

Microprocessor and microcontroller

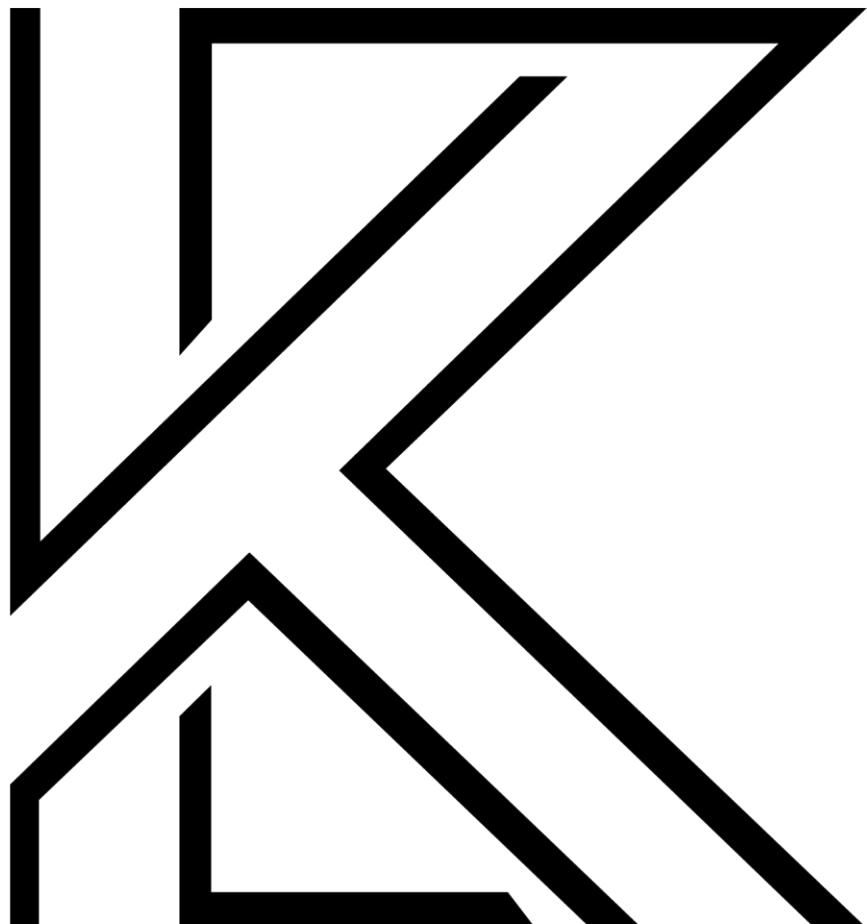
فَإِنَّ مَعَ الْعُسْرِ يُسْرًا ﴿٥﴾

إِنَّ مَعَ الْعُسْرِ يُسْرًا ﴿٦﴾

*For indeed, with hardship comes ease.
Indeed, with hardship comes ease.*

[Surah Iushiraah Ayah 5-6]

Course Teachers
Md. Farhad Hossain | Arif Istiaque Rupom



Remember in your prayers.

INDEX

Section A:

Introduction	5
Turing Machine	7
Computer Memory.	8
John Von Neumann Architecture	9
SAP-1 Architecture	11
Real Mode memory addressing	14
Protected mode memory addressing.	17
Registers of 8086	18
Flags	20
Data Addressing Modes	22
PyQ	26
Architecture of 8086	28
Bus Interface Unit of 8086	29
Working of execution unit	30
ALU	31
Machine Language	32
Byte-1 and Byte-2 Machine Language.	33
Examples solved	35
PUSH-POP Instructions	36
Program Invisible Register	38
Addition	39
Subtraction	41
Comparison	42
Multiplication	43
Division	44
Basic Logic Instructions	44
SHIFT,ROTATE	47
JUMP	48
Procedures	50
Interrupt	51
HLT,NOP,LEA	52
PyQ's	54
Some Problems with Solutions	59

Section B:

Hardware Specifications of 8086	71
Demux Address Bus Diagram	73
Read Timing Diagram	74
Fully Buffered 8086 Diagram	75
Write Timing diagram	76
Microcontrollers	77
8255A PPI	77
Examples on 8255A	87
More Problems Suggested	91
8254 PIT	95
Examples on 8254	104
8259A PIC	109
DMA controller	113
Assembly Codes QnA and PyQ's	115

Microprocessor:

A microprocessor is a programmable device that takes input, processes it by performing arithmetic and logical operations, and generates the desired output.

In simple terms, it is a compact digital chip that can fetch instructions from memory, decode them, execute the operations and deliver results.

It acts as the brain of many systems, enabling them to process data and perform tasks efficiently.

Combinational vs Sequential logic

Combinational	Sequential.
<ol style="list-style-type: none">1. Outputs depend on the current inputs only.2. Memory less, does not store any previous state.3. Operates continuously based on input changes.4. Simple design and implementation.5. Cannot handle sequential tasks.6. Examples: Adders, multiplexers, decoders.	<ol style="list-style-type: none">1. Outputs depend on both current input and past states.2. Has memory elements to store past states.3. Operates based on clock signal and timing.4. More complex design due to the inclusion of memory.5. Can handle sequential tasks and state transitions.6. Examples: Flip-flops counter, registers etc.

Calculator vs Microprocessor

Calculator	Microprocessor
<ol style="list-style-type: none">1. Primarily uses combinational logic for intermediate calculations.2. Lacks memory to store past calculations or states.3. Performs isolated, predefined operations.	<ol style="list-style-type: none">1. Combines sequential and combinational logic to execute instructions and manage processes.2. Utilizes memory to store instructions, data, and states.3. Executes a sequence of operations dynamically based on a program.

④ What is Turing Machine - Explain it's Working?

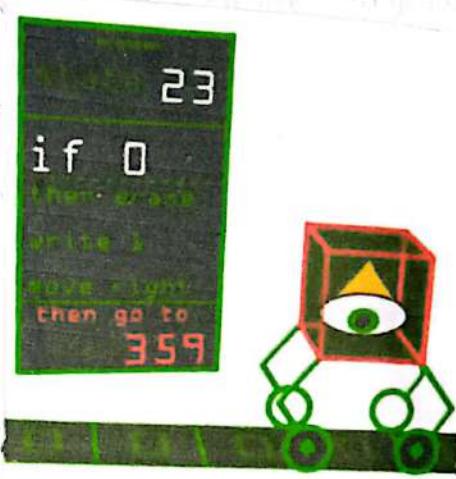
A turing machine is a theoretical computational model introduced by Alan Turing in 1936.

It consists of an infinitely long tape, a head that reads and writes symbols on the tape and a set of rules (or state) that determine the machine's behaviour based on the current symbol under the head and its current state. The machine moves left or right on the tape, reads/writes symbols, and changes states until it halts.

The machine works by:

1. Reading a symbol from the tape.
2. Changing the symbol if necessary.
3. Moving left or right based on a defined set of rules.
4. Transitioning to a new state and repeating the process.

The turing machine is foundational to modern computing, as it formalized the concept of computation and computability, laying the groundwork for the design of the first general-purpose computers by influencing the development of digital logic and algorithms.



Computer Memory:

Computer memory refers to the components or devices used to store data and instructions that are processed by the computer's CPU. It is essential for the operation of the system, enabling quick data retrieval.

There are two primary types of memory:

1. Primary Memory (RAM): Which is fast and volatile, meaning it loses its data when power is off.
2. Secondary Memory (Hard Drives, SSDs): Which is slower but retains data permanently.

Cache Memory is another crucial component, located near the CPU to speed up access to frequently used data.

Computer Memory Mechanism:

RAM: uses transistors and capacitors to store data. Transistors act as a switch, capacitors store bits of data in the form of electrical charges. When capacitor is charged '1' is stored, and when discharged '0' is stored.

HDD/SSD: Includes magnetic and optical storage, where data is stored on magnetic disks or optical media.

HDD: Magnetic storage using rotating platters and read/write heads.

SSD: Flash memory that stores data using floating gate transistor.

ROM: Data is permanently stored using semiconductor technology and cannot be altered easily.

Example: BIOS stored in ROM for booting up a computer.

Q How does cache memory speed up the operation of a microprocessor?

A Cache memory speeds up microprocessor operations by storing frequently accessed data closer to the CPU. This reduces the time needed to fetch data from slower main memory (RAM). Cache operates at a much higher speed than RAM, allowing the CPU to access and process data more efficiently, significantly improving overall performance. It reduces bottlenecks by providing faster access to data, enabling quicker execution of instructions.

S-RAM

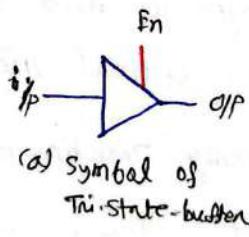
1. Faster, as it uses flip-flops to store data.
2. More expensive due to faster speed and complexity.
3. Used in Cache memory, registers and buffers.
4. Retains data as long as power is supplied.
5. Required less power than DRAM in active mode.

D-RAM

1. Slower, uses capacitors that need constant refreshing.
2. cheaper and more widely used due to simpler design.
3. Used for main system memory (RAM) in most computers.
4. Data is lost when power is turned off.
5. Consumes more power due to constant refreshing.

Q What is Tri-state Buffer? - Explain.

Q A Tri-state buffer is a digital circuit element with three output states: High '1', low '0' and high impedance 'Z'. The high impedance state effectively disconnects the buffer from the circuit, allowing multiple devices to share the same communication line (e.g. common bus) without interference. The buffer is controlled by an enable pin; when enabled, it passes the input signal to the output, and when disabled, it enters the high impedance state, preventing conflicts or short circuits in shared circuitry.



(a) Symbol of Tri-state-buffer

En	input	output
0	X	Z
1	0	0
1	1	1

(b) Truth table for Tri-state buffer.

John Von Neumann Architecture:

The Von Neumann Architecture is a computer design framework where a single bus system connects the CPU, memory and input/output (I/O) devices.

CPU: Central Processing Unit, handles digital data processing. Comprises an ALU (for computations), registers and control unit.

Input/Output: Converts Analog data to digital for the CPU using ADC/DAC and vice-versa.

Memory: Stores both instructions and data.

The buses in between CPU to Memory link those components.

Address bus - Specifies memory location. One directional CPU to RAM.

Data bus - Transfers data between components, bi-directional.

Control bus - Manages data flow and operations. uni-directional CPU-RAM.

This design enables sequential execution, integrating computation and storage efficiently.

Though modern CPU uses each bus as bi-directional.

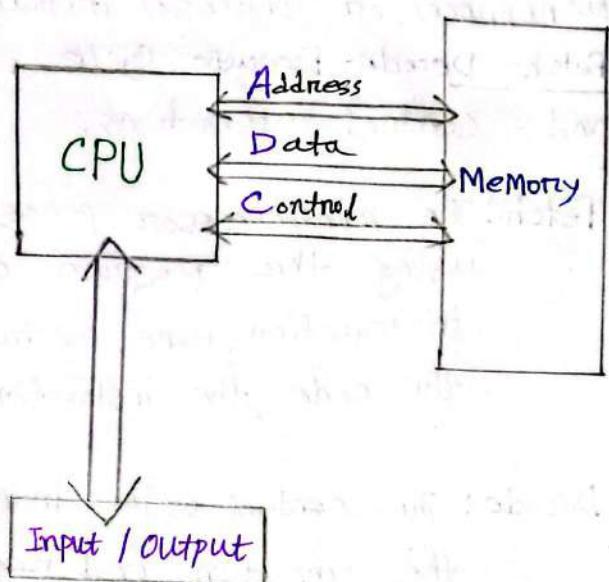


Figure: John Von Neumann Architecture

Working Steps of Microprocessor:

Microprocessor processes instructions in a sequence called the Fetch-Decode-Execute Cycle, enabling it to perform computations and control operations.

Fetch: The microprocessor processes instructions from memory using the program counter (PC), which tracks the next instruction. Here, practically microprocessor tells RAM to send the code, the instruction flow is from RAM to CPU.

Decode: The control unit interprets the fetched instruction, identifying the operation and required data. Compilers/Assembler converts the data from high-to-low level machine language.

Execute: The instruction is carried out by the CPU, involving the ALU to other components and results are stored as output when needed.

This cycle repeats for each instruction in the program.

□ Explain the role of control sequence in the process?

□ The control sequence (CS) manages the operation flow by generating control signals. These signals ensure hardware units like the program counter (PC), Registers, RAM and ALU work in synchronization. It acts as the bridge between hardware and software via firmware using microinstructions stored in control logic.

During system startup, BIOS initializes hardware and loads the 'Bootloader', which enables the operating system to communicate with hardware. This (FS) ensures seamless execution of micro-instructions, making hardware and software integration efficient.

SAP-1 Architecture:

The SAP-1 (Simple As possible-1) computer architecture is a basic design that demonstrates fundamental computer operations, with components interconnected through a common bus.

Registers: Temporary storage for data, connected to the ALU. Each register has an internal tri-state buffer to enable selective communication with the bus.

Arithmetic Logic Unit: Positioned on the right side of the bus, it performs basic operations using data from registers A and B.

Program Counter (PC): Tracks the next instruction's address and has an enable pin to activate it only during instruction fetch cycles.

RAM: Stores instructions and data located on the left side of the bus.

Control Sequence: Directs system operations like instruction decoding, execution and data flow control.

Tri-state Buffers: Prevents short circuits on the bus by ensuring only one component communicates at a time.

Operation Flow: Instructions are fetched from RAM by the PC, processed through ALU, and data moves between components via the bus under the CS's control.

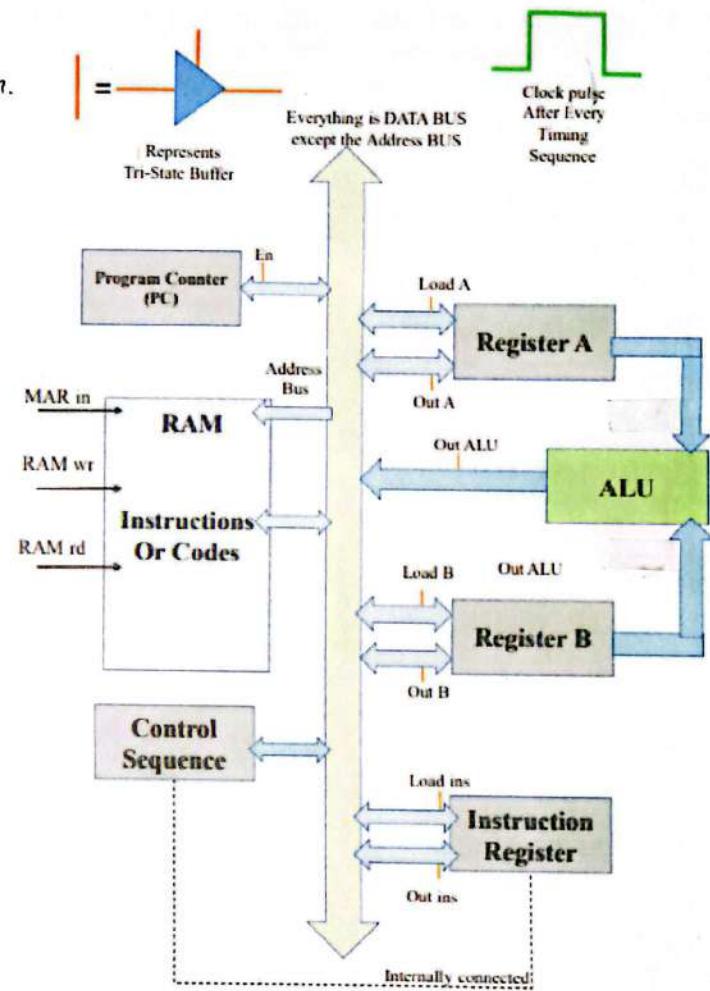
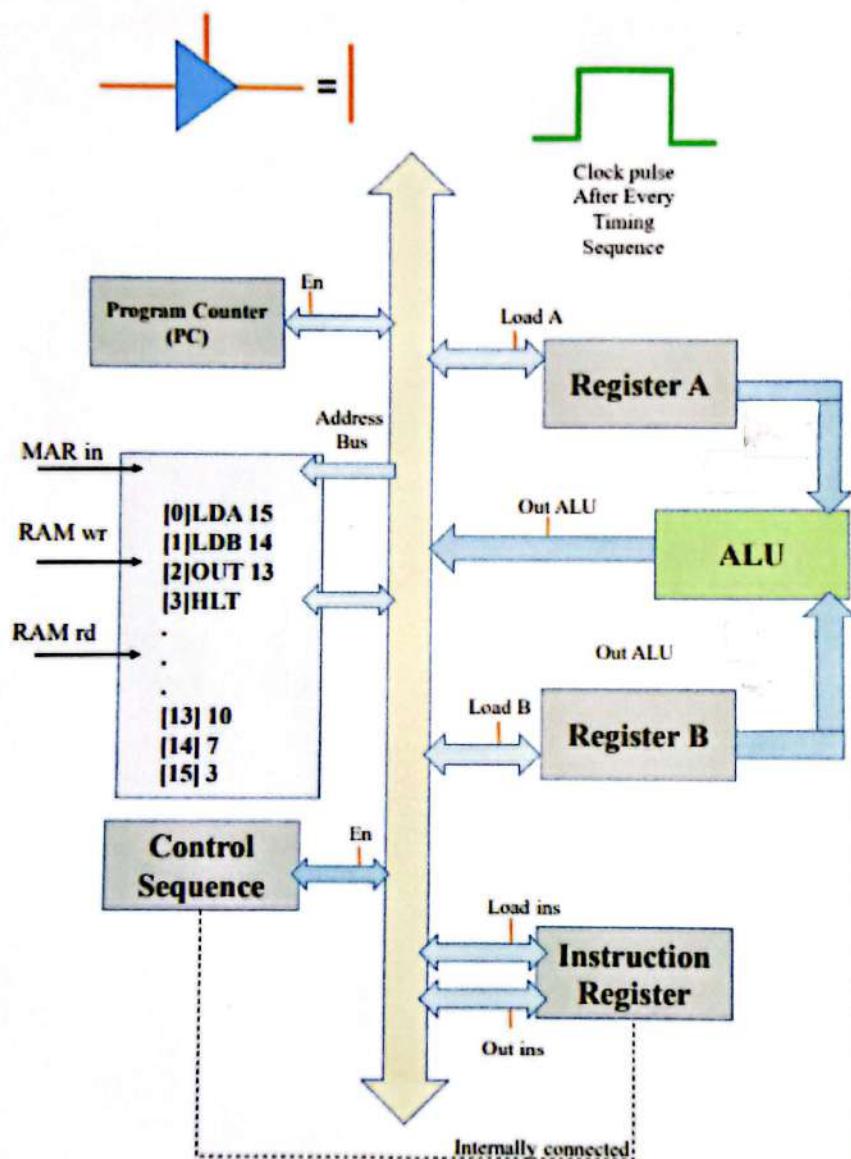


Figure : SAP-1 Architecture (Simplified)



Stage	Pin On/Off	Comments
LDA 15		
Fetch	T₁ Pc "En", Cs "En", Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [0] is selected
	T₂ Ram_rd, Load ins, Clk pulse($\uparrow\downarrow$), All pins off.	Instruction from Ram[0] Sends to Ins Register
	T₃ Pc (\uparrow), Clk pulse($\uparrow\downarrow$), All pins off.	Program counter incremented for next stage.
Decode	Instruction Decoded as "Load Data from Ram Line [15] to Register A"	
Execute	T₄ Out ins, Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [15] is selected
	T₅ Ram_rd, Load A, Clk pulse($\uparrow\downarrow$), All pins off.	Data (3) Loaded and Saved to Register A
LDA 14		
Fetch	T₁ Pc "En", Cs "En", Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [1] is selected
	T₂ Ram_rd, Load ins, Clk pulse($\uparrow\downarrow$), All pins off.	Instruction from Ram[1] Sends to Ins Register
	T₃ Pc (\uparrow), Clk pulse($\uparrow\downarrow$), All pins off.	Program counter incremented for next stage.
Decode	Instruction Decoded as "Load Data from Ram Line [14] to Register B"	
Execute	T₄ Out ins, Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [14] is selected
	T₅ Ram_rd, Load B, Clk pulse($\uparrow\downarrow$), All pins off.	Data (7) Loaded and Saved to Register B
ALU Operation	Out A, Out B, ALU "En", Clk pulse($\uparrow\downarrow$), All pins off.	Data (3 & 7) are sent to ALU for Operation (Lets Assume Operation is Addition 3+7=10)
OUT 13		
Fetch	T₁ Pc "En", Cs "En", Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [2] is selected
	T₂ Ram_rd, Load ins, Clk pulse($\uparrow\downarrow$), All pins off.	Instruction from Ram [2] Sends to Ins Register
	T₃ Pc (\uparrow), Clk pulse($\uparrow\downarrow$), All pins off.	Program counter incremented for next stage.
Decode	Instruction Decoded as "Show Output Data from Ram Line [13]"	
Execute	T₄ Out ins, Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [13] is selected
	T₅ Ram_wr, Clk pulse($\uparrow\downarrow$), All pins off.	Data (10) is send via common Data Bus to Output
HLT		
Fetch	T₁ Pc "En", Cs "En", Mar_in, Clk pulse($\uparrow\downarrow$), All pins off.	Ram Address [3] is selected
	T₂ Ram_rd, Load ins, Clk pulse($\uparrow\downarrow$), All pins off.	Instruction from Ram [3] Sends to Ins Register.
	T₃ PC Haults	Program Counter Haults, CPU Stops.

Q1 Explain Behavioral and structural Design of microprocessor.

A The behavioral design focuses on the functionality of the microprocessor how it processes instructions, manages control signals and executes tasks. It defines the logical flow of data, instruction decoding, and arithmetic logic operations, abstracting away hardware specifics. The truth table

Structural design of Microprocessor emphasizes the physical organization and components of the microprocessor, such as ALUs, Registers, cache buses, and transistor level implementation. It deals with the arrangement of these components to achieve desired behaviours. The K-map extracted from truth table.

Q2 What is X86 Architecture?

A The X86 architecture is a family of instruction set architectures initially developed by Intel. It defines how a processor handles instructions and interacts with memory. Widely used in PCs. It ensures compatibility for running operating systems and applications across multiple processor brands like Intel and AMD.

Devices using X86 (i.e. Intel and AMD) processors behave similarly, executing the same instructions due to a shared architecture. However their internal structures - core designs, cache configurations and fabrication processes vary, impacting performance, efficiency and power consumption.

Q3 Why are co-processors are used with CPU's?

A Co-processors are specialized processors that assist the CPU by handling specific tasks, such as floating-point calculations, graphics processing or AI predictions. This offloading allows the CPU to focus on general purpose tasks, improving efficiency and speeds.

Co-processors predict repetitive or resource intensive operations, ensuring faster and optimized execution, particularly in data heavy applications like scientific computations or multimedia processing.

Real mode memory Addressing:

Real mode memory addressing is a basic memory addressing scheme by the 8086 microprocessor, which accesses physical memory by combining segment and offset addresses. It divides memory into 64 KB, addressed via a 20 bit physical address calculated as.

$$\text{PA} = \text{CS} \times 10 + \text{IP}$$

physical address → Segment ↑ Offset ↑

If microprocessor accesses only the 1 b of the memory, then we call it the real mode addressing?

8086 has a 16-bit architecture but uses a 20-bit address bus to access upto 1Mb ($= 2^{20}$ bit) memory. Real mode enables compatibility with older 16-bit software by segmenting memory for efficient addressing.

8086 microprocessor has 20 lines which can access any location of the RAM and can perform read, write operations.

Role of OS:

In real mode, the operating system OS acts as a simple manager for basic hardware resources. OS doesn't control any read or write operation of the CPU.

