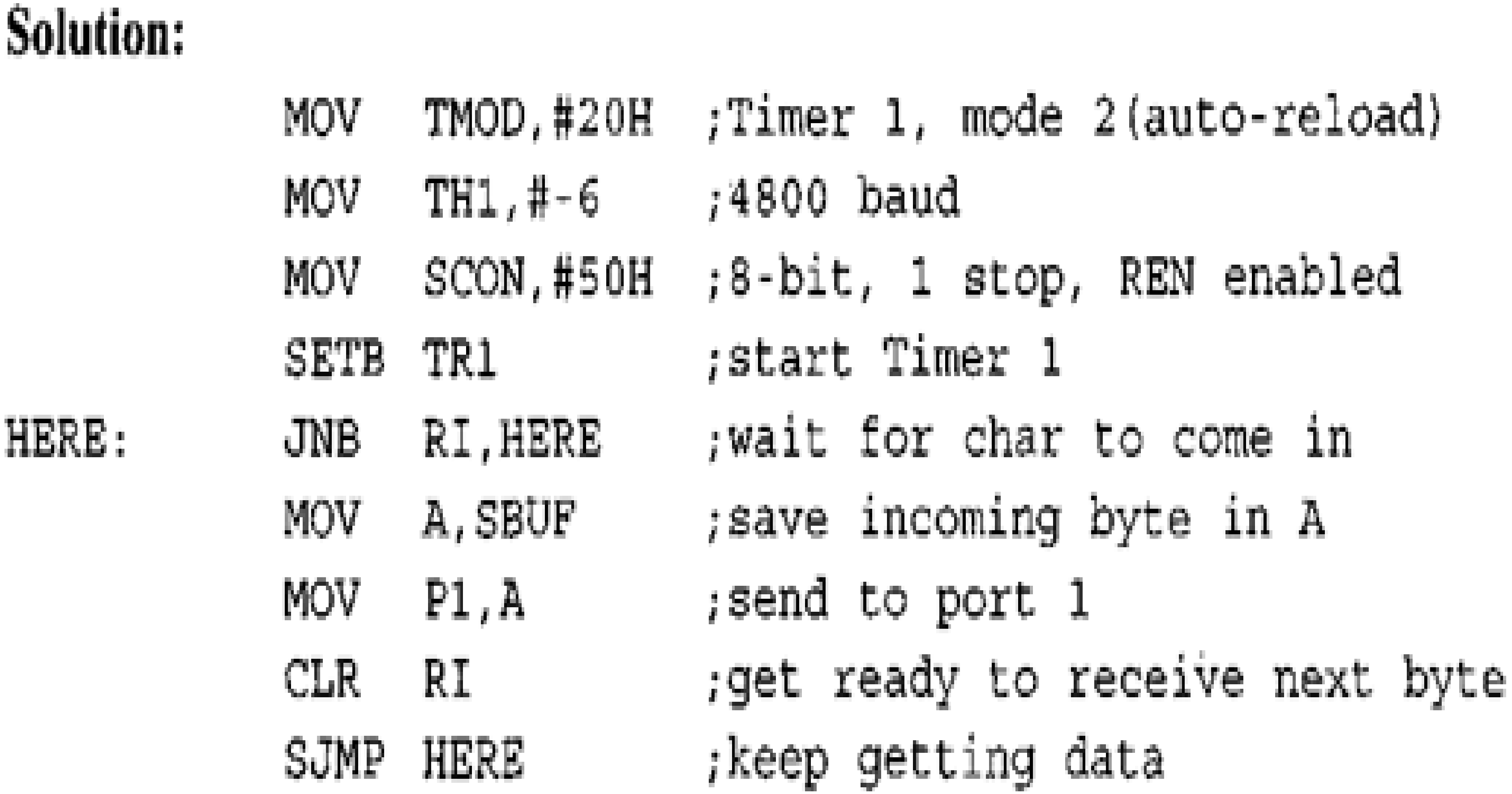
Write a program to transfer the message “YES” serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.