OS primarily:

loads and execute programs: Handles program loading and allocates memory within 1 Mb limit.

Manages interrupts: sets up and handles interrupt vectors for hardware communication.

Provides basic I/O Services: Offers low level access to hardware like disk drives and display.

The OS lacks advanced features like multitasking, advanced protection of the memory or user space-kernel space separation, due to real mode limitations.

Working of real mode addressing:

- The CPU selects a segment using one of the segment registers (CS, DS, SS, ES)
- It adds a **16-bit** offset to calculate the effective **20-bit** physical address.

Example: Show how real mode addressing is done to access the memory address **20500X** for the code.

Physical address,

$$\begin{aligned} PA &= CS \times 10 + IP \\ &= 2000 \times 10 + 500 \\ &= 20500X \end{aligned}$$

or $\frac{20500X}{10}$
Result: CS
Reminder: IP.

Example: Suppose we have to store an Andia to **40800X**.

Physical address,

$$\begin{aligned} PA &= DS \text{ or } ES \times 10 + SI \\ &= 4000 \times 10 + 800 \\ &= 40800X \end{aligned}$$

Limitations:

- Can access only 1MB of memory locations.
- No memory protection, thus wrong code can be read/write unconsciously.
- Segmentation of the RAM complicates programming.

Fig.1 shows how the segment plus offset addressing scheme selects a memory location. This illustration shows a memory segment that begins at location **10000H** and ends at location **1FFFFH** - 64KB in length. It also shows how an offset address sometimes called a displacement of **F000H** selects location **1F000H** in the memory system.

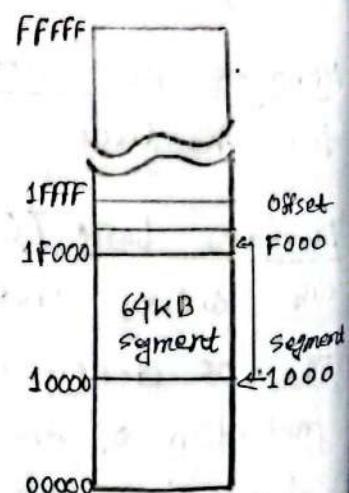


Fig.1: Real mode memory Address

Protected Mode memory addressing:

Protected mode memory addressing (80286 and above) allows access to data and programs located above the first 1M byte of memory, as well as within the first 1M byte of memory.

Why protected mode memory addressing is required?

It uses advanced features like memory protection, virtual memory and multitasking, essential for modern operating systems.

1. Extended Memory Access: It allows addressing memory beyond 1MB, up to 4GB (or more in 64-bit systems).
2. Memory Protection: Prevents unauthorized memory access, crucial for stability and security.
3. Multitasking: Facilitates safe execution of multiple processes by isolating memory segments.
4. Virtual memory: Enables efficient memory usage by mapping virtual addresses to physical memory.

How it works?

1. Segment registers (e.g. CS, DS) store selectors instead of segment addresses. A selector points to a descriptor in a descriptor table. Descriptor table describes memory segments, including base addresses, size (limit) and access rights. Two tables exist:
 - Global Descriptor Table (GDT): Shared across all processes.
 - Local Descriptor Table (LDT): Specific to individual processes.
2. The CPU combines the base address from the descriptor and the offset address to compute the effective memory address.
3. Access rights control read, write or execution permissions, ensuring security.

The operating system configures and manages descriptor tables (GDT/LDT). It handles memory segmentation and enforces access rights. It transitions the CPU to protected mode during boot and co-ordinates multitasking by isolating and protecting process memory.

Q Write down the name of all general purpose register, Segment register and special purpose register in 8086 UP.

Q The general purpose registers are:

1. AX (Accumulator)
2. BX (Base index)
3. CX (Count)
4. DX (Data)
5. BP (Base pointer)
6. SI (Source Index)
7. DI (Destination Index)

Q The Segment Registers are:

1. CS (Code segment)
2. DS (Data segment)
3. ES (Extra segment)
4. SS (Stack segment)

Q The Special purpose registers are:

1. (Instruction pointer) IP.
2. (Stack pointer) SP.
3. FLAGS.

Q Justify "code segment register cannot be distributive register".

Q The address of next instruction should be execute found in both IP and CS. If only CS is changed, the address of the next instruction would be unpredictable. Thus Code segment cannot be changed. This is why code segment register cannot be a distributive register.

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

FLAGS

IP.

CS
DS
ES
SS

Registers of 8086:

Registers in 8086 and above processors are small storage units within the CPU used for temporary data storage and operations. They are categorized as below.

General purpose Registers:

Performs arithmetic, logical data manipulation and temporary data storage.

Ax: Accumulator register performs arithmetic and logical operations handle input/output data.

Bx: Base register stores base addresses for memory operations.

Cx: Count register used in loop and string operations for counting iterations.

Dx: Data registers handle data during multiplication and division. can store high order results.

Segment registers:

Cs: Code segment register points to executable instructions.

Ds: Data segment register holds the base addresses of data in memory.

Ss: Manages the stack for function calls and local variables.

Es: Extra segment register used in string and memory operation.

Pointers and Index registers:

Sp and Bp: Stack pointer and Base pointer manage stack operation and frame pointers.

Si and Di: Source and destination index used for data movement, specially in string operation.

Special purpose registers:

Ip: Instruction pointer tracks the next instruction to execute.

Flag registers: indicates the states of operations and controls processor functions.

Q Describe the function of various flags bits of 8086 CPU.

The diagram illustrates a 16-bit register structure with the following labels and bit assignments:

- Overflow**: Bit 15 (labeled with an arrow)
- Interrupt sign**: Bit 14 (labeled with an arrow)
- Auxiliary**: Bit 13 (labeled with an arrow)
- Carry**: Bit 12 (labeled with an arrow)
- Direction**: Bit 11 (labeled with an arrow)
- Trap**: Bit 10 (labeled with an arrow)
- Zero**: Bit 9 (labeled with an arrow)
- Parity**: Bit 8 (labeled with an arrow)
- X**: Bits 7, 6, 5, 4 (all labeled with an arrow)
- 0**: Bit 3 (labeled with an arrow)
- D**: Bit 2 (labeled with an arrow)
- I**: Bit 1 (labeled with an arrow)
- T**: Bit 0 (labeled with an arrow)
- X**: Bits 15, 14, 13, 12 (all labeled with an arrow)
- A**: Bit 11 (labeled with an arrow)
- P**: Bit 10 (labeled with an arrow)
- X**: Bit 9 (labeled with an arrow)
- C**: Bit 8 (labeled with an arrow)

(i) Carry flag: Holds the carry after addition and borrow after subtraction.

If $c=1$, borrow or carry present.

$c \geq 0$, otherwise

(ii) Parity flag: It checks data transmission by checking if the number of set bits in the result is even ($PF=1$) or odd ($PF=0$).

(ii) Auxiliary carry: Signals carry/borrow between the lower four bits (half carry, half borrow) after the addition and subtraction respectively. $A=0$; carry absent, $A=1$ carry present.

(iv) Zero flag: It indicates if the result of an operation is zero or not $Z=1$; result is zero, $Z=0$; result is not zero.

(v) Sigh flag) It indicates whether the result of an operation is negative ($S=1$) or positive ($S=0$).

(vi) Trap flag: Enables single step execution for debugging; generating and interrupting after each instruction.

$T > 1$; debugging is enabled.

T=0; debugging is disabled.

(vii) Direction flag) Controls the processing direction of string operators (movement or decimation).

D=0; increment mode; cLD

$D=1$; decrementation mode; STD

(viii) Interrupt flag: Controls the operation of INTR pin.

$I=1$; Interrupt pin is enabled.

$I=0$; Interrupt pin is disabled.

(ix) Overflow flag: Indicates if the sign arithmetic result exceeds the capacity of the destination.

' $O=1$ '; overflow present.

' $O=0$ '; overflow absent.

Data addressing modes:

MOV instruction is a very common and flexible instruction, it provides a basis for the explanation of data-addressing modes.

The figure 3-1 illustrates the MOV instruction and defines the direction of data flow. The source is to the right and the destination is to the left, next to the opcode MOV.

An opcode or operation code, tells the microprocessor which operation to perform.

Note that memory to memory data transfer is not allowed by any instruction except for the MOVS instruction. MOV instruction always copies the source data into the destination. It never actually picks the data and moves it.

The source and destinations are often called operands.

Register Addressing: Transfer a copy of a byte or word from the source register or contents of a memory location to the destination register or memory location.

Immediate Addressing: Transfer the source, an immediate byte, word, double word or quadword or data, into the destination register or memory location.

Direct Addressing: Moves a byte or word between a memory location and a register. The instruction set doesn't support memory to memory transfer, excepts with the MOVS instruction.

Register indirect Addressing: Moves a byte or word between a register and a memory location addressed by an index or base register. The index and base registers are BP, BX, DI; SI

Type	Instruction	Source	Address Generation	Destination
Register	MOV AX,BX	Register BX		Register AX
Immediate	MOV CH,3AH	Const 3AH		Register CH
Direct	MOV 1234H,AX	Register AX	DS : 10H + DISP 10000H + 1234H	Memory address 11234H
Register indirect	MOV [BX],CL	Register CL	DS : 10H + BX 10000H + 0300H	Memory address 11030H
Base-plus-index	MOV [BX+SI],BP	Register BP	DS : 10H + BX + SI 10000H + 0300H + 0200H	Memory address 11020H
Register relative	MOV CL,[BX+4]	Memory address E000H	DS : 10H + BX + 4 10000H + 0300H + 4	Register CL
Base relative-plus-index	MOV ARRAY[BX+SI],DX	Register DX	DS : 10H + ARRAY + BX + SI 10000H + 1000H + 0300H + 0200H	Memory address 11020H
Scaled index	MOV [EBX+2*ESI],AX	Register AX	DS : 10H + EBX + 2 * ES 10000H + 03000H + 00000H + 00000H	Memory address 11000H

Note: EBX = 0000000H, ESI = 0000000H, ARRAY = 1000H, and DS = 1000H

FIGURE 3-2 80386-Core2 data addressing modes

Base plus index Addressing: transfers a byte or word between a register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI).

Register Relative Addressing: Moves a byte or word between a register and the memory location addressed by an index or base register plus a displacement.

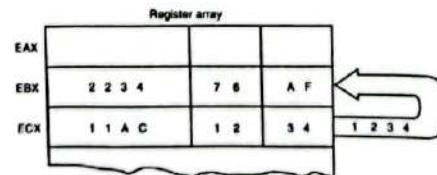
Base relative plus index addressing: transfers a bytes or word between a register and the memory location addressed by a base and an index register plus a displacement.

Register Addressing:

Explain the operation of MOV BX, CX instruction with proper diagram.

Hence the instruction $\text{MOV } \underset{\substack{\text{destination} \\ (\text{16 bit})}}{\text{BX}}, \underset{\substack{\text{source} \\ (\text{16 bit})}}{\text{CX}}$ means, move the data of the register CX into the register BX. The figure / diagram is shown below.

FIGURE 3-3 The effect of executing the $\text{MOV BX}, \text{CX}$ instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.



Immediate Addressing:

Explain the operation $\text{MOV EAX}, 3456\text{H}$ instruction with proper diagram.

Hence the instruction $\text{MOV } \underset{\substack{\text{dest} \\ (\text{32 bit})}}{\text{EAX}}, \underset{\substack{\text{source} \\ (\text{16 bit})}}{3456\text{H}}$ means, move the data 3456H immediately to the destination register EAX (32 bit). The diagram is shown below.

FIGURE 3-4 The operation of the $\text{MOV EAX}, 3456\text{H}$ instruction. This instruction copies the immediate data (13456H) into EAX.



Direct Data Addressing:

Explain the operation of $MOV AL, [1234H]$ instruction when $DS = 1000H$ with proper diagram.

Hence the given instruction $MOV AL, [1234H]$ means destination \uparrow source \uparrow Address

move data from the memory location $11234H$ into AL register, the memory location is determined via.

$$\begin{aligned} PA &= DS * 10 + [1234H] \\ &= 1000H * 10 + [1234H] \\ &= 10000H + 1234H \\ &= 11234H \end{aligned}$$

Shown in the diagram below.

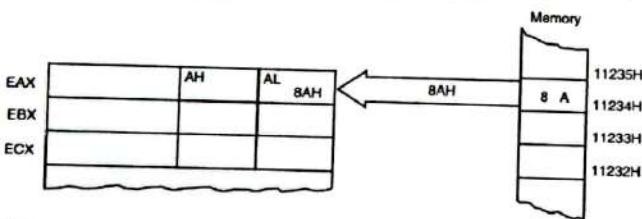


FIGURE 3-5 The operation of the $MOV AL, [1234H]$ instruction when $DS = 1000H$.

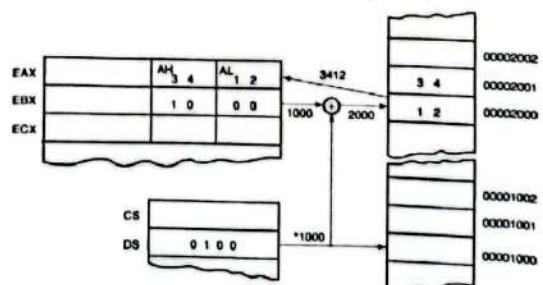
Register indirect Addressing:

Explain the operation $MOV AX, [BX]$ instruction when $BX = 1000H$ and $DS = 0100H$ with proper diagram.

Hence the given instruction $MOV AX, [BX]$ means a word stored at memory bytes $2000H$ and $2001H$, transfer it into register AX .

$$\begin{aligned} PA &= DS * 10 + [BX] \\ &= 0100H * 10 + 1000H \\ &= 2000H. \end{aligned}$$

Since AX (16 bit) can store 16 bit data, thus after storing the data of $2000H$ (8 bit) location, the data of $2001H$ (8 bit) also stored.



*After DS is appended with a 0

FIGURE 3-6 The operation of the $MOV AX, [BX]$ instruction when $BX = 1000H$ and $DS = 0100H$. Note that this instruction is shown after the contents of memory are transferred to AX .

Base plus index addressing.

Explain the operation $MOV DX, [BX+DI]$ instruction when $DS = 0100H$, $BX = 1000H$, and $DI = 0010H$.

Here the instruction $MOV DX, [BX+DI]$ means $BX = 1000H$, $DS = 0100H$ and $DI = 0010H$ which translate into memory address $02010H$ as follow.

$$DS * 10 + BX + DI$$

$$= 0100H \times 10 + 1000H + 0010H$$

$$= 02010H$$

Shown in the diagram

Register relative addressing:

Explain the operation $MOV AX, [BX+1000H]$ instruction when $BX = 0100H$ and $DS = 0200H$.

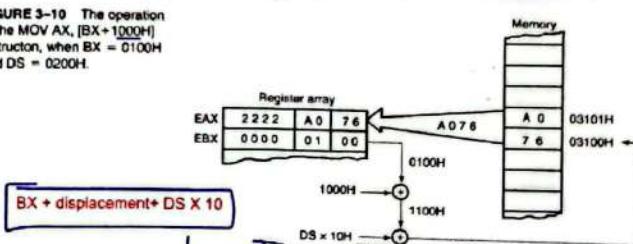
Here the instruction $MOV AX, [BX+1000H]$ means move the data from address location $03100H$ to the register AX . The source address is calculated as.

$$DS * 10 + [BX + 1000H]$$

$$= 2000H + 1100H$$

$$= 3100H$$

FIGURE 3-9 An example showing how the base-plus-index addressing mode functions for the $MOV DX, [BX + DI]$ instruction. Notice that memory address $02010H$ is accessed because $DS = 0100H$, $BX = 1000H$, and $DI = 0010H$.



Explain $MOV AX, [BX+SI+100H]$. Note $DS = 1000H$, $BX = 0020H$, $SI = 0010H$

The source address.

$$DS * 10 + [BX + SI + 100H]$$

$$= 10000H + 0180H$$

$$= 10130H$$

since AX contain 16 bit data hence, the corresponding $10131H$ location's data is also stored.

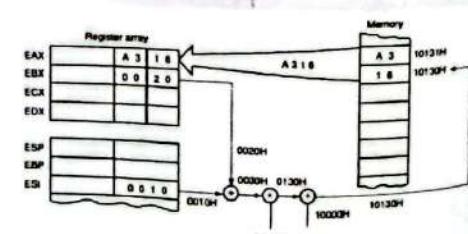


FIGURE 3-12 An example of base-relative-plus-index addressing using a $MOV AX, BX + SI + 100H$ instruction. Note: $DS = 1000H$.

Q What do you mean by microprocessor? What are the main tasks of a microprocessor?

A The microprocessor, sometimes referred to as the CPU-central processing unit, is the controlling element in a computer system. The microprocessor controls memory and I/O through a series of connections called buses.

The buses selects an I/O or memory device, transfer data between an I/O device or memory and the microprocessor, and control the I/O and memory system.

Memory and I/O are controlled through instructions that are stored in the memory and execute by the microprocessor.

The microprocessor performs three main tasks for the computer system

- 1) Data transfer between itself and the memory or I/O systems.
- 2) Simple arithmetic and logic operations and
- 3) Program flow via simple decisions

Although these are simple tasks, it is through them that the microprocessor performs virtually any series of operations or tasks.

Q What do you mean by 8-bit and 16-bit Microprocessor? How does the cache memory speedup the microprocessor?

A n-bit microprocessor means how many bit a microprocessor can process (i.e. as the operation).

8-bit means a microprocessor can process only 8 bit addition, subtraction operation etc.

16-bit can process data of 16 bit for any kind of operation to be performed.

A A CPU cache places a small amount of memory directly on the CPU. This memory is much faster than the system RAM because it operates at the CPU's speed rather than the system bus speed.

Cache memory holds frequently used instructions or data which the processor may require next and it is faster access memory than RAM.

This reduces the need for frequent slower memory retrievals from main memory, hence the processor becomes faster.

- What is buses? With a block diagram explain a Micro-processor system showing MP, ROM, RAM and peripherals with connections and direction of different buses?
- A bus is a common group of wires that interconnect components in a computer system.

The buses that interconnect the sections of a computer system transfer address, data and control information between the microprocessor and its memory and I/O systems.

There are three buses exists for this transfer of information

1. The address bus request a memory location from the memory or I/O location from I/O devices.
2. The Data bus transfer information between the micro-processor and its memory and I/O Address space.
3. The Control bus contains lines that select the memory or I/O and cause them to perform a read or write operation. The Control bus connections are

MRDC: Memory Read Control.

MWTC: Memory Write Control.

IORC: Input Output Read control.

IOWC: Input output Write control.

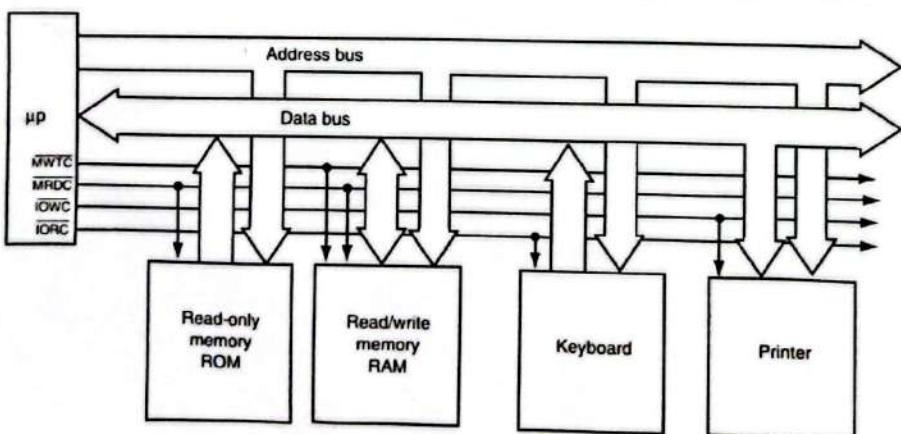


FIGURE 1-12 The block diagram of a computer system showing the address, data, and control bus structure.

Architecture of Microprocessor 8086:

The Architecture of Microprocessor 8086 is divided into two main components:

1. BIU: There are three main components in the BIU-Bus Interface Unit. It takes data from the memory interface.

(i) Instruction stream byte queue: Microprocessor 8086 have six byte registers that are used for storing pre-fetched of instructions.

(ii) Segment registers: 8086 have total 4 segment registers

ES - Extra Segment Register.

CS - Code segment register.

SS - Stack Segment Register.

DS - Data Segment Register.

IP - Instruction pointer: points the data of next instruction. So it will have address of next instruction to be executed.

(iii) Σ : BIU generates 20 bits address using segment register and offset pointer registers.

Physical Address.

$$PA = \underbrace{SR \times 10H}_{\substack{\text{Starting Address} \\ 16 \text{ bit}}} + \underbrace{OP}_{\substack{\text{20 bits} \\ \downarrow \text{offset - 16 bit}}}$$

2. EU: There are four main components in the EU-Execution unit of 8086.

(i) Control system: is used to provide control signal to all the elements in the architecture. It provides features like read and write control signal to all the elements.

(ii) General purpose Registers: There is four general purpose

Registers. AX (16 bit) $\rightarrow AH$ (8 bit) + AL (8 bit)

BX (16 bit) $\rightarrow BH$ (8 bit) + BL (8 bit)

CX (16 bit) $\rightarrow CH$ (8 bit) + CL (8 bit)

DX (16 bit) $\rightarrow DH$ (8 bit) + DL (8 bit).

SP - Stack pointer

BP - Base pointer.

SI - Source index

DI - Destination index

There is also memory pointers named SP, BP, SI & DI.

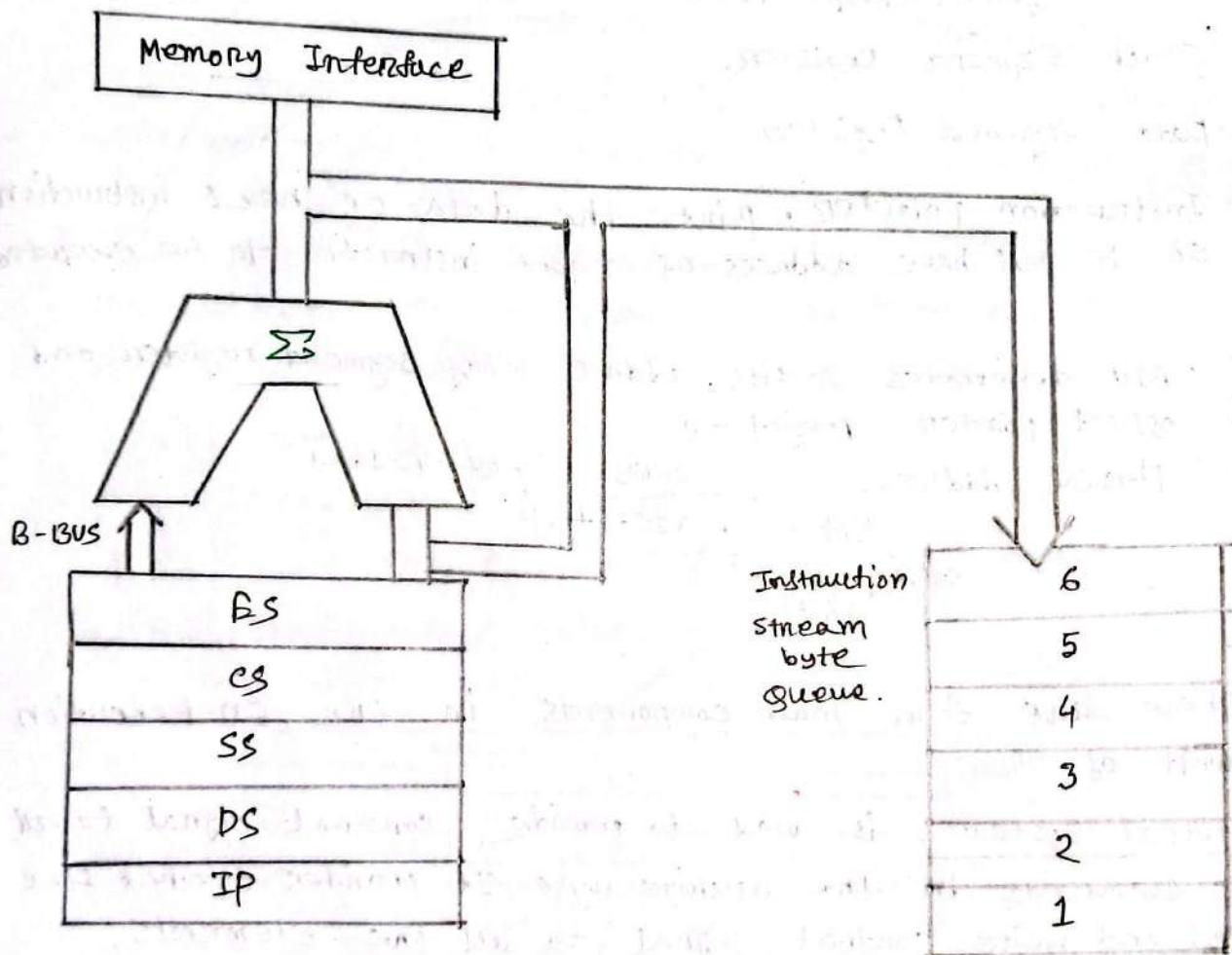
(iii) ALU: That performs all the arithmetic and logical operations. After every execution flags of 8086 will get modified when ALU task ends.

(iv) Operands and flags:

Operands in EU are the data items (register, memory or immediate values) on which instructions operate.

Flags in EU are status indicators (e.g. zero, carry, sign, overflow) set or cleared based on the result of an operation to influence program decisions.

■ Working of Bus Interface unit:



BIU is responsible for establishing communications with external peripheral devices and memory via system bus.

Main purpose of BIU is as follows.

- Fetches the instructions from memory.
- Reads/writes data from/into Input output and Memory.
- It provides the address relocation facility.

Instruction queue:

BIU prefetches six instructions byte in advance from memory. These instructions are stored in a group of high speed registers known instruction queue, which works on FIFO- first in first out order. "The process of fetching the next instruction in advance while the EU is executing the current instruction is pipelining."

Segment Registers:

Segment registers are used to store the starting address of memory segment. BIU generates 20 bits address using segment registers and offset pointer registers.

$$PA = SR \times 10H + OP$$

IP holds the address of next instruction which is to be executed next. It contains the off-set value of the next address.

The 8086 MP has the capability of addressing 1 MB memory which is divided into 16 local segments. Each segment contains 64 KB memory. But at any instant 8086 works with only four 64 KB segments.

Working of Execution unit:

The simultaneous operation of BIU and EU is possible only when the EU does not require the system bus.

EU informs BIU about where the next instruction or data to be fetched. The EU performs following functions:

- It picks up the instruction from the instruction queue of BIU.
- It decodes the instructions and then executes.
- It updates the status of flag registers.

P.T.O.

Control unit:

It controls and co-ordinates all the activities of sub units. It fetches, decodes and executes instructions. It also gives control signals like read, write etc.

EU

Control System

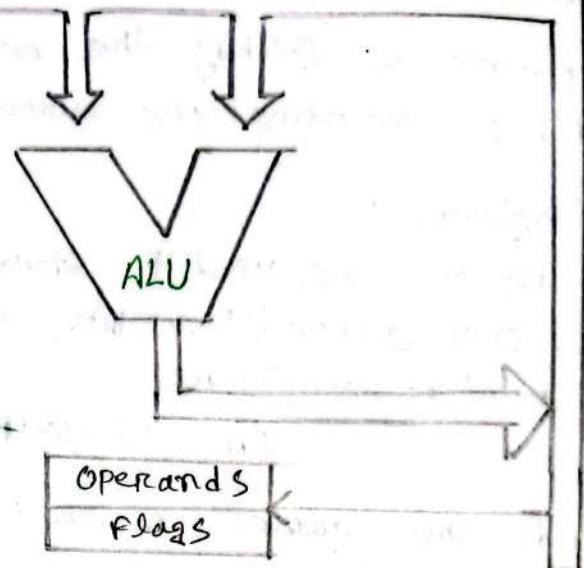
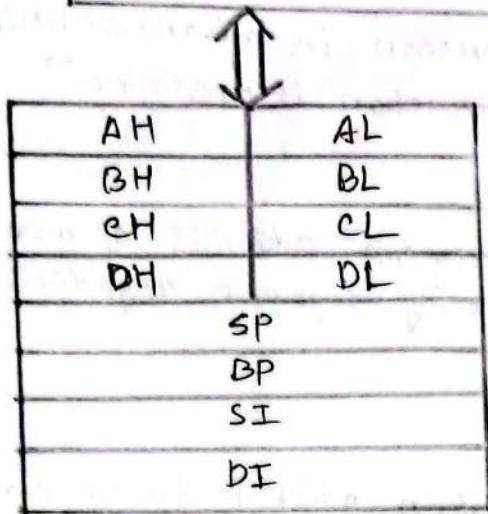


Figure: Execution Unit - EU of 8086 MP.

ALU: Performs all the arithmetic and logical operations. The results may be stored in general purpose registers or index registers. It updates flags register after every instruction.

General purpose Registers and pointers of EU:

EU have four general purpose registers. These registers are used for accessing data very fast.

- EU have Sp. Stack pointer that indicates the starting point of the stack and BP-Base pointer that is used to indicate the any other middle position of the stack.
- Index registers are SI-Source Index and DI-Destination index for data and extra segments, respectively.

Machine Language:

'Machine language is the microprocessor's native binary code, used as instructions to control its operations.'

It serves as the fundamental layer for executing tasks, with each binary instruction directly understood by the processor.

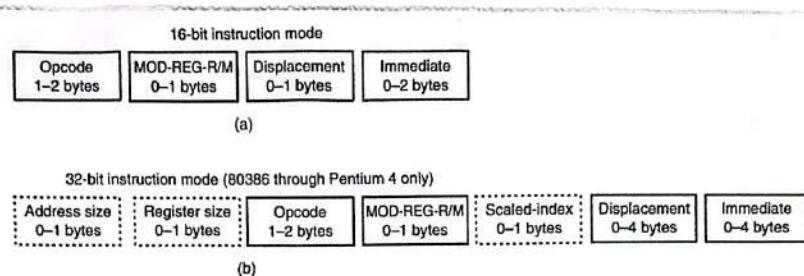


FIGURE 4-1 The formats of the 8086-Core2 instructions. (a) The 16-bit form and (b) the 32-bit form.

16 & 32 bit instruction:

The 16-bit instruction mode was introduced with the 8086 processor to support 16 digit/bits registers and memory addressing, reflecting the hardware's word size.

The 32-bit instruction mode was introduced in 80386 and later processors to enhance performance, expand addressing and enable larger data operations.

The 16-bit operation instruction model explained below:

- Opcode (1-2 bytes): Specifies the operation to perform.
- MOD-REG-R/M (0-1 bytes): Indicates addressing mode, register or memory reference.
- Displacement (0-1 bytes): Represents memory offset for effective address.
- Immediate (0-2 bytes): Contains constant data or operands.

Byte-1 & Byte-2 Machine Language Instructions.

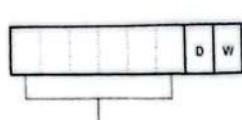


FIGURE 4-2 Byte 1 of many machine language instructions, showing the position of the D- and W-bits.

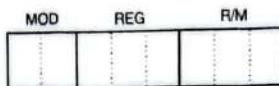


FIGURE 4-3 Byte 2 of many machine language instructions, showing the position of the MOD, REG, and R/M fields.

Byte-1: First byte of the instruction shown above figure 4-2 is known as the opcode byte (first 8 bits of machine language) are explained below.

1. Opcode (6 bits): Determines the operation (e.g. addition, move subtraction etc) of the microprocessor. For MOV instruction the opcode is '100010'.
2. D-bit (Direction): Not to be confused with the instruction mode or direction flag bit. 'D' indicates Direction flag bits.

If 'D=0' Data moves from REGn to R/M (destination in register).

If 'D=1' Data moves from R/M to REGn (destination is memory or another register). appears in instructions like MOV.

3. W-bit (Word/Byte): Indicates whether the data are a byte or a word (W).

'W'=1 operates on a word (16 or 32 bit).

'W'=0 operates on a byte (8 bit).

Byte-2: The second byte of the instruction shown in fig 4-3 is known as the MOD-REG-R/M byte (last 8 bits of the Machine Language) are explained below.

1. MOD (2 bits): specifies addressing mode (direct, indirect or with displacement). where 11 indicates register addressing and others (00, 01, 10) are memory addressing with optional displacement. (Table 4-1).
2. REG (3 bits): Identifies the register for the instruction based on Table 4-3.
3. R/M (3 bits): Selects a register or memory location based on 'MOD'. For MOD = '11' R/M based on TABLE 4-3 R/M register is identified since MOD = '11' indicates register addressing. for other MOD values (00, 01, 10) based on TABLE 4-4. given below.

MOD	Function
00	No displacement
01	8-bit sign-extended displacement
10	16-bit signed displacement
11	R/M is a register

TABLE 4-1 MOD field for the 16-bit instruction mode.

Code	W = 0 (Byte)	W = 1 (Word)	W = 1 (Doubleword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

TABLE 4-3 REG and R/M (when) MOD = 11 assignments.

R/M Code	Addressing Mode
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]*
111	DS:[BX]

*Note: See text section, Special Addressing Mode.

TABLE 4-4 16-bit R/M memory-addressing modes.

For the instruction `MOV DL, [DI]` convert to its machine language form.

For `MOV` instruction the opcode is '100010', D=1 indicates 'Transfer to register' W=0 means 'Byte'

MOD = '00' From the Table 4-1 MOD = '00' means no displacement

R/M = '010' means DL (Table 4-3), for R/M = '01'; since MOD = '00' table 4-4 is used R/M = '101' means R/M = DS:[DI]. {fig: 4-5}.

Opcode	D	W
1 0 0 0 0 1 0 1 0		

MOD	REG	R/M
0 0	0 1 0	1 0 1

Opcode = MOV
D = Transfer to register (REG)
W = Byte
MOD = No displacement
REG = DL
R/M = DS:[DI]

FIGURE 4-5 A `MOV DL,[DI]` instruction converted to its machine language form.

For the instruction `MOV [1000H], DL` convert to its machine language form.

For `MOV` opcode is 100010

W=0 means Byte

D=0 means Transfer from Register.

R/M = '010' means DL (Table 4-3).

R/M = '110' means DS:[BP]

(special addressing mode) using (BP)

add Displacement from Byte 4 = 1000H.

Opcode	D	W
1 0 0 0 0 1 0 0 0		

MOD	REG	R/M
0 0 0	1 0 1	1 1 0

Displacement—low							
0 0 0 0 0 0 0 0							

Displacement—high							
0 0 0 1 0 0 0 0							

Byte 1
Byte 2
Byte 3
Byte 4

Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 1000H

FIGURE 4-6 The `MOV [1000H],DL` instruction uses the special addressing mode.

For the instruction `MOV [BP], DL` convert to its machine language form.

For `MOV` opcode is 100010

D=0 means transfer from Register.

W=0 means Byte

REG = DL (Table 4-3).

R/M = DS:[BP]

MOD = because R/M is BP (special addressing) using BP adds

Displacement = 00H.

Opcode	D	W
1 0 0 0 0 1 0 0 0		

MOD	REG	R/M
0 1	0 1 0	1 1 0

8-bit displacement							
0 0 0 0 0 0 0 0							

Byte 3

Opcode = MOV
D = Transfer from register (REG)
W = Byte
MOD = because R/M is [BP] (special addressing)
REG = DL
R/M = DS:[BP]
Displacement = 00H

FIGURE 4-7 The `MOV [BP],DL` instruction converted to binary machine language.

PUSH

The push operation stores data from a register, memory, or immediate value onto the stack.

It decrements the stack pointer (SP) and moves the data to the stack in a specified order. For a 16-bit processor, 2 bytes are pushed, while a 32-bit processor pushes 4 bytes. Special instructions like **PUSHA** and **PUSHAD** save all general-purpose registers, requiring 16 or 32-bit stack space, respectively. In 64-bit mode, 8 bytes are used per push. This operation is useful for saving processor states.

■ Graphically explain the effect of **PUSH AX** instruction on ESP and stack memory locations.

■ Referring to the fig 4-13.

1. The **PUSH AX** instruction stores the contents of the **AX** register onto the stack.
2. The stack pointer SP decremented by 2 to **07FEH**.
3. The value in **AX (6AB3H)** is stored at memory locations **37FFH** and **37FEH** within the stack segment.

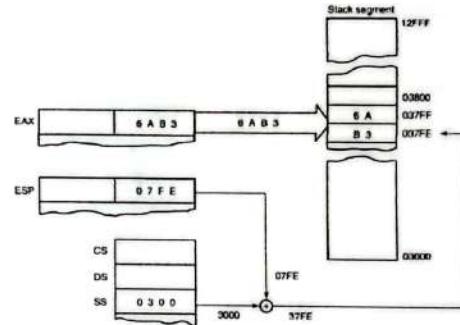


FIGURE 4-13 The effect of the **PUSH AX** instruction on ESP and stack memory locations 37FFH and 37FEH. This instruction is shown at the point after execution.

■ Graphically explain the stack cycling nature for **PUSH CX** instruction.

■ Referring to the fig 4-16.

1. The **PUSH CX** instruction stores the contents of the **CX** register onto the stack.
2. The stack pointer decrements from **10000H** to **FFFFEH**, illustrating the stack cycling nature.
3. The value (**A037H**) is written on the stack at **1FFFH & 1FFFEH**.

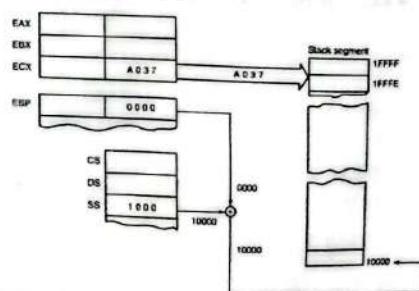


FIGURE 4-16 The **PUSH CX** instruction, showing the cyclical nature of the stack segment. The instruction is shown just before execution, to illustrate that the stack bottom is contiguous to the top.

POP

The pop operation retrieves data from the stack and places it into a specific register, segment register or memory location. (i.e. Inverse of PUSH operation).

Special instructions like POPA and POPAD restore all registers from the stack, requiring 16 or 32 bytes. POPF and POPFD retrieve data for flag registers. For each pop 2 or 4 bytes are moved based on the processor. This operation is essential for restoring processor states.

■ Show Diagrammatically the operation of POP BX instruction.

■ Referring to the fig 4-16.

1. The POP BX instruction retrieves the last value stored on the stack into the BX Register. (inverse of PUSH).
2. The stack pointer () is incremented by 2 to 1008H.
3. The value previously stored (392FH) is moved into the BX register.

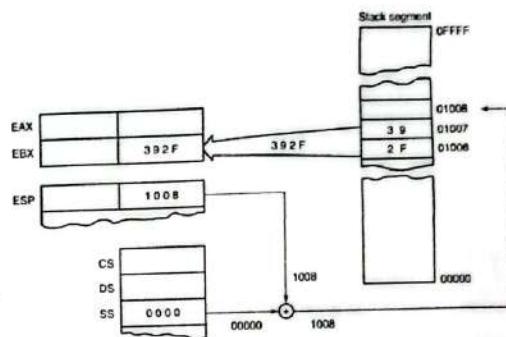


FIGURE 4-15 The POP BX instruction, showing how data are removed from the stack. This instruction is shown after execution.

■ Show Diagrammatically the operation of LDS BX,[DI] instruction.

■ The instruction LDS BX,[DI] (Load pointer using DS) loads a segment register and a general purpose register with values from memory.

Operation:

- The instruction uses the DS:DI pointer to read a 4-byte value from memory.
- The first two bytes (11000H & 11001H) are loaded into the BX register as 7A12H.
- Next two bytes (11002H and 11003H) are loaded into the register as 1000H.

Purpose:

- Used to initialize BX and DS with data for further operations involving segmented memory.

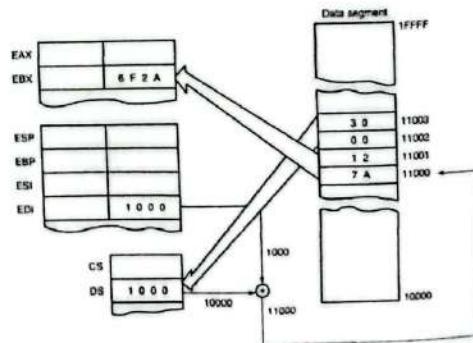


FIGURE 4-17 The LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.

Program Invisible Registers:

There are internal microprocessor registers (e.g. segment descriptor) not directly accessible by programmers. They store crucial control information, such as base addresses, limit values and access rights for memory segments.

Control of ALU Output: Program Invisible Registers manage the memory segmentation process and ensure correct addressing of operands. By doing so they direct the ALU to operate on the correct data, influencing the accuracy and security of arithmetic and logic operations in protected or virtual memory environments.

Racing Conditions in Microprocessors:

Racing Condition occurs when two or more processes or instructions attempt to access or modify the same resource (like memory or registers) simultaneously, leading to unpredictable behaviour due to improper synchronization.

How to avoid Racing conditions:

- Use locks: Use program invisible Register to store one O/P and avoid racing conditions.

Arithmetic & Logical Instructions:

These are instructions that perform operations on data in a microprocessor.

Arithmetic means dealing with numeric data the result is also numeric, stored in registers. Include Addition, subtraction, Multiplication, Division, increment and decrement operation.

Logical instructions has no result just has decision values, stored in flags. Include bitwise AND, OR, X-OR, NOT and Comparison operations.

Addition, subtraction and comparison is done using one circuit. Explain with example.

In 8086 microprocessor addition, subtraction and comparison operations share the same ALU circuitry by leveraging binary arithmetic. The efficient use of shared circuitry minimizes hardware complexity.

1. Addition: For binary addition the ALU adds corresponding bits, considering carry.

$$\begin{array}{r} 1010 \text{ (10)} \\ + 0110 \text{ (6)} \\ \hline 1000 \text{ (6)} \end{array}$$

2. Subtraction: Performed using 2's complement. Negate the second operand (6) \rightarrow (1010) and then adding.

$$\begin{array}{r} 1010 \text{ (10)} \\ + 1010 \text{ (-6)} \\ \hline 0100 \text{ (4)} \end{array}$$

3. Comparison: Done via subtraction but the result is not stored in registers. Instead flags are set.

From the sub-result

0100

$ZF=0; CF=0; SF=0$
indicates $10 > 6$

Addition:

The 8086 microprocessor performs addition using ADD instruction, which adds two 8 bits or 16 bit operands and stores the result in a destination. The operands can be registers, memory locations or immediate values, except for segment registers.

For example ADD AX, BX adds the contents of BX to AX and the result is stored in AX.

If a carry occurs, it is flagged in the Carry flag (CF). The ADC instruction performs addition with carry for multi-word operations, useful for larger numbers.

Immediate addition is employed whenever constant or known data are added. In example 5-2 DL is first loaded with 12H by using an immediate move instruction. Next, 33H is added to the 12H in DL by an immediate addition instruction. After the addition the result 45H moves to DL.

Example 5-2
MOV DL, 12H
ADD DL, 33H

and the flags change as follows.

$Z=0$ (result not zero)

$C=0$ (no carry)

$A=0$ (no half carry)

$S=0$ (result positive)

$P=0$ (odd parity)

$O=0$ (no overflow)

Increment Addition INC adds 1 to a register or a memory location except segment register.

Example:

$INC BL \quad BL = BL + 1$

$INC SP \quad SP = SP + 1$

Addition with Carry instruction ADC adds the bit in the carry flag C to the Operand data. A carry occurs if the sum is greater than $FFFFH$.

Example:

$ADC AL, AH \quad AL = AL + AH + \text{carry}$

$ADC CX, BX \quad CX = CX + BX + \text{carry}$

Explain the instruction $ADD AX, CX$ $ADC BX, DX$.

Fig 5-1 illustrates this addition, so that the placement and function of the carry flag can be understood.

The $ADD AX, CX$ instruction first

add the two contents of BX and CX and saves carry flag as '0' if no carry occurs, And '1' if carry occurs.

The $ADC BX, DX$ instruction adds the '1' or the '0' in the carry flag to the most significant 16 bits of the result. This program adds $BX-AX$ to $DX-CX$ with the sum appearing in $BX-AX$.

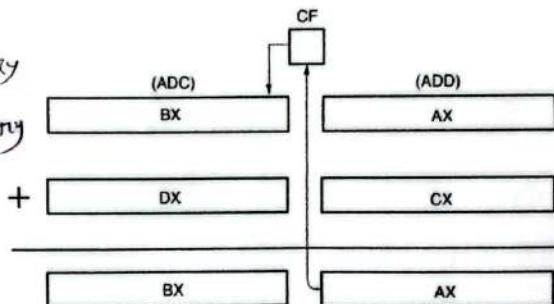


FIGURE 5-1 Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

Subtraction:

The 8086 microprocessor performs subtraction using the SUB instruction, which subtracts the source operand from the destination operand. Operands can be 8-bit or 16-bit and may include registers, memory locations or immediate values.

Example:

SUB AX, BX Subtracts the value of BX from AX. Storing the result in AX.

Flags like zero (ZF), carry (CF) and sign (SF) update based on the result.

Immediate subtraction: allows immediate operands for the subtraction of constant data.

Example 5-10

MOV CH, 22H
SUB CH, 44H

Presents a short sequence of instructions that subtract 44H from 22H. The data 22H is loaded to CH via immediate movement.

Then the sub instruction subtracts 44H from the 22H. After the subtraction the difference (0DEH) moves into the CH register. The flags change as follows for this subtraction.

Z = 0 (result not zero)

C = 1 (borrow)

A = 1 (half borrow)

C = 1 (result negative)

P = 1 (even parity)

O = 0 (no overflow)

Both carrying flags (C & A) hold borrows after a subtraction instead of carries.

Decrement Subtraction: Decrement subtraction DEC subtracts '1' from a register or the contents of a memory location.

Example: DEC BH BH = BH - 1
 DEC CX CX = CX - 1

Subtraction with Borrow: A SBB instruction functions as a regular subtraction, except that the carry flag C, which holds the borrow, also subtracts from the difference.

Explain the instructions SUB AX, DI
 SBB BX, SI

fig 5-2 shows how the borrow propagates through the carry flag C. The example uses the SUB instruction to subtract DI from AX, then uses SBB to subtract with borrow SI from BX.

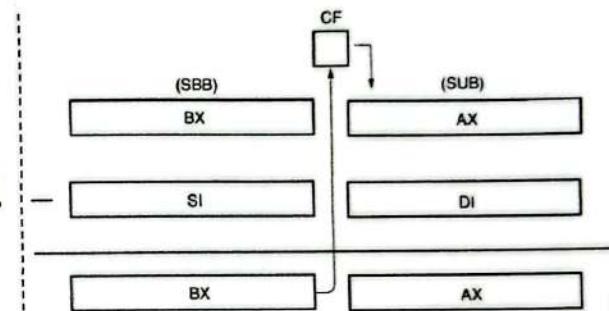


FIGURE 5-2 Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.

Comparison:

The comparison instruction CMP is a subtraction that changes only the flag bits, the destination operand never changes.

A comparison is useful for checking the entire contents of a register or a memory location against another value. A CMP is normally followed by a conditional jump instruction which tests the condition of a flag bits.