Solution:

```
            MOV    TMOD,#20H ;Timer 1, mode 2
            MOV    TH1,#-3   ;9600 baud
            MOV    SCON,#50H ;8-bit, 1 stop bit, REN enabled
            SETB   TR1       ;start Timer 1
AGAIN:      MOV    A,#"Y"    ;transfer "Y"
            ACALL TRANS
            MOV    A,#"E"    ;transfer "E"
            ACALL TRANS
            MOV    A,#"S"    ;transfer "S"
            ACALL TRANS
            SJMP   AGAIN     ;keep doing it
;-----serial data transfer subroutine
TRANS:      MOV    SBUF,A    ;load SBUF
HERE:       JNB    TI,HERE   ;wait for last bit to transfer
            CLR    TI        ;get ready for next byte
            RET
```

Program the 8051 to receive bytes of data serially, and put them in PI. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

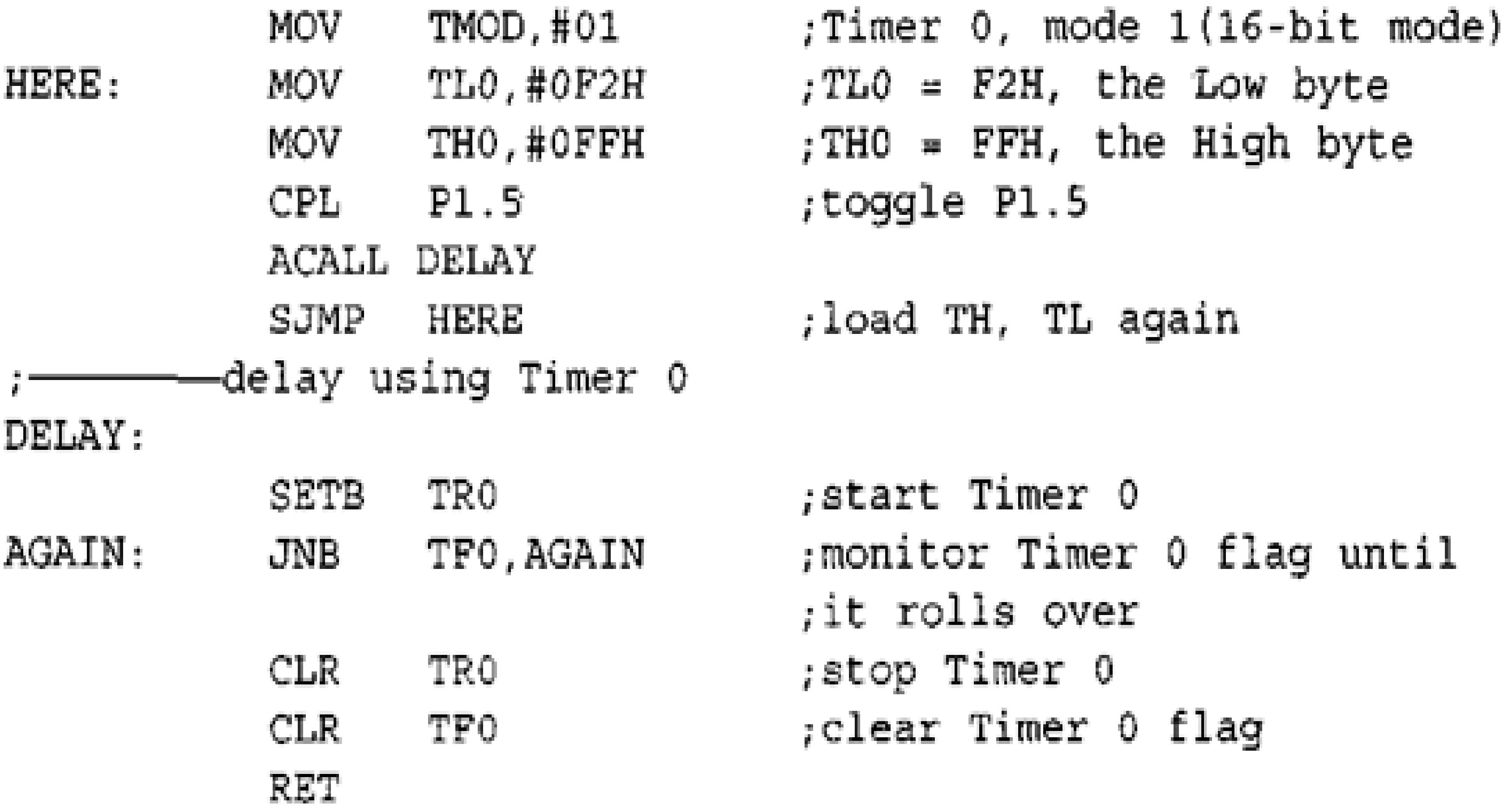


Solution:

```
            MOV   TMOD,#20H   ;Timer 1, mode 2(auto-reload)
            MOV   TH1,#-6     ;4800 baud
            MOV   SCON,#50H   ;8-bit, 1 stop, REN enabled
            SETB  TR1         ;start Timer 1
HERE:       JNB   RI,HERE     ;wait for char to come in
            MOV   A,SBUF      ;save incoming byte in A
            MOV   P1,A        ;send to port 1
            CLR   RI          ;get ready to receive next byte
            SJMP  HERE        ;keep getting data
```

PROGRAMMING TIMERS OF 8051

In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PI.5 bit. Timer 0 is used to generate the time delay. Analyze the program.



```
          MOV    TMOD,#01        ;Timer 0, mode 1(16-bit mode)
HERE:     MOV    TL0,#0F2H       ;TL0 = F2H, the Low byte
          MOV    TH0,#0FFH       ;TH0 = FFH, the High byte