Suppose comparing A & B. if.

Sign	Zero	Result
0	0	A > B
0	1	A = B
1	0	A < B
1	1	Logically zero & -ve not possible

Multiplication: 8086 supports unsigned (**MUL**) and signed (**IMUL**) multiplication.

For 8bit $AL * CL$ result in 16 bit AX and for 16 bit operand in AX results in $AX * DX$ 32 bit EAX .

Overflow occurs if result exceeds destination size, so carry flag **CF** and overflow flag **OF** are set if upper half AH or $DX \neq 0$.

To avoid overflow we always check **CF/OF** and use wide registers or mask high bits if not needed. Use **CWD** convert Byte to Word or **CWD** convert word to doubleword before signed multiplication.

Modern microprocessors handle overflow using subroutines that detect and manage excess value beyond register limits. These subroutine checks status flags **CF** or **OF**, and redirect execution to code that stores extended results or adjusts calculations.

This method is better because it ensures data integrity, supports arbitrary precision, and avoids system crashes, especially in system critical applications like finance, encryption or scientific computing. Using subroutine also keeps the main code cleaner, modular and easier to debug or upgrade.

TABLE 5-8 Example 8-bit multiplication instructions.

Assembly Language	Operation
MUL CL	AL is multiplied by CL; the unsigned product is in AX
MUL DH	AL is multiplied by DH; the signed product is in AX
MUL BYTE PTR[SI]	AL is multiplied by the byte contents of the data segment memory location addressed by SI; the signed product is in AX
MUL TEMP	AL is multiplied by the byte contents of data segment memory location TEMP; the unsigned product is in AX

TABLE 5-9 Example 16-bit multiplication instructions.

Assembly Language	Operation
MUL CX	AX is multiplied by CX; the unsigned product is in DX-AX
IMUL DI	AX is multiplied by DI; the signed product is in DX-AX
MUL WORD PTR[SI]	AX is multiplied by the word contents of the data segment memory location addressed by SI; the unsigned product is in DX-AX

DIVISION:
8086 microprocessor uses DIV unsigned and IDIV signed instruction to perform division.

For 8 bit division, dividend in AX and divisor in AL register.
Result: quotient in AL , remainder in AH

For 16 bit division, dividend in DX:AX , divisor in 16 bit register
Result: quotient in AX , remainder in DX .

Divided by zero or overflow triggers INT 0 - divide error.
No carry and overflow is used hardware handles exceptions.

TABLE 5-12 Example 8-bit division instructions.

Assembly Language	Operation
DIV CL	AX is divided by CL, the unsigned quotient is in AL and the unsigned remainder is in AH
IDIV BL	AX is divided by BL, the signed quotient is in AL and the signed remainder is in AH
DIV BYTE PTR[BP]	AX is divided by the byte contents of the stack segment memory location addressed by BP, the unsigned quotient is in AL and the unsigned remainder is in AH

TABLE 5-13 Example 16-bit division instructions.

Assembly Language	Operation
DIV CX	DX-AX is divided by CX, the unsigned quotient is AX and the unsigned remainder is in DX
IDIV SI	DX-AX is divided by SI, the signed quotient is in AX and the signed remainder is in DX
DIV NUMB	DX-AX is divided by the word contents of data segment memory NUMB; the unsigned quotient is in AX and the unsigned remainder is in DX

Basic Logic Instruction:

Basic logic instructions in 8086 microprocessor perform bitwise operations directly on binary data. These instructions are essential for data masking, setting, clearing, toggling bits and testing conditions without affecting values. They play a vital role in low-level programming, decision-making, and hardware-level operations, while also updating processor's flags based on the result.

AND: Performing bitwise AND on two operands. Resulting bit is 1 only if both bits are 1. The AND operation clears bits of binary number.
The task of clearing a bit in a binary number is called masking.
AND operation is commonly used for masking. Does not affect the Carry Flag but affects ZF, SF, PF based on the result.

OR: Performs a bitwise OR in two operands, it performs logical addition and is often called the Inclusive-OR function. It sets bits to 1 where either operand has a 1. Used to set specific bits in a register or memory location without changing others. Does not store any carry but updates other flags like Zero (ZF), sign (SF), Parity (PF).

X-OR: Performs bitwise exclusive-OR, result is 1 if the bits differ, useful for toggling bits or clearing a register (e.g X-OR AX, AX=0) Like AND/OR it affects ZF SF PF but not CF. The difference from OR operation is that, 1,1 condition for X-OR produces '0' at the output while '1' for the OR operation.

TEST: The test operation performs AND operation, the difference is that AND instruction changes the destination operand, whereas the TEST doesn't. That it's performing an AND operation without storing the result. Used to test if specific bits are set. Unlike 'CMP' instruction TEST only single bit. This instruction is followed by either JZ - jump if zero or JNZ - jump if not zero instructions. It updates flags like ZF, SF, PF but leaves the original operand unchanged - ideal for condition checks.

NOT / NEG: NOT performs bitwise inversion '1 becomes 0 and 0 becomes 1', does not affect any flag, is used to flipping bits. (i.e. does one's complement).

Negate NEG finds the two's complement (i.e. negates the operand). It affects flags like carry (CF), zero (ZF), sign (SF) overflow (OF).

Assembly Instruction	Explanation
AND AL, 0Fh AND AX, 00FFh	8-bit: Bitwise AND between AL and 0Fh (clears upper nibble) 16-bit: Clears upper byte of AX, keeps lower byte
OR AL, OF0h OR AX, F0F0h	8-bit: Sets upper nibble of AL using OR 16-bit: Sets specific bits of AX using OR mask
XOR AL, AL XOR AX, AX	8-bit: Clears AL (AL = 0), common zeroing trick 16-bit: Clears AX (AX = 0)
TEST AL, 0Fh TEST AX, 8000h	8-bit: Tests lower 4 bits of AL (sets flags, no change in AL) 16-bit: Tests MSB of AX (checks sign bit)
NOT AL NOT AX	8-bit: Inverts all bits of AL 16-bit: Inverts all bits of AX
NEG AL NEG AX	8-bit: Two's complement of AL (AL = -AL) 16-bit: Two's complement of AX (AX = -AX)

SHIFT:
Shift instruction position or move numbers to the left or right within a register or memory location. They also perform simple arithmetic such as multiplication by power of 2^{+n} (left shift) and division by power of 2^{-n} (right shift).

The microprocessor's instruction sets contains four different shift instructions. Two are logical shifts and two are arithmetic shifts (i,j).

The arithmetic Shift left and logical shift left are identical.

But the Right shift are different.

Logical shift operations function with unsigned numbers, and arithmetic shifts function with signed numbers.

Logical shift multiply or divide unsigned data, whereas arithmetic does with signed data. A shift left always multiplied by 2 for each bit position and at right shift it is divided by 2 for each bit position.

For example, value $0101B \approx (5)_{10}$

Shift left by 1 yields $1010B \approx (10)_{10}$

and, (0010B) shift right yields $(0101B) \approx (5)_{10}$

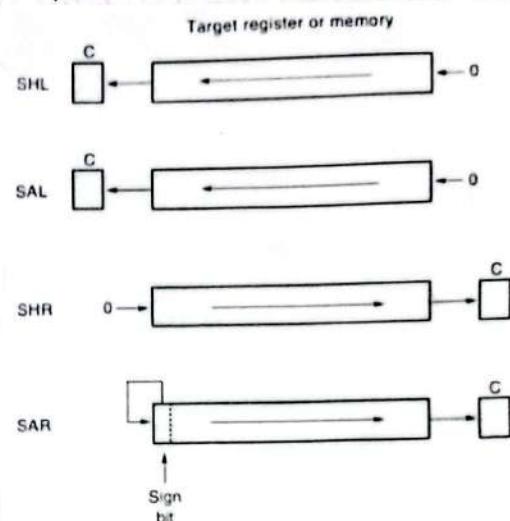


FIGURE 5-9 The shift instructions showing the operation and direction of the shift.

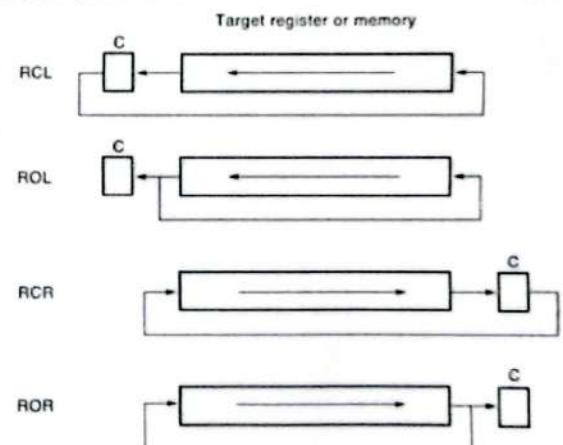


FIGURE 5-10 The rotate Target register or memory instructions showing the direction and operation of each rotate.

ROTATE:

Rotate instructions position binary data by rotating the information in a register or memory location, either from one end to another or through the carry flag.

Rotate Left (ROL): Bits are shifted left, the MSB moves to the LSB.

Example:

$10110000B \rightarrow ROL \rightarrow 01010001B$

Rotate Right (ROR): Bits are shifted right, the LSB moves to the MSB position.

Example:

$10110000B \rightarrow ROR \rightarrow 01011000B$

Rotate Through Carry: (RCL / RCR): similar to ROL / ROR but includes the carry flag (CF) in rotation.

Example: (RCL)

$CF = 1; AL = \underline{1}0000000B \rightarrow RCL \rightarrow 00000001B, CF = 1$

Used in encryption, data masking and low level bit manipulation.

JUMP:

Jump instructions are used to change the flow of execution in a program. In 8086, they help in decision making and looping structures by transferring control to another part of the program. These instructions can be either (i) Unconditional (always jump) or (ii) conditional (jump only if a specific condition is met based on flags like zero, sign, carry etc.)

(i) Unconditional jump:

- JMP label: Transfers control unconditionally to the specific label.
 - CALL label: Calls a subroutine (stores return address).
- Used in loops, subroutines, and skipping blocks of code.

(ii) Conditional jump:

Based on flag status, allows decision-making in code.

TABLE 6-1 Conditional jump instructions.

Assembly Language	Tested Condition	Operation
JA For unsigned operands	$Z = 0$ and $C = 0$	Jump if above
JAE	$C = 0$	Jump if above or equal
JB	$C = 1$	Jump if below
JBE	$Z = 1$ or $C = 1$	Jump if below or equal
JC	$C = 1$	Jump if carry
JE or JZ	$Z = 1$	Jump if equal or jump if zero
JG For signed operands	$Z = 0$ and $S = 0$	Jump if greater than
JGE	$S = 0$	Jump if greater than or equal
JL	$S \neq 0$	Jump if less than
JLE	$Z = 1$ or $S \neq 0$	Jump if less than or equal
JNC	$C = 0$	Jump if no carry
JNE or JNZ	$Z = 0$	Jump if not equal or jump if not zero
JNO	$O = 0$	Jump if no overflow
JNS	$S = 0$	Jump if no sign (positive)

Types of Unconditional jump:

1. Short Jump (8-bit displacement):

- Range: -128 to +127; 1 byte signed number from current IP.
- Uses 1 byte signed offset.
- Commonly used in small loops.
- Efficient and faster due to smaller instruction size.

2. Near Jump (16-bit displacement):

- Range: +32K bytes within current code segment.
- Uses 2 byte offset to update the IP.
- CS - code segment remains unchanged.
- Used when branching within the same segment but beyond 8-bit range.

3. Far Jump (Intersegment jump):

- Transfer control to a different code segment.
- Updates both CS & IP 4-byte address: 2 byte for IP & 2 byte for CS.
- Used for subroutine calls or branches across segments.
- Slightly slower due to segment switching.

Short & Near jumps are intra-segment because they only changes IP within the current CS.

Far jump changes both IP & CS. Allowing jumps in different segments.

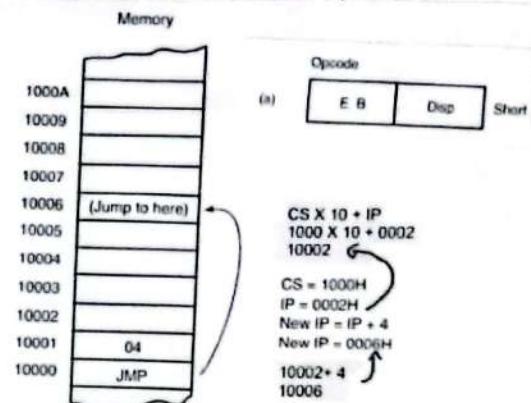


FIGURE 6-2 A short jump to four memory locations beyond the address of the next instruction.

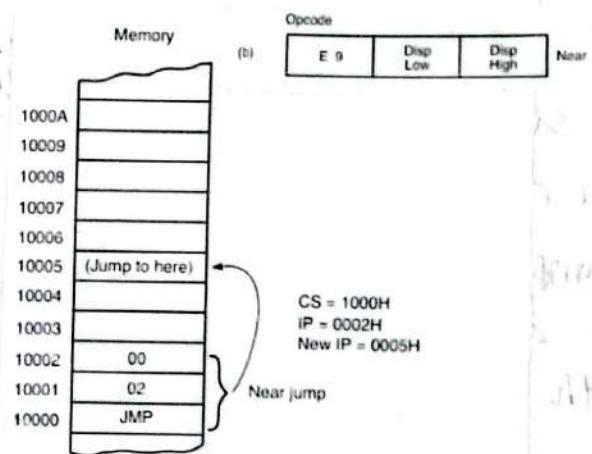


FIGURE 6-3 A near jump that adds the displacement (0002H) to the contents of IP.

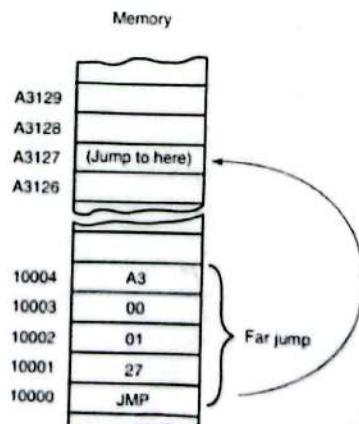
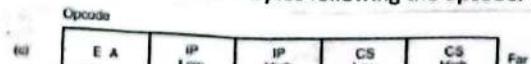


FIGURE 6-4 A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.



Procedures:

Procedures (subroutine, method or functions) in 8086 microprocessor are subprograms or blocks of code designed to perform specific tasks. They help in modular programming by allowing code reuse and better structure.

A Procedure is invoked using the **CALL** instruction and execution returns to the calling program with the **RET** instruction.

CALL:

call is used to invoke a procedure. It saves the return address on the stack before transferring control.

- **Near Call:**

Transferring control to a procedure within the same code segment, only the **IP** is saved and modified.

- **Far Call:**

Transfers control to a procedure in a different code segment, both **CS** & **IP** are saved and changed.

RET:

Ret is used to return from a procedure to the calling program. It retrieves the saved address from the stack.

- In a Near Return only the **IP** is restored. (figure 6-8), on the next page.

- In a Far Return both **CS** and **IP** are restored.

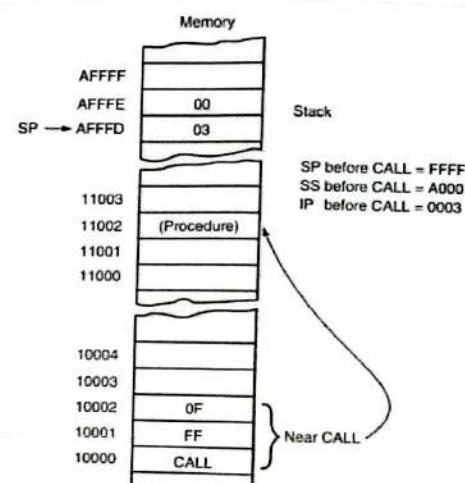


FIGURE 6-6 The effect of a near CALL on the stack and the instruction pointer

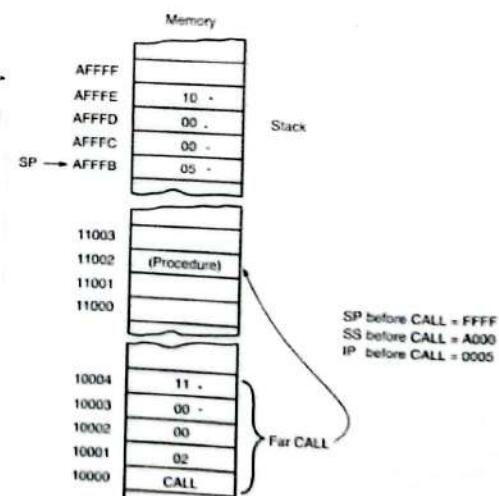


FIGURE 6-7 The effect of a far CALL instruction

Interrupt:

An interrupt is a mechanism by which a microprocessor's normal execution flow is paused to respond to urgent tasks or events. It temporarily transfers control to a special routine called an Interrupt Service Routine **ISR**. And resumes the previous task after handling the interrupt.

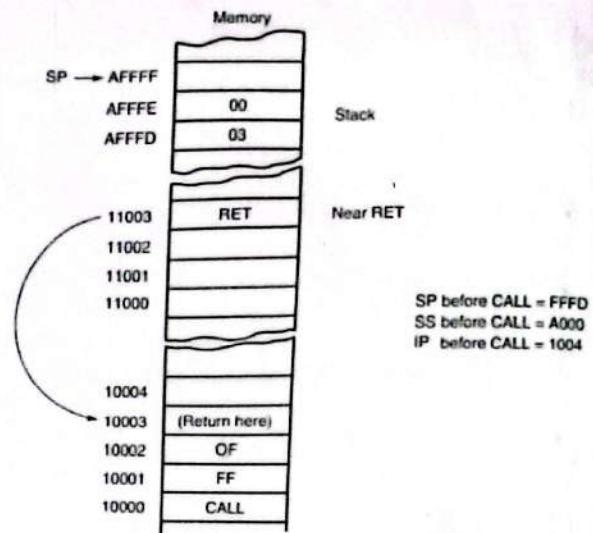


FIGURE 6-8 The effect of a near return instruction on the stack and instruction pointer.

• IRET - Interrupt Return:

- Used to return from ISR in real mode.
- Restores IP, CS and Flags from the stack.

• IRETD - Interrupt Return Doubleword:

- Used in 32-bit protected Mode.
- Restores EIP, CS & EFLAGS.



FIGURE 6-9 Interrupts in a typical personal computer.

Interrupt improves response time, support multitasking and are vital for handling real time hardware/software events efficiently.

INTn: Triggers Software interrupt number n (e.g. INT 21H for DOS).

INTO: Interrupt if overflow flag is set. (OF set).

INT3: Special 1-byte breakpoint interrupt, mainly for debugging.

WAIT - (Wait for Test Input)

- Pauses micro processor's execution until the TEST pin is low.
- Used to synchronize with external Hardware (e.g. co-processors).
- Ensures proper timing before continuing.
- No flag is affected
- Common in math co-processor interfacing.

HLT: (Halt)

- Stops CPU execution until an external interrupt or reset occurs.
- Puts CPU in low power idle state.
- Used in OS idle loops.
- Requires interrupts to resume operation.
- Doesn't affect any flags or registers.

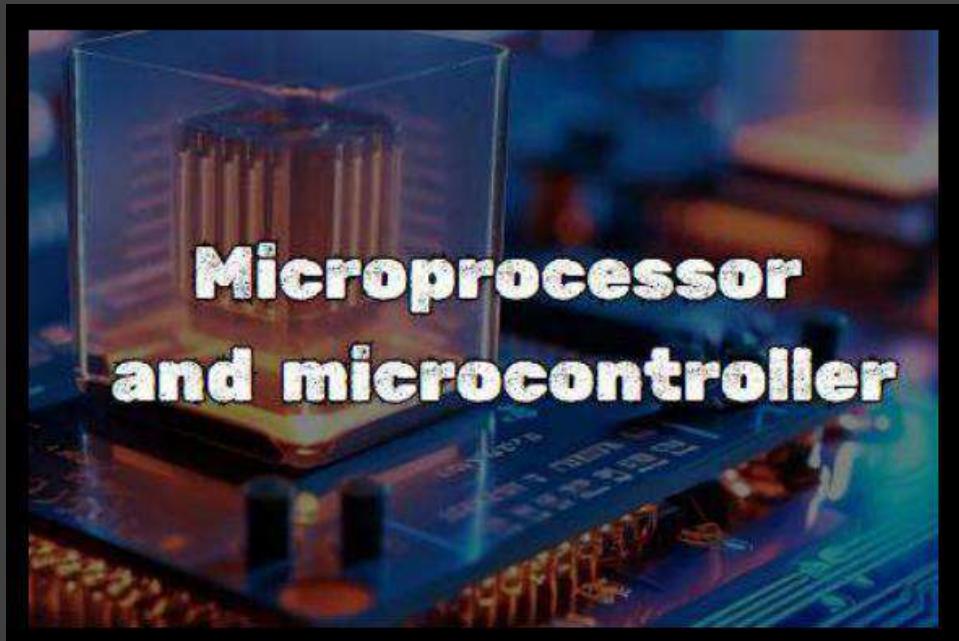
NOP: (No operation)

- CPU does nothing for 1 CLK cycle.
- Used for timing adjustment and instruction alignment.
- Common in delay loops and debugging.
- A single byte instruction
- Safe to insert without altering program flow.

LEA: (Load Effective address).

- Loads the address not the value of a memory operand into a register.
- Syntax (LEA destination, source) \rightarrow LEA BX, [SI+10]
- Used for pointer arithmetic, accessing array elements, structures etc.
- Does not access memory, only computes the effective address.

Operation	Instruction	Example
Move	MOV	MOV AX, BX (AX = BX)
Push	PUSH	PUSH AX (AX value to stack)
Pop	POP	POP BX (Stack value to BX)
Load Segment	LDS , LES	LDS AX, [1234H] (AX = DS:1234)
Load Effective Address	LEA	LEA SI, [BX+4] (SI = BX+4)
Add	ADD , ADC	ADD AX, BX (AX += BX)
Subtract	SUB , SBB	SUB CX, DX (CX -= DX)
Comparison	CMP	CMP AX, BX (Compare AX, BX)
Multiplication	MUL , IMUL	MUL BX (AX = AL × BX)
Division	DIV , IDIV	DIV CX (AX ÷ CX)



SECTION -A

Previous year Questions

Solved using ChatGPT

1. What is a Turing machine? Explain how it resembles a modern microprocessor.

- A Turing Machine is a theoretical computing model invented by Alan Turing, which can perform computations using a tape, a head, and a set of rules.
- It can **read, write, and move** along the tape, simulating any algorithm.
- Like modern microprocessors, it processes input, stores memory, and gives output — forming the basis of how **today's CPUs** execute instructions and manage memory.

2. Explain the operation of the following instructions:

i) `MOV CH, CL` :

- Copies the contents of **CL register** into **CH register**.
- It's a register-to-register data transfer.

ii) `MOV AL, NUMBER` :

- Moves a **constant value** (**NUMBER**) directly into **AL register**.
- Example: `MOV AL, 25H` → **AL = 25H**

iii) `POPF` :

- Pops the **top value of the stack** into the **FLAGS register**.
- Restores the state of the flags.

iv) `PUSH 1234H` :

- Pushes the **immediate value 1234H** onto the stack.

3. Explain why serial communication is better than parallel communication.

- **Serial communication** sends data **bit by bit** over **one line**, while **parallel** sends **multiple bits at once** over **multiple lines**.
- **Serial** is **better for long distances** as it reduces signal interference and cost.
- **Less cabling, cheaper, and more reliable** for modern communication like USB, Bluetooth.
- **Parallel** is faster **only over short distances**, but suffers from timing issues.

4. Why should you clear T and I flag bits in software interrupts? Write down the instructions.

- **T (Trap) Flag** enables single-step debugging.
- **I (Interrupt) Flag** enables hardware interrupts.
- During **software interrupt execution**, these should be **cleared** to prevent conflicts or accidental interruptions.

Instructions to clear them:

- `CLI` – Clears **Interrupt Flag (IF)**
- `CLC` – Clears **Carry Flag** (not needed here but often grouped)
- `PUSHF` followed by manipulation can also clear **Trap Flag (TF)** if needed.

5. Compare the maskable and non-maskable interrupt showing examples for both hardware and software interrupt.

- ◆ **Maskable Interrupts (INTR):**

- These **can be disabled or "masked"** using the `CLI` (Clear Interrupt) instruction.
- The CPU checks for these interrupts only if the **Interrupt Flag (IF)** is set (i.e., $IF = 1$).
- Used for **non-urgent** events like keyboard inputs or disk I/O.
- **Example (Hardware):** Keyboard interrupt (IRQ1)
- **Example (Software):** `INT 21H` (DOS function call)

- ◆ **Non-Maskable Interrupts (NMI):**

- These **cannot be disabled**; they are **always recognized** by the processor.
- Typically used for **emergency or critical situations** (e.g., hardware failure, power loss).
- Always has **higher priority** than maskable interrupts.
- **Example (Hardware):** Parity error detection, power failure detection
- **Example (Software):** Not commonly used in software; usually triggered by hardware directly.

6. For each of the following statements, state the addressing modes of the source and destination operands.

i) `MOV BX, 1000H`

- **Destination:** Register (BX)
- **Source:** Immediate

ii) `MOV ALPHA, AX`

- **Destination:** Direct (ALPHA is a memory label)
- **Source:** Register (AX)

iii) `ADD CX, +2[BX]`

- **Destination:** Register (CX)
- **Source:** Based with displacement (Effective address = $BX + 2$)

iv) `MOV [BX]+ALPHA, CX`

- **Destination:** Based with displacement ($BX + ALPHA$)
- **Source:** Register (CX)

7. What is the effect of `XOR AX, AX` instruction on the register AX? Why is it preferred over alternative ways of achieving the same result?

- `XOR AX, AX` clears the AX register by setting all its bits to zero.
- It is faster and smaller in size than `MOV AX, 0`.
- Does not affect memory and uses less CPU clock cycles, so it's more efficient and commonly used in assembly for clearing registers.

8. i) A memory location has a physical address 04A37H. If the segment number is 40FFH, compute the offset address. Also, if the offset address is 123BH, compute the segment address.

Formula:

$$\text{Physical Address} = (\text{Segment} \times 10\text{H}) + \text{Offset}$$

a) Offset = $04A37H - (40FFH \times 10H) = 04A37H - 40FF0H = 0427H$

b) Segment = $(\text{Physical Address} - \text{Offset}) / 10H = (04A37H - 123BH) / 10H = 0377CH / 10H = 377CH$

8. ii) Determine the physical address of a memory location given by 0A51:CD90H.

$$\text{Physical Address} = (0A51H \times 10H) + CD90H = 0A510H + CD90H = 172A0H$$

9. In the instruction "RET pop-value", what is the significance of 'pop-value'?

- 'pop-value' specifies how many bytes to remove from the stack (used for cleaning parameters passed to a procedure).
- After returning, it adjusts the stack pointer (SP) by popping the return address and also the specified number of extra bytes.
- Used in far and near procedure returns with parameters.

10. Name 5 different addressing modes with examples.

1. Immediate Addressing

→ Operand is part of the instruction.

Example: `MOV AL, 05H`

2. Direct Addressing

→ Address of operand is given directly.

Example: `MOV AX, [1234H]`

3. Register Addressing

→ Operand is in a register.

Example: `MOV AX, BX`

4. Register Indirect Addressing

→ Address is in register (like BX, SI, DI).

Example: `MOV AL, [BX]`

5. Base Plus Index Addressing

→ Combines base and index registers.

Example: `MOV AX, [BX+SI]`

11. Differentiate between MOV and LEA instruction

- **MOV:** Transfers actual data from source to destination.
- **LEA:** Loads the **effective address** (not the data) into a register.

12. Which logic instruction can be used to clear/mask a bit? Explain that logic gate operation.

- **Instruction:** AND
- **Explanation:** When you AND a bit with 0, the result is 0; with 1, the bit remains unchanged.
Example: AND AL, 0F0H clears lower 4 bits of AL.

13. Which logic instruction can be used to reset a bit? Explain that logic gate operation.

Instruction:

AND (Logical AND)

Explanation:

The AND instruction is used to reset (clear) a specific bit by ANDing it with 0. Any bit ANDed with 0 becomes 0, and any bit ANDed with 1 remains unchanged.

14. Which logic instruction can be used to set a bit? Explain that logic gate operation.

- **Instruction:** OR
- **Explanation:** When you OR a bit with 1, it becomes 1; with 0, it remains unchanged.
Example: OR AL, 08H sets bit 3 of AL register.

Problem: Write an assembly language program to add AX, BX, CX and DX register contents and store the result in AX register.

$$\text{ADD AX, BX} \longrightarrow AX = AX + BX$$

$$\text{ADC AX, CX} \longrightarrow AX = AX + BX + CX$$

$$\text{ADC AX, DX} \longrightarrow AX = AX + BX + CX + DX$$

#Problem: Write an assembly language program to Add 12H & 33H. Also show what change in flag register will occur?

MOV DL, 12H \rightarrow

ADD DL, 33H $12H = 0001\ 0010$

$33H = 0011\ 0011$

$45H = 0100\ 0101$

Changes in flag register

Z=0 (Because result is not zero)

C=0 (No final carry)

A=0 (Because, no half carry transfer)

S=0 (Result is positive / MSB = 0)

P=0 (Odd Parity)

O=0 (No overflow)

Problem: Execute the following program and show what change will occur in the Memory map also in the Flag registers?

MOV AX, 30F0 H

MOV BX, 2000 H

ADD AX, WORD PTR [BX]

$$\hookrightarrow AX = AX + [BX]$$

Hence,

$30F0 \text{ H} = 0011\ 0000|1111\ 0000$

$FF3C \text{ H} = 1111\ 1111|0011\ 1100$

DS Memory Map	
3C	A
FF	
0F	
FC	
00	
11	

$[BX] = FF3C$

$\begin{array}{r} 1 \\[-1ex] | \\ 0011\ 0000\ 0010\ 1100 \end{array} \Rightarrow 302C \text{ H}$

MSB = 0

#Changes in flag register:

Z = 0 (Because result is not zero)

C = 1 (End carry occurs)

A = 1 (Because the half carry transfer)

S = 0 (Because the result is positive)

P = 0 (Odd Parity)

O = 1 (Overflow occurs)

In this stage, no change occurs in the memory map.

But when we have an extra command:

ADD [BX], AL → Byte-sized contents stored
in memory location [BX] will

Hence, $\boxed{BX} = 3C$

be added with AL

$AL = 2C$

$\boxed{BX} = \boxed{BX} + AL$

$3C = 00\ 11\ |\ 1100$

$2C = 00\ 10\ |\ 1100$

$$\begin{array}{r} (+) \boxed{BX} = \\ \hline \end{array}$$

$\begin{array}{r} 0110\ |\ 1000 \\ \hline 6\ |\ 8 \end{array}$

$$= 68H$$

$\therefore \boxed{BX} = 68H.$

So, there is a change in the
memory map at location $[2000H]$.

DS Memory Map

3C 68	← 2000
FF	← 2001
0F	← 2002
FC	← 2003
00	← 2004
11	← 2005

Problem: Execute the following program.

Show that, what change will occur in the memory map also in the flag register.

MOV BX, 2000H

MOV AX, 2001H

ADD [BX], AL

$$AL = 01H = 0000 \mid 0001$$

$$[BX] = FFH = 1111 \mid 1111$$

$$\underline{1} \quad 0000 \mid 0000$$

FF 00	← 2000
00	← 2001
30	← 2002
20	← 2003
50	← 2004

After Addition, $[BX] = 00H$

Changes in flag register:

Z = 1 (because the result is zero)

C = 1

A = 1 (Half carry transfer)

S = 0 (Since MSB = 0, result is positive)

P = 1 (Even parity, since 0 1's)

O = 1 (Overflow occurs when carry comes)

□ Problem: What will be the memory change after executing the following assembly language?

MOV AX, 30F0 H

MOV BX, 2000 H

ADD WORD PTR [BX+1], AX

// ADD the contents of AX register with the word-sized contents addressed by memory location [BX+1] on [2000+1]

DS Memory	
3C	← 2000
FF	← 2001
0F	← 2002
66	← 2003
77	← 2004

$$\begin{array}{rcl}
 \text{AX} & = 30F0 \text{ H} & = 0011 \quad 0000 \\
 & & \downarrow \quad \downarrow \\
 & & 1111 \quad 0000 \\
 [\text{BX}+1] & = 0FFF \text{ H} & = 0000 \quad 1111 \\
 & & \downarrow \quad \downarrow \\
 & & 1111 \quad 1111 \\
 \hline
 & & 0100 \quad 0000 \\
 & & \downarrow \quad \downarrow \\
 & & 1110 \quad 1111 \\
 & & \downarrow \quad \downarrow \\
 & & = 40EF \text{ H.}
 \end{array}$$

Changes in memory location:

3C	← 2000
EF	← 2001
40	← 2002
66	← 2003
77	← 2004

The Flag Values:

$O = 0$
 $S = 0$
 $Z = 0$
 $A = 1$ (half carry)
 $P = 1$ (Even parity transfer)
 $C = 0$

Increment Addition (INC)

- INC adds 1 to a register or, a memory location.
- INC instruction adds 1 to any register, on memory location except a segment register!

Example: INC DS }
 INC CS } Are not Valid!!

- With indirect memory increment, the size of the data must be described by using BYTE PTR[], WORD PTR[BX] or DWORD PTR[].

Example: ↳ Double word = 32 bit

INC BL → BL = BL + 1

INC Data1 → Adds 1 to the data segment memory location contents

Indirect increment: addressed by Data1.

INC [DI] → This is incorrect!! Because assembler can not determine whether the location addressed by DI is byte, word or, double word.

INC WORD PTR [DI] → This is correct!

Problem: What change will occur in the Memory map after executing the program?

MOV SI, 2003H

MOV BX, 2006H

INC BYTE PTR [BX]

INC WORD PTR [SI]

#After INC BYTE PTR [BX]:

11	← 2000
12	← 2001
1C	← 2002
19	← 2003
F0	← 2004
DC	← 2005
CE	← 2006
15	← 2007

$$CE = 1100 \quad 1110$$

$$(+1) = 0000 \quad 0001$$

$$\begin{array}{r} 1100 \quad 1111 \\ \hline 1100 \quad 1111 \end{array} \Rightarrow CFH$$

After 4th Line:

$$F019H = 1111 \quad 0000 \quad | \quad 0001 \quad 1001$$

$$(+1) = 0000 \quad 0000 \quad | \quad 0000 \quad 0001$$

$$\begin{array}{r} 1111 \quad 0000 \quad | \quad 0001 \quad 1010 \\ \hline \end{array}$$

11	← 2000
12	← 2001
1C	← 2002
1A	← 2003
F0	← 2004
DC	← 2005
CF	← 2006
15	← 2007

$$= F01AH$$

■ Problem: What change will occur in the memory map?

MOV SI, 2000 H

INC WORD PTR [SI+1]

Here, $2000 + 1 = 2001 \text{H}$.

So, word-sized contents from the memory location [2001 H] is taken and adds 1 with it.

66	← 2000
0203	← 2001
5252	← 2002
63	← 2003
77	← 2004
88	← 2005

$$5202 \text{H} = 0101 \quad 0010 \quad 0000 \quad 0010$$

$$(+1) = 0000 \quad 0000 \quad 0000 \quad 0001$$

$$\hline \quad \quad \quad 0101 \quad 0010 \quad 0000 \quad 0011$$

$$= 5203 \text{H}.$$

■ Question: INC instruction do not affect the Carry flag bit - explain.

→ Carry does not change because we often use increment instruction in Program that depend upon the contents of the carry flag.

Increment is used to point to the next memory element in the byte-sized array of data only.

The INE (increment) instruction is used to add 1 to the value of a register or memory location. It does not affect the carry flag because it does not involve any arithmetic operation that can generate a carry or borrow.

Addition with Carry (ADC)

- An addition with Carry instruction (ADC) adds the bit in the carry flag (C) to the operand data.
- Like ADD, ADC affects the flags after the addition.
- ADC are used to add numbers more than 16 bits.

Write an assembly language program to add the following two numbers: 1616 3232 H & 1717 3131 H

MOV AX, 3232H

MOV BX, 3131H

ADD AX, BX

Mov CX, 1616H

Mov DX, 1717H

ADC CX, DX

Here, the LSB 1 word-sized

data is stored in AX

and MSB 1 word-sized data

is stored in CX.

Problem: Write an assembly language program to add two 64 bit numbers. Numbers are F3BA 3300 AE0C 0110 and DF00 19CE 017A 3344H.
Show change in flag bits.

AX	BX	CX	DX
F3BA	3300	AE0C	0110
DF00	19CE	017A	3344
<hr/>			
1 D2BA	4CCE	AF86	3454

MOV DX, 0110H

ADD DX, 3344H

MOV CX, AE0CH

ADC CX, 017AH

MOV BX, 3300H

ADC BX, 19CEH

MOV AX, F3BAH

ADC AX, DF00H

Change in Flag bits

Z=0

C=1 (End carry)

A=0 (No half carry)

S=0 (Since result is positive)

P=1

O=1 (Overflow occurs)

The Result is: D2BA 4CCE AF86 3454, C=1.

Note that: The sign bit, S=0 always for Addition operation. Because the result is always positive!!

Question: Which change will happen in the memory map after executing the program.

Here, Initial Borrow = 0.

MOV DI, 2002H

SBB BYTE PTR [DI+1], 3

Here,

$$88H = 1000 \quad 1000$$

SUB: $(-) 3H = 0000 \quad 0011$

$$\underline{85H} = 1000 \quad 0101$$

$$[2003H] = 85H$$

55	← 2000
66	← 2001
77	← 2002
88 85	← 2003
99	← 2004
11	← 2005
21	← 2006

Question: Execute the following program and show the change in the memory map?

Where initial carry = 0.

MOV DI, 2002H

DEC DI

DEC DI

SBB WORD PTR [DI], 4

→ After 4th line, DI will take

Word-sized data from memory location 2000
then subtract 4 and carry (borrow).

55	← 2000
66	← 2001
77	← 2002
88	← 2003
99	← 2004
11	← 2005

$6655H = 0110\ 0110\ 0101\ 0101$

$SUB: 04H = 0000\ 0000\ 0000\ 0100$

$\underline{-}$ $\Rightarrow 6651H = 0110\ 0110\ 0101\ 0001$

Comparison (CMP)

- CMP is a subtraction that changes only the flag bits, destination operand never changes.
- CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bit.
- Comparison is useful for checking entire contents of a register or a memory location against another value.

Examples:

- CMP CL, BL \rightarrow Instruction performs CL-BL
- CMP [DI], CH \rightarrow CH subtracts from the byte-sized contents of DS memory location addressed by DI.

Hardware Specifications of 8086 microprocessor.

DC Characteristics:

It is impossible to connect anything to the pins of the microprocessor without knowing the input current requirement for an input pin and the output current drive capability for an output pin. The input (Table 9-1) and output (Table 9-2) are given below for 8086 microprocessor.

TABLE 9-1 Input characteristics of the 8086 and 8088 microprocessors.

Logic Level	Voltage	Current
0	0.8 V maximum	$\pm 10 \mu A$ maximum
1	2.0 V minimum	$\pm 10 \mu A$ maximum

TABLE 9-2 Output characteristics of 8086 and 8088 microprocessors.

Logic Level	Voltage	Current
0	0.45V max	2.0 mA max
1	2.4 V max	-400 μA max.

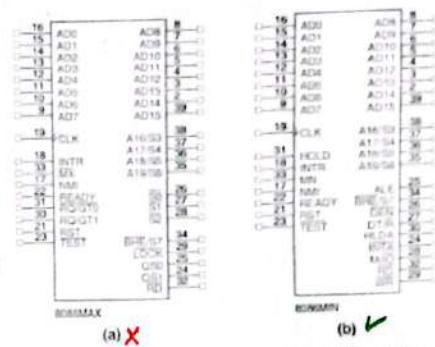


FIGURE 9-1 (a) The pin-out of the 8086 in maximum mode; (b) the pin-out of the 8086 in minimum mode.

Pin connections:

Address / Data BUS: ($AD_0 \sim AD_{15}$) Multiplexed lines used for data (during data phase) and address (during address phase).

Address Bus: ($A_{16} \sim A_{19}$) Carries higher order address bits and status signals.

ALE-(Address Latch Enable): Signals when to latch address from multiplexed lines.

RD, WR: Control signals for read/write operation, active low.

INTR, NMI: Interrupt request lines (maskable and nonmaskable)

CLK: Input for clock signal to synchronize operations.

RESET: Initializes the microprocessor.

READY: Waits for slower peripherals to catch up.

MN/MX: Selects minimum or maximum mode of operation.

DEN, DT/R, HOLD, HLDA: Used for data direction, control and DMA

Direct Memory Access

INTA: Acknowledges interrupt request.

TEST: Used with WAIT instruction for synchronization.

Q Explain the minimum mode of operation in 8086 MP.

Q Used when only one processor - single CPU in the system.
It is simpler to control, ideal for small, non-multiprocessor systems.

Activated by connecting MN/MX pin to +5V.

Control pins in minimum mode:

ALE: Signals when to latch the Address.

RD: Indicates a memory or I/O read operation.

WR: Indicates a memory or I/O write operation.

INTA: Acknowledges external interrupt.

DT/R: Controls data direction.

DEN: Data Enable pin Activates data from external transivers.

HOLD & HLDA: Used for DMA operations.

Q Why the address and data pins of 8086 are multiplexed?
Which problem arises due to multiplexing and how to solve it?

Q To reduce the total number of pins, 8086 shares certain pins address and data ($AD_0 \sim AD_{15}$) and status control lines, minimizing chip size and cost, the pins are multiplexed.

This creates timing issues, as the CPU shares lines for different purposes at different times.

Use latches like (74LS373) to demultiplex the address/data lines. The ALE - Address Latch Enable signals help latch the address during the valid address phase, allowing separation of address from data during execution. (Figure - 9.6).

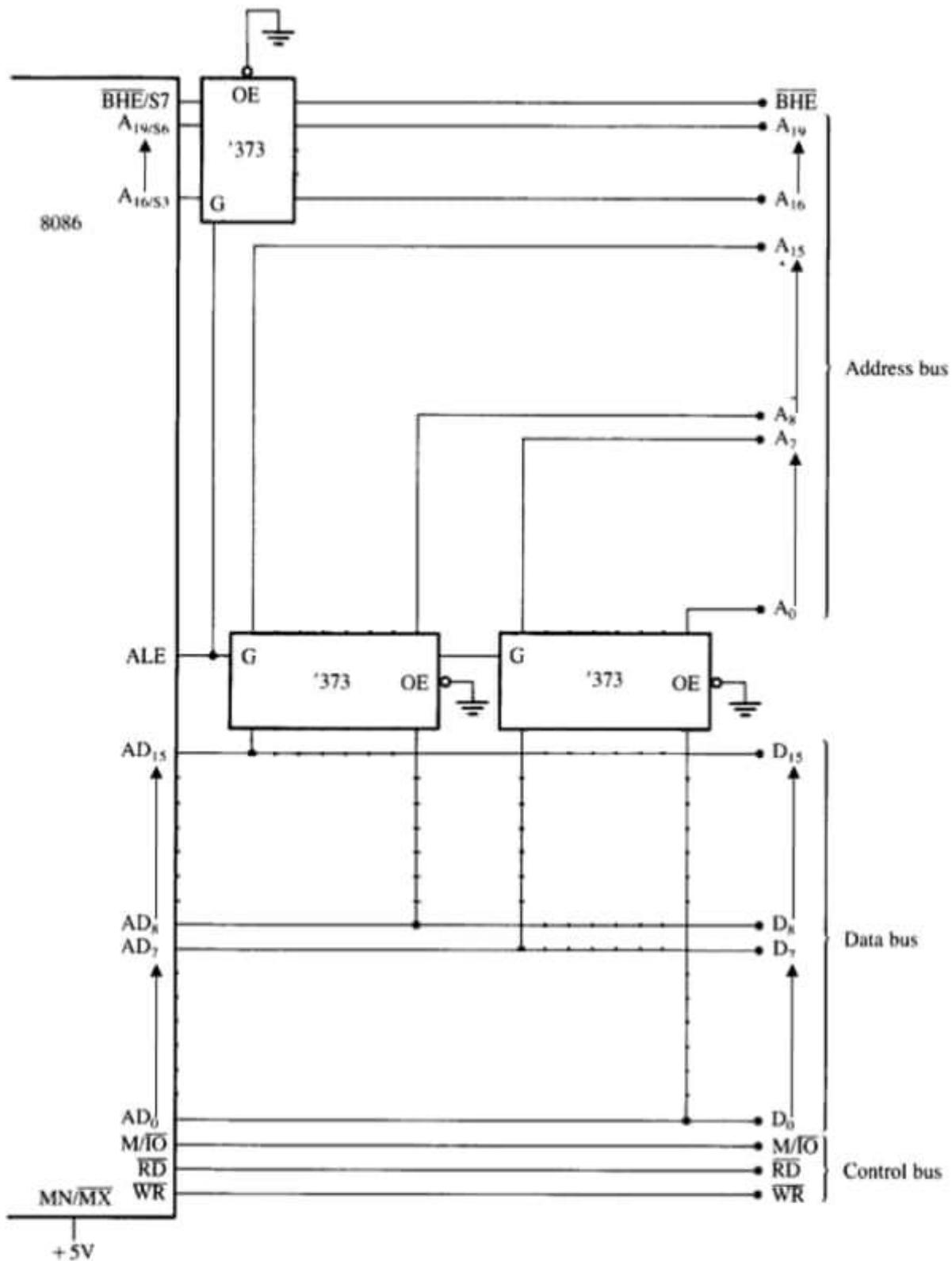


FIGURE 9–6 The 8086 microprocessor shown with a demultiplexed address bus. This is the model used to build many 8086-based systems.

Why Buffering is required in microprocessors - Explain?

- To drive multiple loads the buffering is necessary. The microprocessor output cannot directly drive many components due to limited current capability.
- Buffers prevent data loss or distortion over longer distances or heavy loads, helps maintain proper voltage levels, ensuring reliable communication.
- Buffer protects the microprocessor from potential damage caused by faulty external devices. Especially in 8086 where multiplexing occurs, buffers stabilize and isolate signals during demultiplexing.

A fully buffered 8086 microprocessor is shown in figure-9.8 where

"244 is 74LS244": An octal Buffer/line driver with 3 state outputs. Used for driving bus lines and buffering signals.

"373 is 74LS373": An octal D type transparent latch, commonly used to latch address signals in 8086.

"245 is 74LS245": An octal bidirectional transceiver, allows data flow in both directions, useful for data buses.

Explain the Read cycle timing diagram for 8086 microprocessor.

Read cycle timing (figure 9-10).

CLK: Provides timing to synchronize the read operation.

ADDRESS: High order address is sent via $A_{19} \sim A_{16}$ (unmultiplexed). Valid throughout the cycle.

ADDRESS/DATA: $AD_0 \sim AD_5$ carries lower address (T_1), then switches to data input (T_3, T_4).

RD: Goes low in (T_2), stays low through (T_3) and returns high in (T_4) indicating read complete.