          CPL    P1.5            ;toggle P1.5
          ACALL  DELAY
          SJMP   HERE            ;load TH, TL again
;————————delay using Timer 0
DELAY:
          SETB   TR0             ;start Timer 0
AGAIN:    JNB    TF0,AGAIN       ;monitor Timer 0 flag until
                                 ;it rolls over
          CLR    TR0             ;stop Timer 0
          CLR    TF0             ;clear Timer 0 flag
          RET
```

Assume that XTAL = 11.0592 MHz. What value do we need to load into the timer's registers if we want to have a time delay of 5 ms (milliseconds)? Show the program for Timer 0 to create a pulse width of 5 ms on P2.3.

Solution:

Since XTAL = 11.0592 MHz,

- the counter counts up every 1.085 us.
- This means that out of many 1.085 us intervals we must make a 5 ms pulse.
- To get that, we divide one by the other.
- We need 5 ms/1.085 us = 4608 clocks.
- To achieve that we need to load
- TL and TH with the value 65536 – 4608 = 60928 = EEOOH.
- Therefore, we have TH = EE and TL = 00

```
        CLR   P2.3              ;clear P2.3
        MOV   TMOD,#01          ;Timer 0, mode 1 (16-bit mode)
HERE:   MOV   TL0,#0            ;TL0 = 0, Low byte
        MOV   TH0,#0EEH         ;TH0 = EE( hex), High byte
        SETB  P2.3              ;SET P2.3 high
        SETB  TR0               ;start Timer 0
AGAIN:  JNB   TF0,AGAIN         ;monitor Timer 0 flag
                                ;until it rolls over
        CLR   P2.3              ;clear P2.3
        CLR   TR0               ;stop Timer 0
        CLR   TF0               ;clear Timer 0 flag
```

Assuming that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 kHz frequency on pin PI .5.

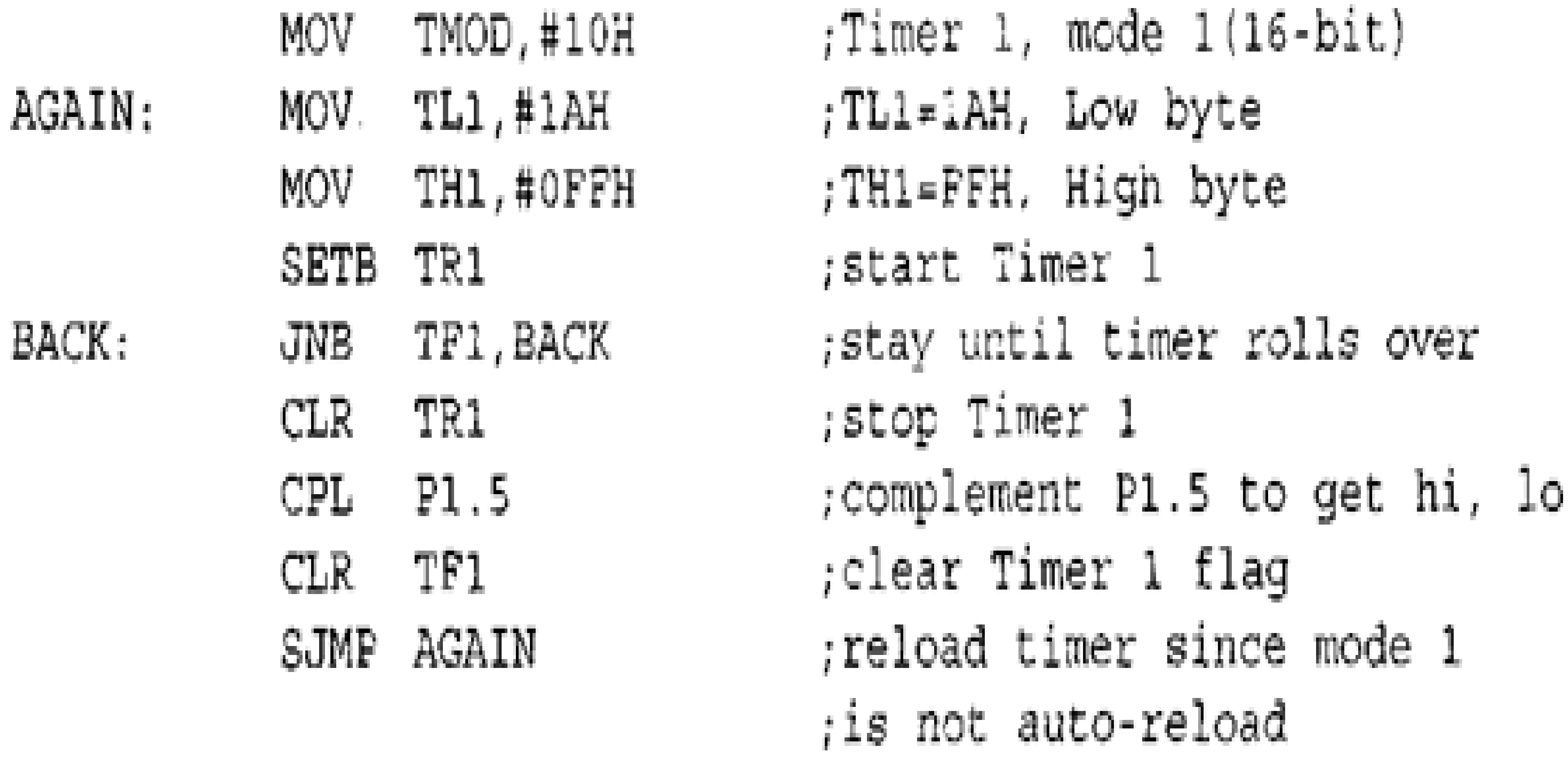
Solution:

1)T = 1 / f = 1 / 2 kHz = 500 us the period of the square wave.

2)1/2 of it for the high and low portions of the pulse is 250 us.

3)250 us / 1.085 us = 230 and 65536 – 230 = 65306. which in hex is FF1AH.