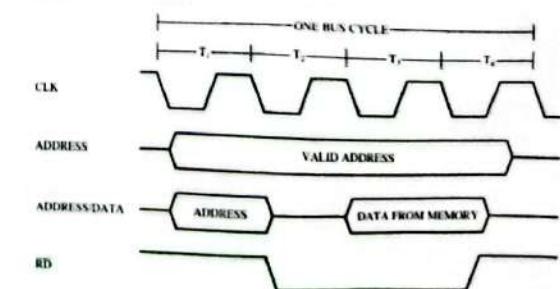


FIGURE 9-10 Simplified 8086/8088 read bus cycle.

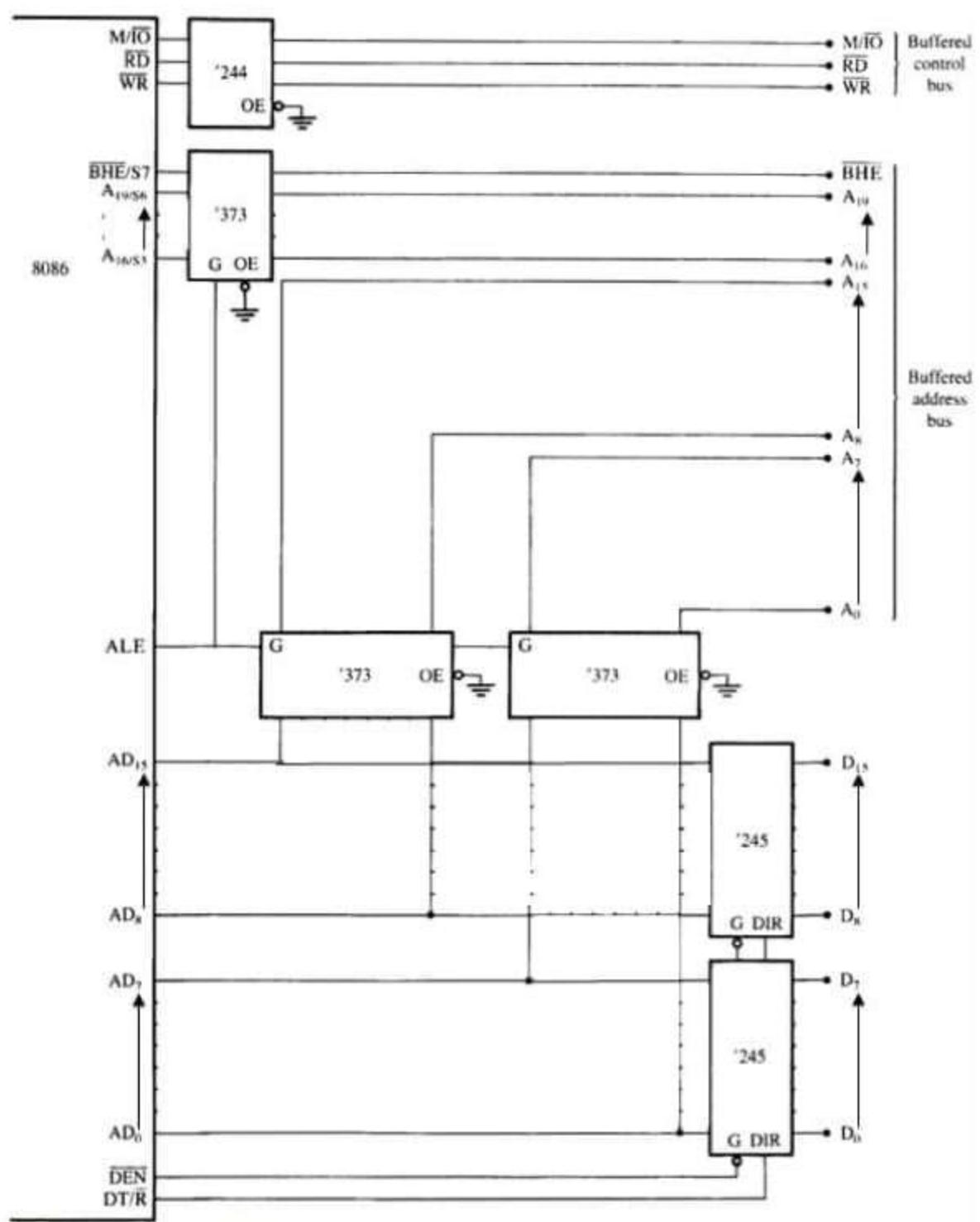


FIGURE 9–8 A fully buffered 8086 microprocessor.

Q Explain the write cycle timing diagram of 8086 mp?

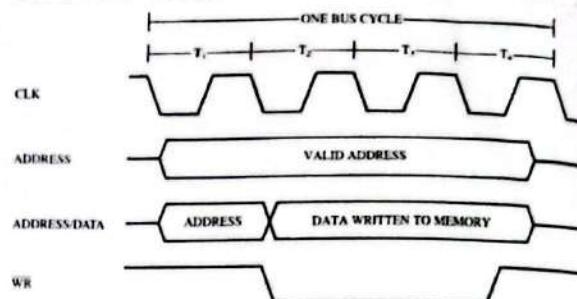
Q Write cycle timing diagram (Figure 9-9).

CLK: Governs the timing.

ADDRESS: $A_{15} \sim A_{16}$ stays valid for entire cycle.

ADDRESS/DATA: Carries address during (T_1) , switches to data output in $(T_2 \sim T_4)$

WR: Goes low in (T_2) , remains low during data transfer $(T_3 \sim T_4)$, returns high after write ends.



Q Differentiate in between minimum and maximum mode operation in microprocessors.

Minimum Mode	Maximum Mode
<ul style="list-style-type: none">Used in single microprocessor systems.Control signals are generated by the 8086 itself.Simpler hardware configuration.MN/MX pin is connected to logic '1'.NO support for multiprocessing.Timing and control handled internally.	<ul style="list-style-type: none">Used in micro-processor or co-processor systems.Control signals are generated by external bus controller(s).More complex hardware setup.MN/MX pin is connected to logic '0'.Supports multiprocessing.Timing and control handled externally.

Microcontroller:

A microcontroller is a compact integrated circuit (IC) designed to perform specific tasks in embedded systems. It includes a processor (CPU), memory (RAM, ROM), and input/output peripherals within a single chip.

Microcontrollers are used in automation, robotics, appliances, automotive systems and IoT devices to execute dedicated functions efficiently with low power consumption.

Microprocessor	Microcontroller
1. A CPU used for general purpose computing.	1. Compact IC with CPU, memory and I/O peripherals.
2. Requires external RAM, ROM and I/O.	2. Has built-in RAM, ROM and I/O ports.
3. High power consumption.	3. Low power consumption.
4. Better for complex processing tasks.	4. Optimized for specific control applications.
5. Expensive due to external components.	5. Cost-effective as everything is integrated.
6. Used in computers, high-performance devices.	6. Used in embedded systems, automation and IoT.

8255A:

The 8255A Programmable Peripheral Interface (PPI) is a 20-pin IC used for interfacing microprocessors with external devices. It has three 8-bit I/O ports (Port A, Port B & Port C) which can be configured as input or output.

It operates in three modes and (BSR) Bit set Reset mode.

The 8255A is used in keyboards, displays, ADC/DAC interfacing and industrial automation to expand the microprocessor's (I/O) capabilities. It simplifies communication between the CPU and peripherals, reducing hardware complexity.

Block Diagram of 8255A:

The block diagram in figure 15.2 shows two 8-bit ports (A and B), two 4-bit ports (C_U and C_L), the Data bus buffers (D₀~D₇) and control logic. This block diagram 15.2(b) shows a simplified but expanded version of the internal structure including a control register.

Port A: 8-bit port used for data transfer in both input and output modes, often interface with ADCs, DACs or displays.

Port B: Another 8-bit port mainly used for control functions and general I/O operations.

Port C: Can be split in two 4-bit groups, used for handshaking signals or additional I/O operations.

Data Bus: An 8-bit bidirectional bus that transfers data between the microprocessor and 8255A.

Control logic:

Manages the mode selection, read/write operations, and communication between CPU and peripheral devices.

The control section has six line, their functions and connections are as follows:

RD: Enables the read operation when the signal is low, the MPU reads data from a selected I/O port of the 8255A.

WR: This control signal enables the write operation, when the signal goes low, the MPU writes into a selected I/O port or the control register.

Reset: An active high signal, it clears the control register and sets all ports in the input mode.

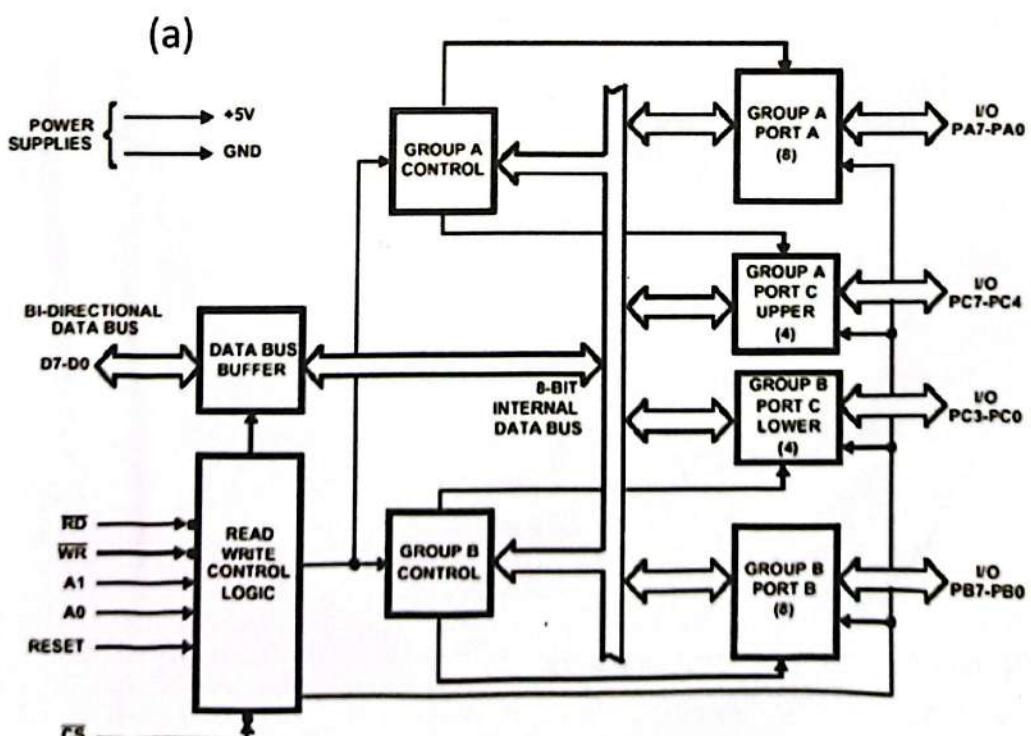
CS, A₀ and A₁: These are device select signals. CS is connected to a decoded address and A₀ and A₁ are generally connected to MPU address lines A₀ and A₁ respectively.

PA3	1	PA4
PA2	2	PA5
PA1	3	PA6
PA0	4	PA7
RD	5	WR
CS	6	RESET
GND	7	
A1	8	D0
A0	9	D1
PC7	10	D2
PC6	11	D3
PC5	12	D4
PC4	13	D5
PC0	14	D6
PC1	15	D7
PC2	16	Vcc
PC3	17	PB7
PB0	18	PB6
PB1	19	PB5
PB2	20	PB4
		PB3

8255A

Pin Names	
D_7-D_0	Data Bus (Bidirectional)
RESET	Reset Input
CS	Chip Select
RD	Read Input
WR	Write Input
A_0, A_1	Port Address
PA ₇ -PA ₀	Port A (Bit)
PB ₇ -PB ₀	Port B (Bit)
PC ₇ -PC ₀	Port C (Bit)
V _{CC}	+5 Volts
GND	0 Volts

(b)



(c)

Figure 1 : 8255A (a) Pin Diagram (b) Pin Names (c) Block Diagram

Explain how chip selection is done in 8255A?

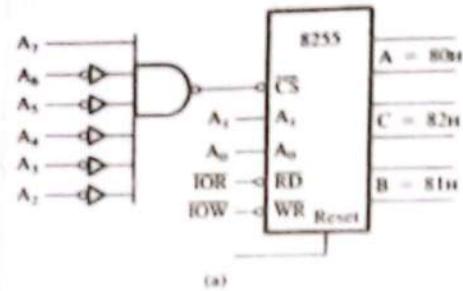
Referring to the figure

The \bar{CS} signal is the master chip select, and A_0 , A_1 specify one of the I/O ports or the control registers as given below.

\bar{CS}	A_1	A_0	Selected
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	X	X	8255A is not selected

\bar{CS}		Hex Address	Port
A ₇ A ₆ A ₅ A ₄ A ₃ A ₂ 1 0 0 0 0 0	A ₁ A ₀	= 80H 0 0 0 1 1 0 1 1	A B C Control Register

(b)



(a)

FIGURE 15.3
8255A Chip Select Logic (a) and I/O Port Addresses (b)

As an example, the port addresses in figure 15.3 (a) are determined by the \bar{CS} , A_0 and A_1 lines. The \bar{CS} line goes low when $A_7=1$ and A_6 through A_2 are at logic 0. When these signals are combined with A_0 and A_1 , the port addresses range from 80H to 83H as shown in the figure 15.3 (b).

Explain the control word for 8255A mc?

The 8-bit microcontroller's control word is explained below.

D₇ (MSB) Mode set flag 1=I/O mode
0=BSR mode.

D₆&D₅ - Mode selection for Group A (00 mode 0,
01 mode 1, 1X mode 2)

D₄ - Port A direction (1=Input, 0=Output).

D₅ - Port C upper PC₃~PC₀ direction as D₄.

D₂ - Mode selection for Group B 0-mode,
1-mode 1.

D₁ - Port B direction (1=Input, 0=Output).

D₀ (LSB) Port C lower PC₃~PC₀ direction as D₁

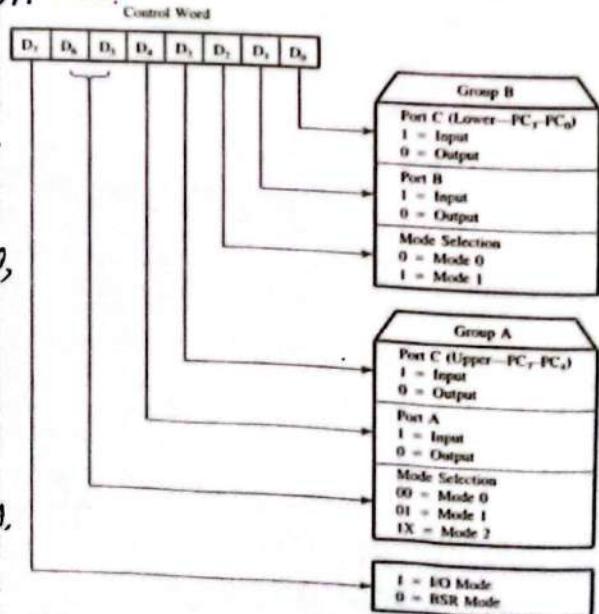


FIGURE 15.4

8255A Control Word Format for I/O Mode

SOURCE: Adapted from Intel Corporation, Peripheral Components (Santa Clara, Calif., Author, 1993), p. 3-103.

Explain the BSR mode of 8255A MC?

BSR-mode is used to set or reset individual bits of port-C without affecting the rest of the ports. It doesn't control data transfer but configures a single bit of port-C as either high or low.

Bit-8 BSR Control Word:

D₇- Enables BSR mode.

D₆ and D₄- Don't care ignored.

D₃ and D₁- selects the bit PC₀ ~ PC₇ to be set /reset (000 = PC₀ ... 111 = PC₇).

D₀- Set the bit logic high 1, and Reset the bit logic-0.

significance:

- operates independently from I/O mode.
- Does not affect other port operations.
- Used for bitwise control of peripherals like relays, LEDs etc.

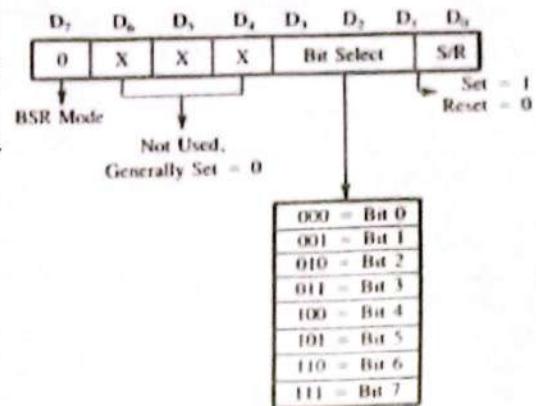


Fig: control word for BSR mode of 8255A.

What is key debounce and it's solution?

Key debounce is the removal of multiple unwanted signals caused when a key is pressed and released.

When a button is pressed, its contacts physically bounce, generating several false triggers. This results in multiple detections of single keypress.

Solutions:

- Software debouncing:
 - Add a delay (eg. 10-20ms) after detecting a keypress before reading again.
 - Use state-maintaining logic to confirm stable press/release
- Hardware debouncing:
 - Use RC filters and Schmitt triggers to smooth signal fluctuations.
 - Ensures only one clean pulse is read per press.

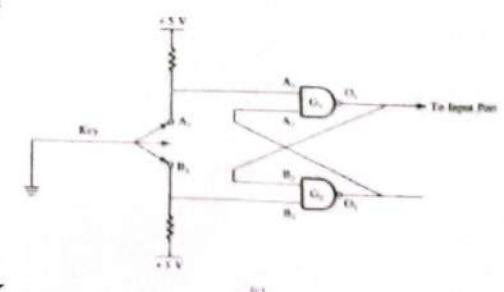
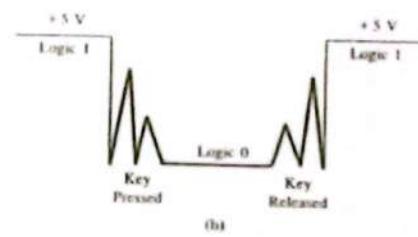


FIGURE 19.17
Debouncing Key (a) Key Bounce (b) and Key Debounce Circuit Using NAND-Gates (c)

I/O Modes of 8255 Microcontroller:

I/O mode in 8255 refers to how ports A, B and C function for data transfer. There are three I/O modes (Mode 0, Mode 1 & Mode 2). These modes define the direction (input / output) and type (simple handshaking) of communication.

Ports A and B can operate in mode 0 or 1, while port C supports Mode 2 for bidirectional communication. Control word determines the mode and configuration.

Mode-0 Basic Input/Output:

- Simple input/output - no handshaking.
- Ports A, B and lower/upper halves of port C used for data transfer.
- Data is latched during output but not latched during input.
- No interrupt features.
- Used for static I/O devices like LED's, switches.

Mode-1 Strobed Input/Output:

- Includes handshaking signals for synchronized I/O.
- Port A or B acts as input or output with strobe and acknowledge signals.
- PC₃-PC₆ are used as control signals (INTR, STB, ACK).
- Data is latched both in input and output.
- Suitable for slow I/O devices.

Mode-2 Bidirectional Data Transfer:

- Only available for port C.
- Allows bidirectional data transfer with handshaking.
- Used when both transmission and reception is needed.
- Handshaking signals are managed by port C (PC₃-PC₇)
- Useful in multiprocessor systems or dual communications.

■ Handshaking:

Handshaking in microcontrollers is a method of co-ordinating data transfer between a microcontroller and peripheral devices using control signals like STB, IBF, OBF, ACK & INTR. It ensures that both sender and receiver are ready before data is transmitted, avoiding loss or corruption. This mechanism is crucial when working with slow or asynchronous devices as it guarantees synchronized communication and reliable data exchange, especially in I/O interfacing or parallel communication systems.

Handshaking makes system operations more robust and error-free, particularly in real-time embedded applications.

■ Explain Uni and Bi directional handshaking of 8255A PPI?

Peripheral Action	8255 status (IBF, INTR, ACK)	Processor Action
sends data and STB	IBF = 1, INTR = 1	Receives data
waits	IBF = 1, INTR = 1	Reads data
.....	IBF = 0, INTR = 0	clears interrupt

In unidirectional mode, the peripheral sends data using STB (Strobe). Upon data arrival, IBF - Input Buffer Full becomes high, generating INTR - Interrupt to the processor. The processor reads the data and clears the flags. This ensures reliable and synchronized input without data loss.

Peripheral Action	8255 Status	Processor Action
requests to send/receive data	IBF or OBF set, INTR generated	Responds via RD/WR signal
Sends or receives ACK	ACK = 1 (Ack active)	Transfers data accordingly.
Completes Transaction	Flags reset	Ready for next transaction.

In Bidirectional Mode, Port A supports both sending and receiving control signals like IBF, OBF, ACK, INTR synchronized data transfer. Processor and peripheral exchange data with mutual handshaking, ensures no overlap or data corruption. This is used in full duplex scenarios.

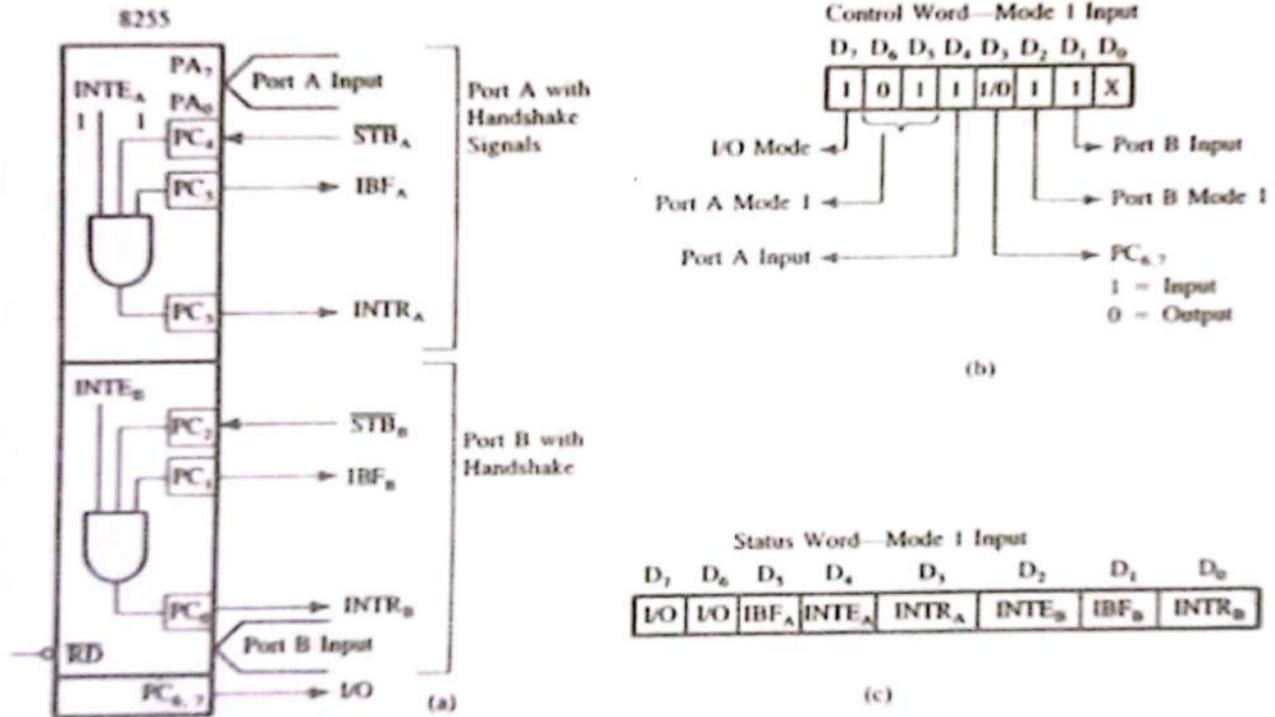


FIGURE 15.8

8255A Mode 1 Input Configuration

SOURCE: Adapted from Intel Corporation, *Peripheral Components* (Santa Clara, Calif.: Author, 1993), p. 3-110.

Draw and explain the timing waveforms for strobed input.

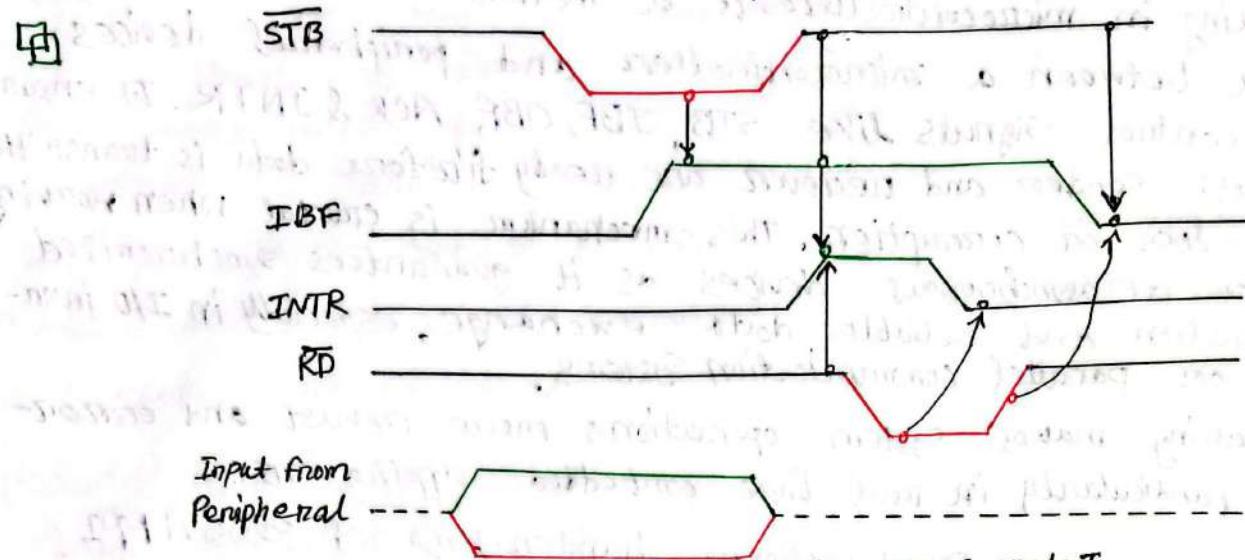


Figure 1: Waveforms of strobe input for 8255A mode I.

STB: Peripheral sends data and lowers STB to signal readiness.

IBF: set by 8255 when data is latched, goes high after a delay from STB.

INTR: 8255 rises interrupt to inform processor data is ready

RD: Processor reads data, IBF is cleared and INTR is reset.

Draw and explain timing waveforms for strobed output of 8255PI.

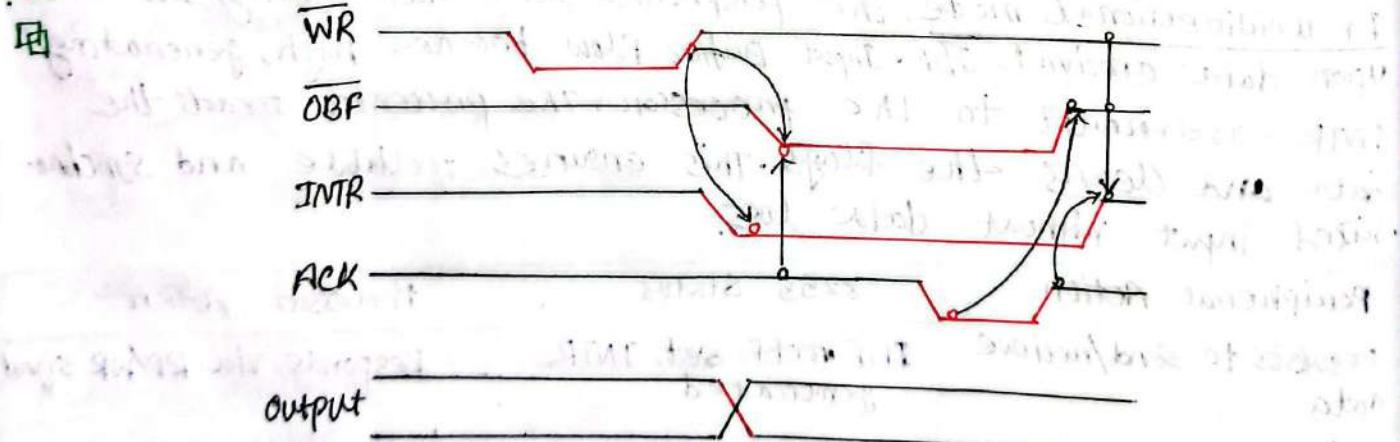


Figure 2: waveform of strobe output for 8255A mode I.

WR: Processor writes data, setting OBF.

OBF: signal data is available goes low, set after data is written.

INTR: 8255 interrupts to notify peripheral.

ACK: Peripheral acknowledges and accepts data, clears OBF and INTR.

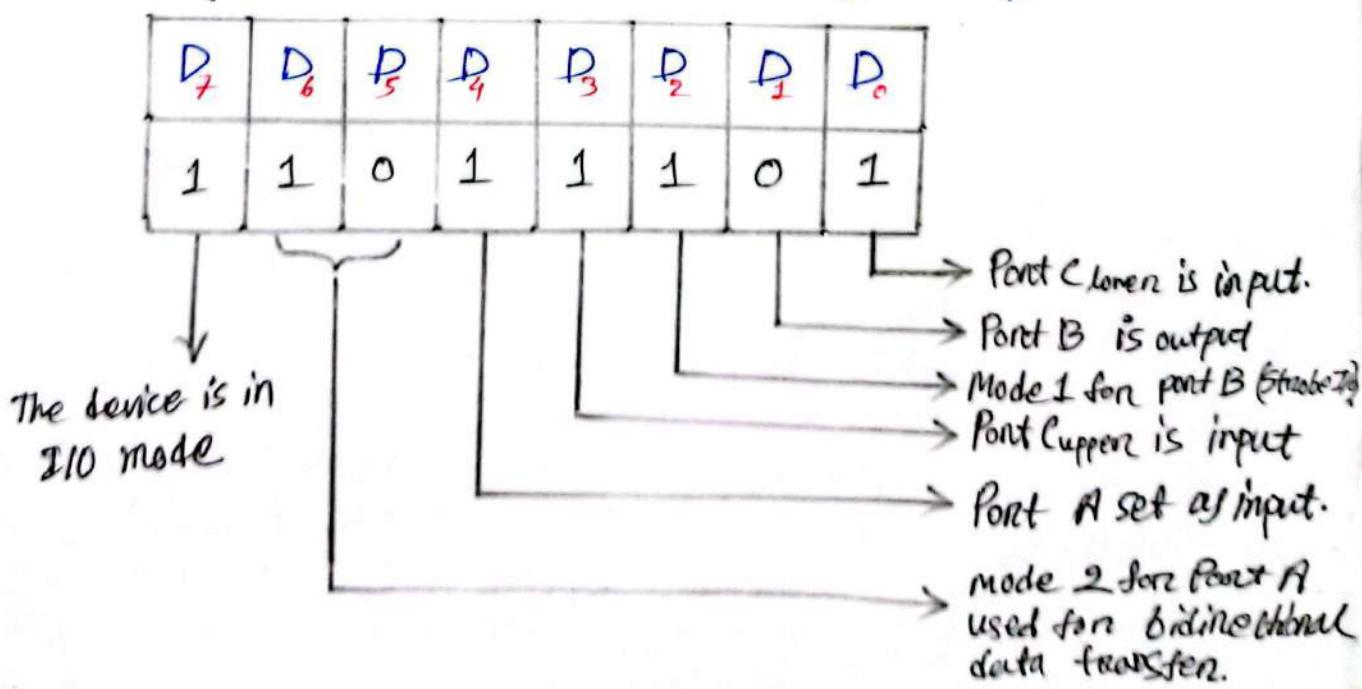
Q Discuss the features of 8255A.

Q The features of 8255A are described below.

- It is a programmable peripheral interface device.
- Has 8-bit ports, Port A, Port B, Port C.
- Can operate in three I/O modes (mode 0, mode 1 & mode 2).
- Has bit set/reset mode for controlling individual bits of Port C.
- Supports both input and output operation.
- Port C is split into two 4-bit ports Cupper and Clover.
- Required control words to set modes and directions.
- Provides handshaking signals for synchronized data transfer.
- Compatible with most microprocessors.
- Useful for real-time interfacing with peripherals like LED's, switches and displays.

Q Explain the control word DDH of 8255 PPI.

Q Converting 'DDH' to binary we get $\frac{1101}{D} \frac{1101}{D}$



Define handshaking using 8255A how microprocessor is operated with output devices - Explain.

Handshaking is a communication method between a microprocessor and I/O devices where control signals are used to ensure data is transferred only when both sender and receiver are ready.

8255A with output devices Using Handshaking:

- 8255A operates in MODE-1 strobed output.
- Microprocessor write data to port A or B.
- Control signals like (OBF - Output Buffer Full, ACK-Acknowledge) from 8255 ensure data is accepted.
- Handshake lines are managed via port C.
- Output device sends ACK after reading data allowing next data transfer.
- Ensures synchronized and reliable communication.

-
1. Identify the port addresses in Figure 15.5.
 2. Identify the Mode 0 control word to configure port A and port C₀ as output ports and port B and port C₁ as input ports.
 3. Write a program to display the reading from Port B and PC₁ to PC₀.

Example
15.1

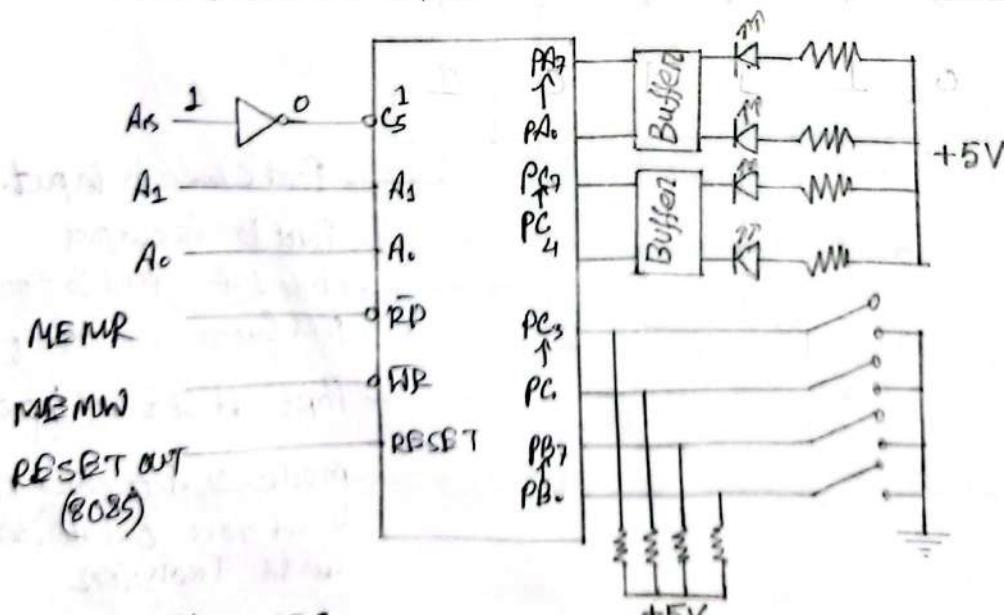
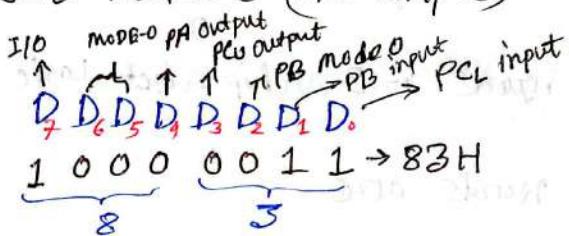


Figure: 15.5

A_{15}	A_{14}	A_{13}	A_{12}	$A_{11} \sim A_4$	A_3	A_2	A_1	A_0	Address of Port
1	0	0	0	- - - 0 - - - 0	0	0	0	0	$PA \rightarrow 8000H$
1	0	0	0	- - - 0 - - - 0	0	0	0	1	$PB \rightarrow 8001H$
1	0	0	0	- - - 0 - - - 0	0	0	1	0	$PC \rightarrow 8002H$
1	0	0	0	- - - 0 - - - 0	0	1	0	1	$CR \rightarrow 8003H$
8				0		2			
8				0		3			

For port address selection only the $A_{15}, A_0 A_1$ are considered and the rest pin are don't care (hence '0' as intel suggested). Using these values of $A_{15}, A_0 \& A_1$ port addresses are selected.

2. The control word for MODE-0 (I/O-simple) are determined below.



Control word $CW = 83H$.

3. The program is written below in Assembly Language.

PPA	EQU	8000H ; Port Address of port A.
PPB	EQU	8001H ; Port Address of port B
PPC	EQU	8002H ; Port Address of port C
PPCR	EQU	8003H ; Port Address of Control Register

MOV AL, 83H ; Loading control word 83H to AL register.

OUT PPCR, AL ; Loading data of AL in Control Register.

IN AL, PPB ; Taking input from port B

OUT PPA, AL ; Output in port A.

IN AL, PPC ; Taking input from port C entire. (xxxx 0110).

SHL AL, 4 ; shifting left from PC_L to PC_U ($PC_U \leftarrow PC_L \times 1000000$)

OUT PPC, AL ; output in Port C

Write a BSR control word subroutine to set bits PC₇ and PC₃ and reset them after 10 ms.
Use the schematic in Figure 15.3 and assume that a delay subroutine is available.

Example
15.2

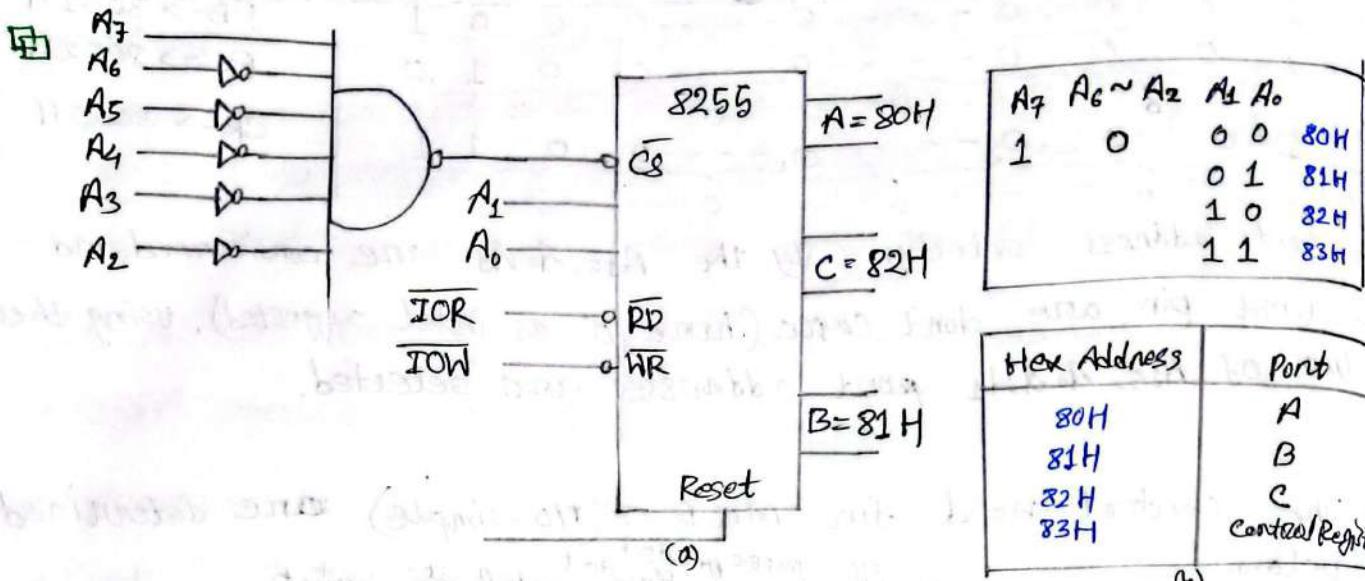


figure 15.3 (a) chip select logic (b) I/O Port addresses.

BSR control words are

	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	
To set bit PC ₇	0	0	0	0	1	1	1	1	= 0FH
To reset bit PC ₇	0	0	0	0	1	1	1	0	= 0EH
To set bit PC ₃	0	0	0	0	0	1	1	1	= 07H
To reset bit PC ₃	0	0	0	0	0	1	1	0	= 06H

Program in Assembly language:

Before the Actual program, lets count CX for 1ms if frequency considered is 1GHz, period $T = \frac{1}{f} = \frac{1}{1\text{GHz}} = 1\text{ns}$.

'Nop' instruction gives 4clk pulse delay.

Thus, if n clockpulse for 10 ms,

$$n \times 4\text{ns} = 10\text{ms}$$

$$n = \frac{10\text{ms}}{4\text{ns}} = 2.5 \times 10^{3+9} = 2.5 \times 10^6 \text{ clk pulse}$$

Converting 2.5×10^6 to Hex we have, $CX = 002625A0$ - 32 bit.
but the msp is 16 bit, so we have to divide CX in 2, 16 bit registers.
 $\text{MOV DX}, 0026H$ & $\text{MOV CX}, 25A0H$ will do.

Program:

; Define Port Addresses
 PPA EQU 80H ; Port A
 PPB EQU 81H ; Port B
 PPC EQU 82H ; Port C
 PPER EQU 83H ; Control Register.

START:

; Set PC7

MOV AL, 0FH ; BSR control word to set PC7
 OUT PPER, AL ; send to Control Register

; Set PC3

MOV AL, 03H ; BSR control word to set PC3
 OUT PPER, AL ; send to Control Register.

; Delay of ~10ms

CALL DELAY

; Reset PC7

MOV AL, 0EH ; BSR control word to reset PC7
 OUT PPER, AL ; BSR control word send to control Register

; Reset PC3

MOV AL, 06H ; BSR control word to Reset PC3
 OUT PPER, AL ; send to Control Register.

HLT ; Stop Execution.

MOV DX, 0028H ; Higher 16 bits of 2625A0H

MOV CX, 25A0H ; Lower 16 bits of 2625A0H

DELAY:

TIMER:

NOP
 NOP
 NOP
 NOP

DEC CX ; decrement CX

JNZ TIMER ; If CX ≠ 0, loop

| DEC DX ; When CX = 0, decrement DX
 | JNZ TIMER ; If DX ≠ 0, continue looping.
 | RET

- More Problems Suggested by Course Teacher

1. Write a simple LED toggling program using 8255A PPI with 8086

```

PPA EQU 80H ; Define Port A address
PPB EQU 81H ; Define Port B address
PPC EQU 82H ; Define Port C address
PPCR EQU 83H ; Define Control Register address

MOU AL, 83H ; Load control word 10000011B: Port A output, Port C lower output, Mode 0
OUT PPCR, AL ; Send control word to 8255A Control Register

MOU AL, 0FFH ; Load AL with 0FFH (all LEDs ON if active low)
OUT PPA, AL ; Output to Port A to turn all LEDs ON

CALL DELAY ; Call delay subroutine

MOU AL, 00H ; Load AL with 00H (all LEDs OFF)
OUT PPA, AL ; Output to Port A to turn all LEDs OFF

CALL DELAY ; Call delay subroutine

JMP $ ; Infinite loop to halt or repeat toggling manually

;-----
DELAY: ; Simple software delay loop
MOU CX, 0FFFFH ; Load CX with maximum value
D_LOOP: LOOP D_LOOP ; Decrement CX and loop until zero
RET ; Return from delay

```

2. Write a simple program for generating a pattern using LED

```

PPA EQU 80H ; Port A address (connected to LEDs)
PPB EQU 81H ; Port B address (input pattern source)
PPC EQU 82H ; Port C address (additional pattern input)
PPCR EQU 83H ; Control register address

MOU AL, 83H ; Load control word: Mode 0, Port A output, Port C output
OUT PPCR, AL ; Send control word to control register

IN AL, PPB ; Read input pattern from Port B
OUT PPA, AL ; Display pattern on LEDs via Port A

IN AL, PPC ; Read input from Port C
SHL AL, 4 ; Shift bits left by 4 positions (move lower nibble to upper nibble)
OUT PPC, AL ; Output modified data back to Port C

HLT ; Stop execution

```

3. Write a simple program for Buzzer control like LED using 8255A PPI with 8086.

```

PPA EQU 80H ; Port A address (buzzer connected)
PPB EQU 81H ; Port B address (input source)
PPC EQU 82H ; Port C address (additional input)
PPCR EQU 83H ; Control register address

MOU AL, 83H ; Load control word: Port A & C = output, Port B = input
OUT PPCR, AL ; Configure 8255A with control word

IN AL, PPB ; Read value from Port B (input condition)
OUT PPA, AL ; Output to Port A to control buzzer ON/OFF

IN AL, PPC ; Read from Port C (optional control or status check)
SHL AL, 4 ; Shift data left by 4 bits
OUT PPC, AL ; Output shifted data back to Port C

HLT ; Halt the program

```

4. Write a simple program for Traffic light control using vehicle count in a Traffic Junction

Assumptions & Setup

- Vehicle count is sensed via Port A (PA - address 00H).
- Traffic Lights (Red, Yellow, Green) for one lane are controlled via Port C (PC - address 02H):
 - PC0 → Green
 - PC1 → Yellow
 - PC2 → Red
- Control Word for 8255 in Mode 0:
 - Port A = Input
 - Port C = Output
- Control Word = `88H`, sent to Control Register at port 03H

Logic Based on Vehicle Count

- If vehicle count < 5 → Green light
- If vehicle count 5–10 → Yellow light
- If vehicle count >10 → Red light

```
ORG 100H

; ===== Initialize 8255 =====
MUI A, 88H           ; PA = input, PC = output, Mode 0
OUT 03H              ; Send control word to control port (03H)

START:
IN 00H               ; Read vehicle count from Port A
MOU B, A             ; Store value in B

CPI 05H               ; Compare with 5
JC GREEN              ; If <5, jump to GREEN light

CPI 0BH               ; Compare with 11 (hex 0B)
JC YELLOW              ; If between 5&10, go YELLOW

; Else >10
CALL RED
JMP START

; === GREEN LIGHT (PC0 = 1) ===
GREEN:
CALL OFFLEDS
MUI A, 01H             ; PC0 = 1 (GREEN)
OUT 02H                ; Send to Port C
JMP START

; === YELLOW LIGHT (PC1 = 1) ===
YELLOW:
CALL OFFLEDS
MUI A, 02H             ; PC1 = 1 (YELLOW)
OUT 02H
JMP START

; === RED LIGHT (PC2 = 1) ===
RED:
CALL OFFLEDS
MUI A, 04H             ; PC2 = 1 (RED)
OUT 02H
RET

; === Turn OFF All LEDs ===
OFFLEDS:
MUI A, 00H
OUT 02H
RET

END
```

5. Write Simple Program for Stepper Motor Control

Assumptions & Setup

- Stepper motor is connected to Port C (PC0 to PC3).
- Sequence for clockwise rotation (4-step mode):

Step	PC3 PC2 PC1 PC0	Hex
1	0001	01H
2	0010	02H
3	0100	04H
4	1000	08H

- Use delay loop between steps for motor timing.
- Control word for 8255:
 - Port C = output

```
ORG 100H

; === Initialize 8255 ===
MVI A, 80H           ; Port C output, Mode 0
OUT 03H              ; Control Register address

; === Begin Motor Rotation Loop ===
START:
CALL STEP1
CALL DELAY

CALL STEP2
CALL DELAY

CALL STEP3
CALL DELAY

CALL STEP4
CALL DELAY

JMP START           ; Repeat indefinitely

; === Step 1 (PC0 = 1) ===
STEP1:
MVI A, 01H
OUT 02H             ; Send to Port C
RET

; === Step 2 (PC1 = 1) ===
STEP2:
MVI A, 02H
OUT 02H
RET

; === Step 3 (PC2 = 1) ===
STEP3:
MVI A, 04H
OUT 02H
RET

; === Step 4 (PC3 = 1) ===
STEP4:
MVI A, 08H
OUT 02H
RET

; === Simple Delay Routine ===
DELAY:
MOU CX, FFFFH
LOOP1:
NOP
LOOP LOOP1
RET

END
```

6. Use TEMP sensor LM35 to measure if Fahrenheit < 98 green LED on in between yellow LED on.

Problem Summary

- LM35 outputs in °C; to convert to °F:

$$F = (C \times 9/5) + 32$$

For comparison in Fahrenheit:

- < 98°F → Green LED on (PC0)
- 98°F to <100°F → Yellow LED on (PC1)
- ≥ 100°F → Red LED on (PC2)
- Approx conversion for thresholds (in Celsius):
 - 98°F ≈ 37°C
 - 100°F ≈ 38°C

Control Word (for 8255 - Mode 0)

Port	Direction
Port A (PA)	Input (LM35)
Port C (PC)	Output (LEDs)
Port B (PB)	Don't care (assume Output)

Control Word = 10001000B = 88H

```

MUI A, 88H      ; Control Word: PA=input, PC=output
OUT 03H          ; Send to Control Register (port address = 03H)

START:
IN 00H          ; Read from Port A (PA - LM35 input)
MOU B, A         ; Store in B for comparison

CPI 25H          ; Compare with 37°C (approx 98°F)
JC GREEN         ; If less than 37, turn ON GREEN LED

CPI 26H          ; Compare with 38°C (approx 100°F)
JC YELLOW        ; If <38 but >=37, turn ON YELLOW

; If =38
CALL RED
JMP START

; === GREEN LED ===
GREEN:
CALL OFFLEDS
MUI A, 01H        ; PC0 = 1 (GREEN)
OUT 02H          ; Send to Port C
JMP START

; === YELLOW LED ===
YELLOW:
CALL OFFLEDS
MUI A, 02H        ; PC1 = 1 (YELLOW)
OUT 02H
JMP START

; === RED LED ===
RED:
CALL OFFLEDS
MUI A, 04H        ; PC2 = 1 (RED)
OUT 02H
RET

; === Turn OFF all LEDs ===
OFFLEDS:
MUI A, 00H
OUT 02H
RET

```

8254 PIT:

The 8254 is a programmable Interval Timer (PIT) designed to generate accurate time delays and frequencies. It is widely used in microprocessor based systems to perform timing control functions such as event counting, real-time clocking and baud rate generation. The 8254 has three 16-bit counters each capable of operating in different modes.