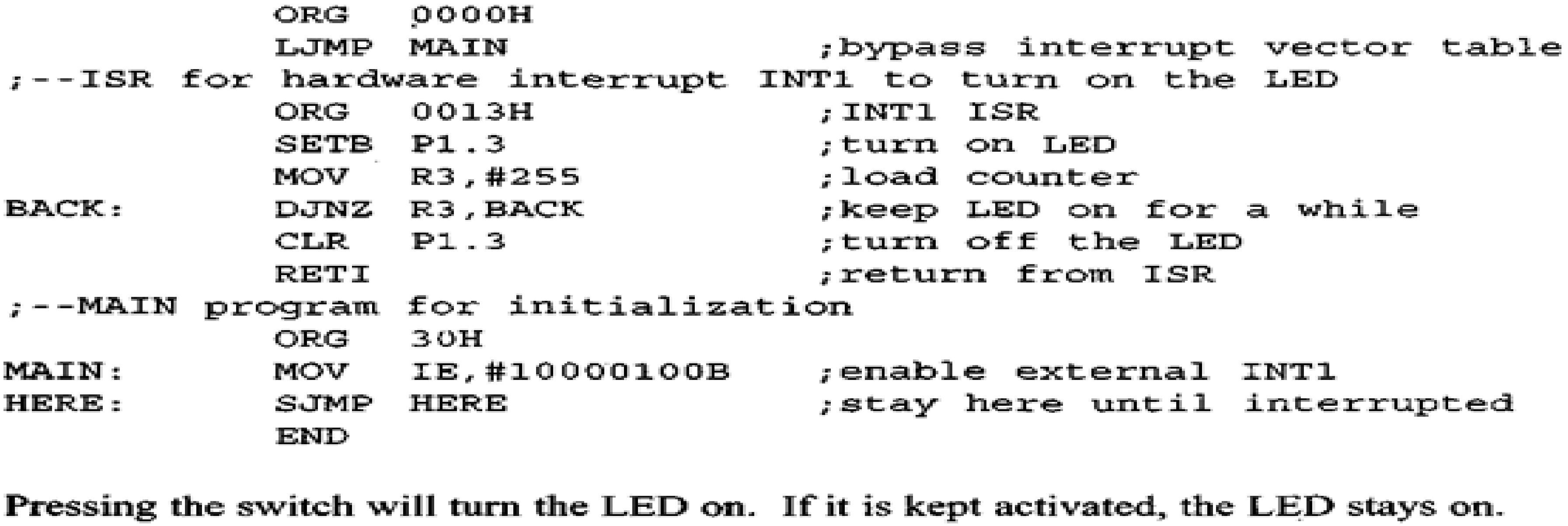
4)TL = 1AH and TH = FFH. all in hex. The program is as follows.



```
        MOV   TMOD,#10H       ;Timer 1, mode 1(16-bit)
AGAIN:  MOV   TL1,#1AH        ;TL1=1AH, Low byte
        MOV   TH1,#0FFH       ;TH1=FFH, High byte
        SETB  TR1             ;start Timer 1
BACK:   JNB   TF1,BACK        ;stay until timer rolls over
        CLR   TR1             ;stop Timer 1
        CPL   P1.5            ;complement P1.5 to get hi, lo
        CLR   TF1             ;clear Timer 1 flag
        SJMP  AGAIN           ;reload timer since mode 1
                              ;is not auto-reload
```

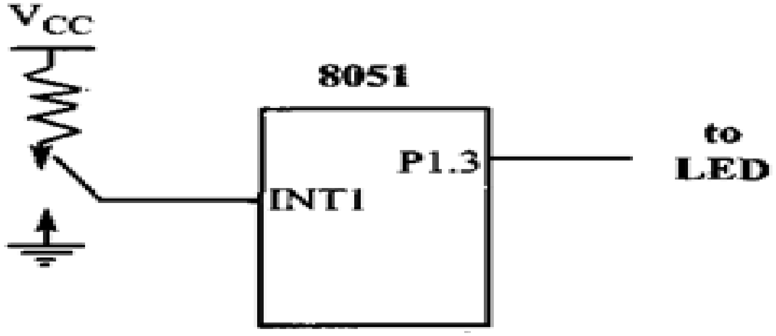
PROGRAMMING EXTERNAL INTERRUPTS OF 8051

Assume that the INT1 pin is connected to a switch that is normally high. Whenever it goes low, it should turn on an LED. The LED is connected to PI .3 and is normally off. When it is turned on it should stay on for a fraction of a second. As long as the switch is pressed low, the LED should stay on.

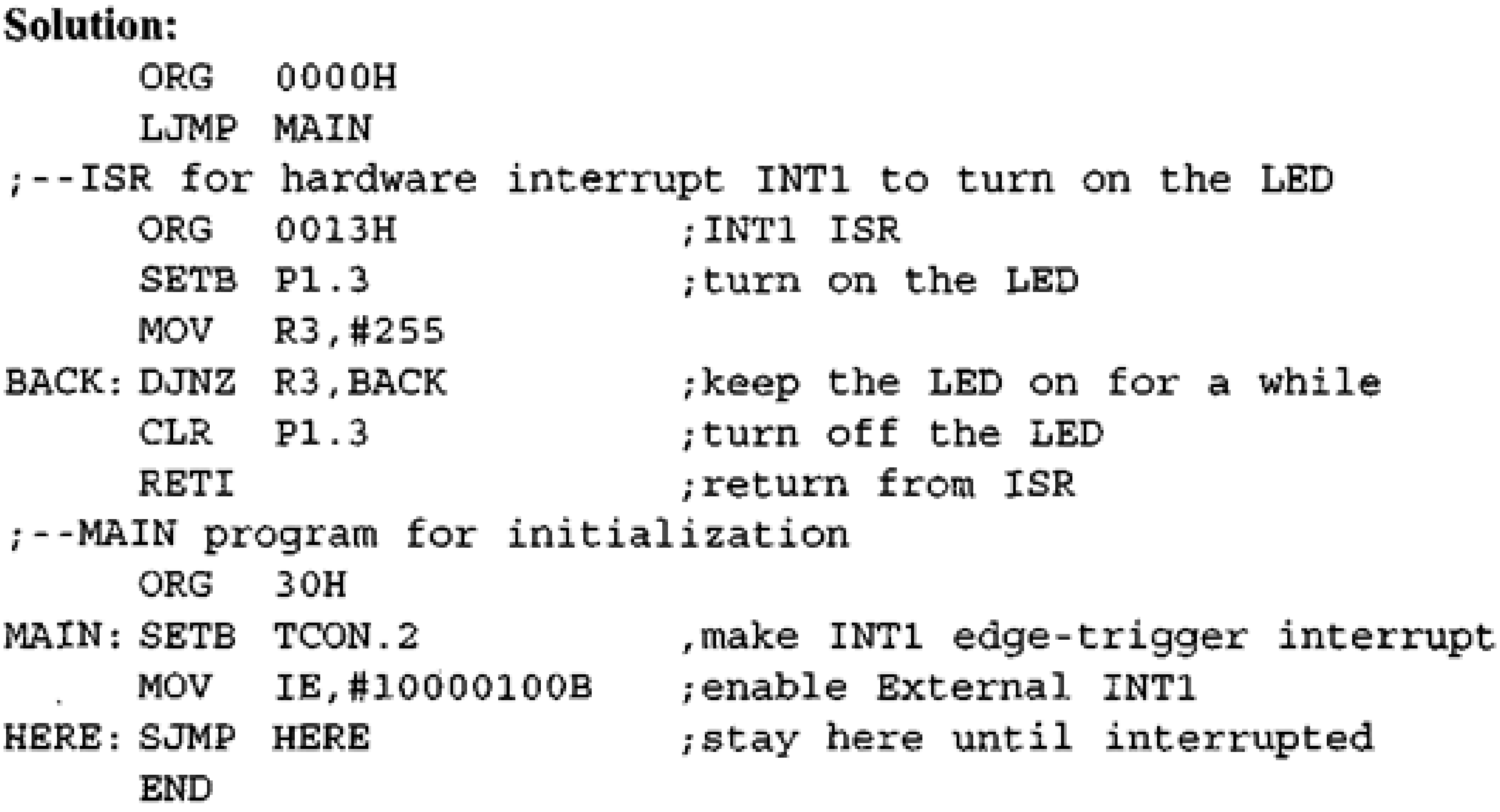


```
          ORG   0000H
          LJMP  MAIN                ;bypass interrupt vector table
;--ISR for hardware interrupt INT1 to turn on the LED
          ORG   0013H               ;INT1 ISR
          SETB  P1.3                ;turn on LED
          MOV   R3,#255             ;load counter
BACK:     DJNZ  R3,BACK             ;keep LED on for a while
          CLR   P1.3                ;turn off the LED
          RETI                      ;return from ISR
;--MAIN program for initialization
          ORG   30H
MAIN:     MOV   IE,#10000100B       ;enable external INT1
HERE:     SJMP  HERE                ;stay here until interrupted
          END
```

Pressing the switch will turn the LED on. If it is kept activated, the LED stays on.



Assuming that pin 3.3 (INT1) is connected to a pulse generator, write a program in which the falling edge of the pulse will send a high to PI.3, which is connected to an LED (or buzzer). In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT1 pin.



Solution:

```
      ORG   0000H
      LJMP  MAIN
;--ISR for hardware interrupt INT1 to turn on the LED
      ORG   0013H             ;INT1 ISR
      SETB  P1.3              ;turn on the LED
      MOV   R3,#255
BACK: DJNZ  R3,BACK           ;keep the LED on for a while
      CLR   P1.3              ;turn off the LED
      RETI                    ;return from ISR
;--MAIN program for initialization
      ORG   30H
MAIN: SETB  TCON.2            ,make INT1 edge-trigger interrupt
      MOV   IE,#10000100B     ;enable External INT1
HERE: SJMP  HERE              ;stay here until interrupted
      END
```