Features:

- 3 programmable 16 bit down counters.
- Operates in six different modes
 1. Interrupt to terminal count.
 2. Hardware Retriggerable one-shot.
 3. Rate generator.
 4. Square wave generator.
 5. Software Triggered strobe, and
 6. Hardware triggered strobe.
- Independent clock and gate input for each counter.
- Read-back command allows status and count reading.
- Provides precise timing and control functions.
- Compatible with Intel microprocessors.

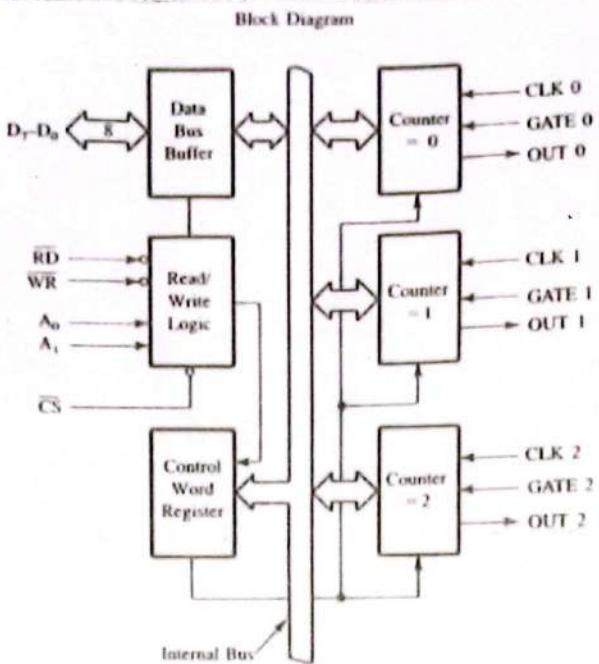
Used: It is primarily used in where precise timing and control are needed such as in real time systems, frequency generation and interrupt handling.

Pin Configuration

D ₇	1	24	V _{CC}
D ₆	2	23	WR
D ₅	3	22	RD
D ₄	4	21	CS
D ₃	5	20	A ₁
D ₂	6	19	A ₀
CLK 0	7	18	CLK 2
OUT 0	8	17	OUT 2
GATE 0	9	16	GATE 2
GND	10	15	CLK 1
GND	11	14	GATE 1
GND	12	13	OUT 1

Pin Names

D ₇ -D ₀	Data Bus (8 Bit)
CLK N	Counter Clock Inputs
GATE N	Counter Gate Inputs
OUT N	Counter Outputs
RD	Read Counter
WR	Write Command or Data
CS	Chip Select
A ₀ -A ₁	Counter Select
V _{CC}	+5 Volts
GND	Ground

FIGURE 15.23
8254 Block Diagram

Pin functions:

A₀ A₁: The address inputs select one of the four internal registers within the 8254.

CS: chip select enables the 8254 for programming and for reading or writing a counter.

V_{CC}: Power connects to +5 V power supply.

GND: Ground. Connect to the system ground bus.

Gates: The gate input controls the operation of the counter in some modes of operation.

D₀~D₇: Bidirectional three state data bus lines connected to system data bus.

CLK: The clock input is the timing source for each of the internal counters. This input is often connected to the PCLK signal from the microprocessor system bus controller. A counter output is where the waveform generated is available.

A ₁	A ₀	function
0	0	Counter 0
0	1	counter 1
1	0	Counter 2
1	1	Control Word

Gate 0	gate ip for count0
Gate 1	gate ip for count1
Gate 2	gate ip for count2

- RD: Read causes data to be read from the 8254 and often connected to the I_{OC}C signal.
- WR: Write causes data to be written to the 8254 and often connects to the Write Strobe I_{OWC}.

Control Word of 8254:

8254 PIT control word format is explained below:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	BCD
SC1	SC0	RW1	RW0	M2	M1	M0		

D₇~D₆: (SC₁-SC₀) selects counter as per table.

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (See Read Operations)

D₅~D₄: (RW₁-RW₀) Read write control as per table

RW—Read/Write:		
RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only.
1	0	Read/Write most significant byte only.
1	1	Read/Write least significant byte first, then most significant byte.

D₃~D₁: (M₂-M₀) selects one of the counter mode as per table

M-MODE:			
M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

D₀: 0 = Binary Counting; 1 = BCD Counting

Read - write operation of 8254:

Write operation:

- Control word must be written before loading the initial count.
- Control format must match Control Word: LSB MSB or LSB then MSB.
- Counter starts with a proper clock and gate signal.

BCD:

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

Read operations:

1. Simple read:

- Stop counter first (inhibit clock).
- Read LSB then MSB

2. Counter Latch command:

- Latches Count without stopping counting.
- Data held until read or reprogrammed.

3. Read-Back Command:

- Latch multiple counters in one command.
- Can monitor OUT pin, Mode/Null flag, etc.
- Efficient for reading multiple counters together.

Explain the Read-Back command for 8254 PTI.

The read-back command is explained below:

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	1	COUNT 0	STATUS 0	CNT2	CNT1	CNT0	0

$$A_0, A_1 = 11$$

$$\overline{CS} = 0$$

$$\overline{RD} = 1$$

$$\overline{WR} = 0$$

The read-back Command allows the user to read the count and status of the counter(s). This command is not available in 8253. The format of the command is shown above.

The command is written in the control register, and count of the specified counter(s) can be latched if COUNT D₅ is '0'. A counter or a combination of counters is specified by keeping the respective CNT bits D₁, D₂ and D₃ high.

For example the control word 1 1 0 1 0 1 1 0 (D6H) written in the control register will latch the counts of Counter 0 and Counter 1, and these counts can be obtained by reading respective counter port addresses. The latched counts are held until they are read or the counters are reprogrammed. The read back command eliminates the need of writing separate counter latch commands for different counters.

Status byte command:

- Send via read back command with status bit D₄ = 0.
- It latches status info (not count) of selected counters.
- Allows reading out pin, null count and mode without stopping counting.
- Read from the counters data port after issuing the command.
- Useful for monitoring counter states via software.

(a) Read-Back Command Format							
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	1	COUNT	STATUS	CNT 2	CNT 1	CNT 0	0
D ₅ : 0 = Latch Count of Selected Counter(s)							A ₀ , A ₁ = 11
D ₄ : 0 = Latch Status of Selected Counter(s)							CS = 0
D ₃ : 1 = Select Counter 2							RD = 1
D ₂ : 1 = Select Counter 1							WR = 0
D ₁ : 1 = Select Counter 0							
D ₀ : Reserved for Future Expansion, Must Be 0							

(b) Status Byte							
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
OUTPUT	NULL COUNT	RW1	RW0	M2	M1	M0	BCD
D ₇ : 1 = Out Pin is 1							
0 = Out Pin is 0							
D ₆ : 1 = Null Count							
0 = Count Available for Reading							
D ₅ , D ₄ : Counter Programmed Mode							

FIGURE 16.28
Read-Back Command Format (a) and Status Byte (b)

Explain the following control word for 8254 timer.

B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	1	1	0	1	0	1

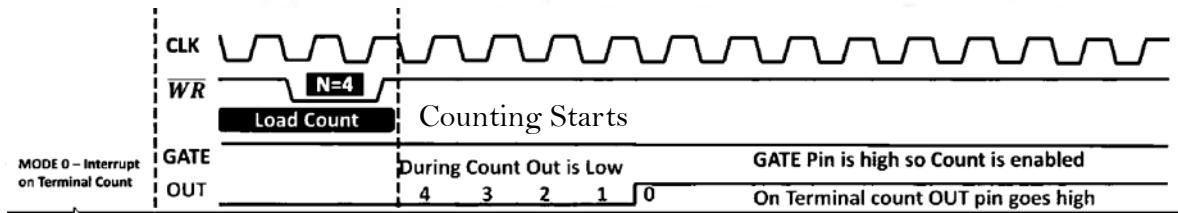
The control word is broken down as.

SC_1	SC_0	RW_1	RW_0	M_2	M_1	M_0	BCD
1	0	1	1	0	1	0	1

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

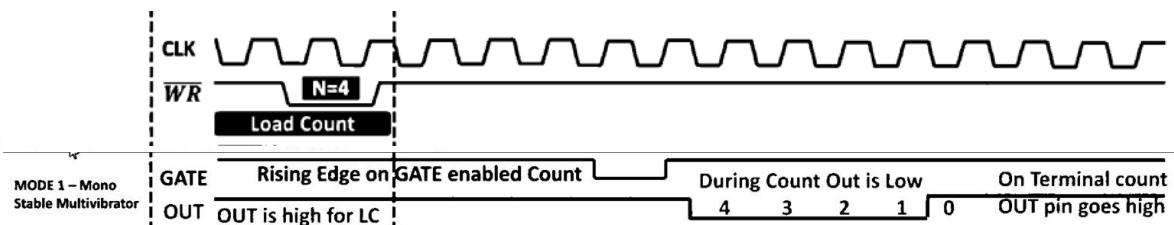
Selects Counter 2 Read/Write Mode 2 Operates in BCD
 LSB then Rate Generator mode.
 MSB

- Explain the six modes of operating 8254 PIT.
- The mode of operation for 8254 PIT are selected based on the $M_2 \sim M_1$ bits of its control word. The operation for each mode is explained below.



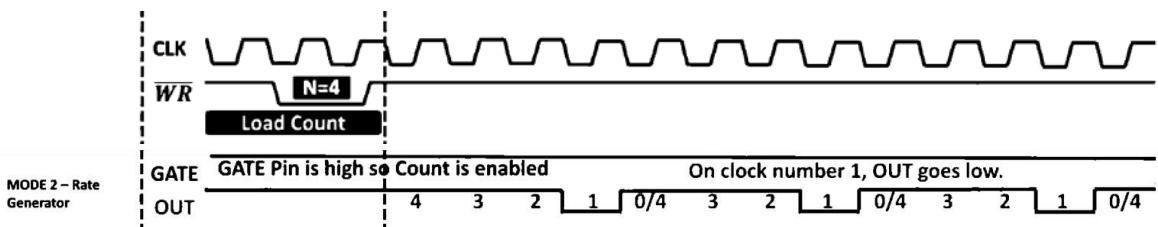
MODE 0: Interrupt on terminal count

- Counter starts after loading count and GATE = 1.
- Out is low during counting.
- When count reaches '0' OUT goes high.
- Counting starts from clock's falling edge.



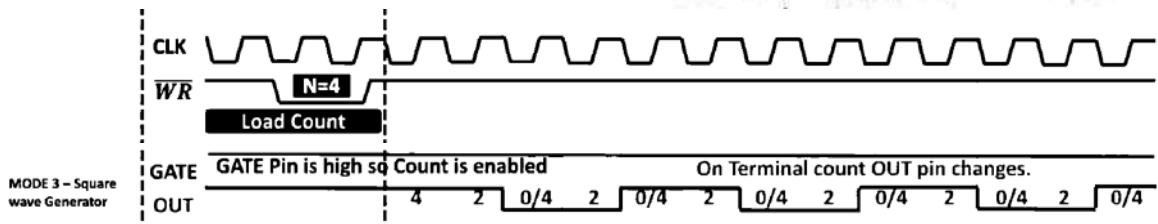
MODE 1: Hardware Retriggerable one-shot / Monostable multivibrator mode

- Counting starts on rising edge of GATE, after the corresponding clock falling edge.
- Out is high for one full count duration.
- Retriggerable - new GATE restarts count.
- Suitable for generating fixed length pulses.



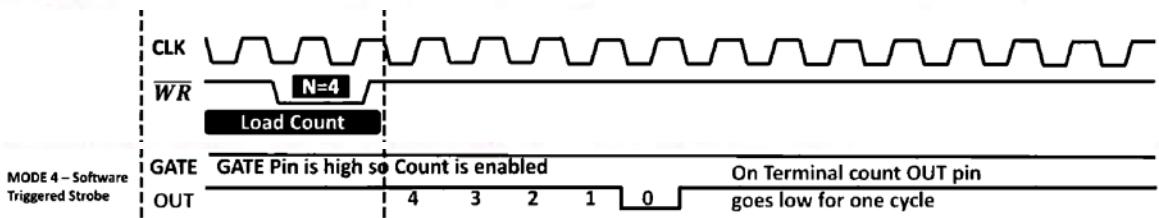
MODE-2: Rate Generator.

- continuous pulse generator.
- OUT goes low for one clock at terminal count.
- Count restarts automatically.
- Ideal for generating periodic interrupts.



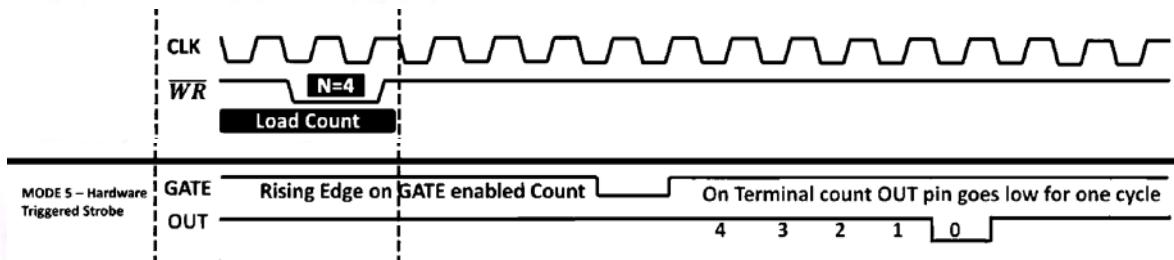
MODE-3: Square Wave Generator.

- Produces square wave output.
Count is divided into two halves (high & low).
- For square-wave generating count even numbers must be taken and decrement is two
- OUT toggles at terminal (0) count. (i.e. for first 0 edge of OUT is low and for second edge '0' OUT is high).
- Used in waveform generation.



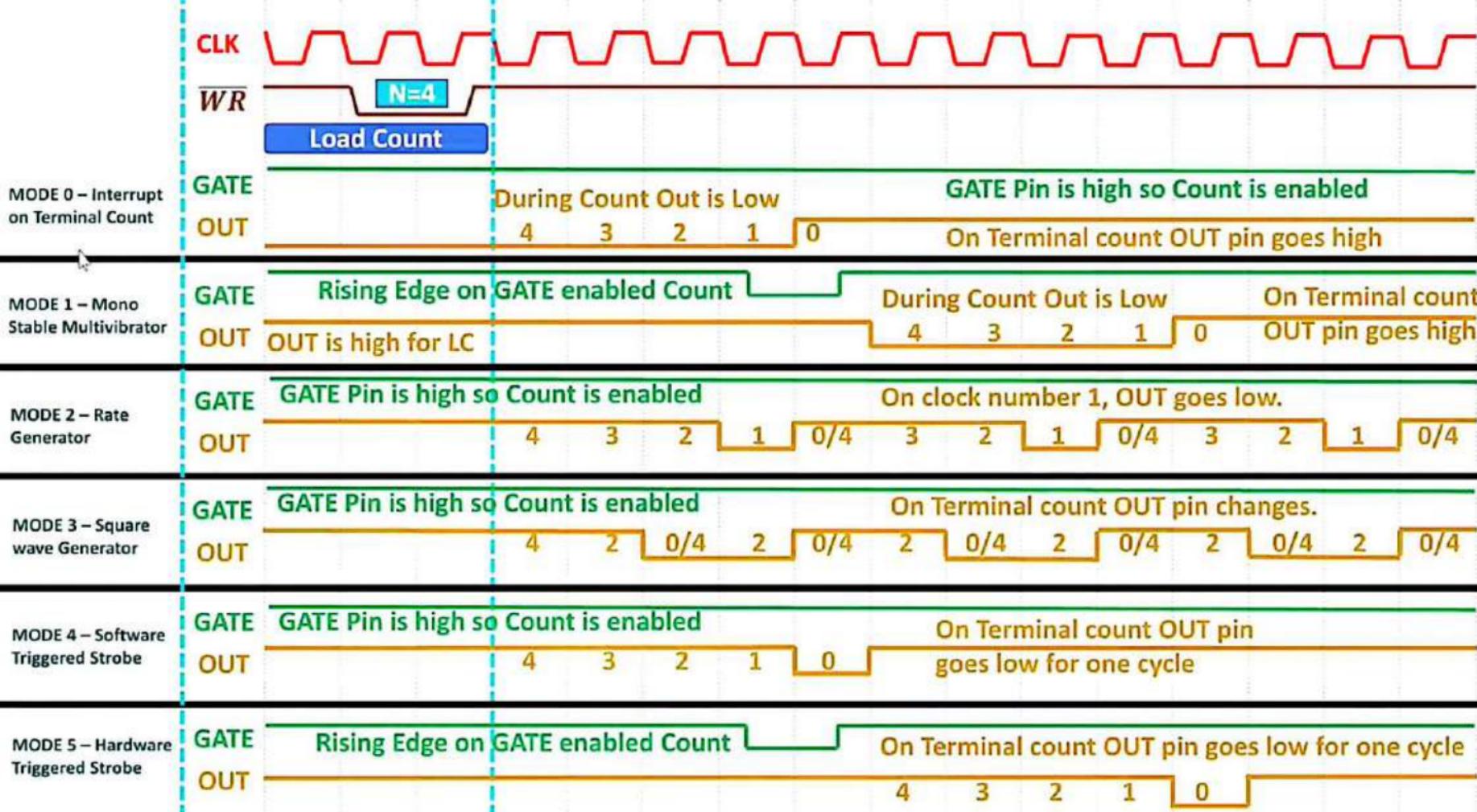
MODE-4: Software Triggered Strobe

- counting starts immediately after loading.
- OUT stays high during counting.
- At terminal count OUT goes low for one clock.
- Best for delayed pulses.



MODE-5 : Hardware Triggered Strobe:

- Counting starts on GATE's rising edge.
- OUT remains high during counting.
- OUT goes low for clock at terminal count.
- Used for externally triggered delayed pulses.



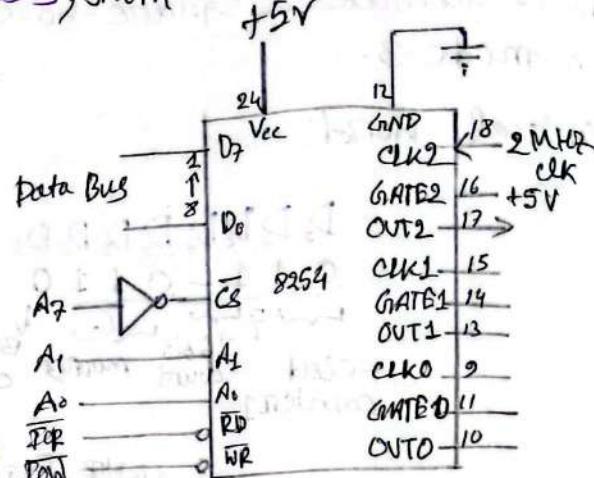
Write instructions to generate a pulse every 50 μs from Counter 0 (refer to Figure 15.26).

Prev Pg.
Next Pg.

To generate a pulse for every 50μs, from Counter 0,
it should be initialized to mode 2
(Rate generator) and state 0 should
be high (i.e. 5V).

Control Word:

D₇ D₆ D₅ D₄ D₃ D₂ D₁ D₀
 1 0 0, 0 1 0 1 0 0 = 14H
 Select Load 8 bit CNT. MODE-2



Binary Count figure 15.26.

$$\text{Count} = \frac{50 \times 10^{-6}}{0.5 \times 10^{-6}} = 100 \\ = 64H$$

Alternative,

Given,
clock frequency is 2MHz = 0.5μs
pulse should be generated at 50μs per

$$\text{Count} = \frac{2\text{MHz}}{20\text{kHz}} = 100 = 64H \\ \uparrow 1/50\mu s$$

Instructions:

PULSE: MVI A, 00010100; Load the Accumulator with 14H - control word
OUT 83H ; send the control word to control register.

MVI A, 64H ; Load Accumulator with 64H - this is the pulse
OUT 80H ; output for counter 0 with low order byte.

HALT ; Stop program execution.

Example

15.4

Write instructions to generate a 1 kHz square wave from Counter 1 (refer to Figure 15.26). Assume the gate of Counter 1 is tied to +5 V through a 10 k resistor. Explain the significance of connecting the gate to +5 V.

Q Generate a square wave with period of the i/p frequency to 8254 is 1MHz.
A To generate a square-wave from counter 1, it should be initialized in mode-3.

Control Word:

D₇, D₆, D₅, D₄, D₃, D₂, D₁, D₀
 0 1 1 1 0 1 1 0 = 76H
 Select counter 1 16 bit count MODE-3 Binary count.

$$\text{Count} := \frac{1 \times 10^3}{0.5 \times 10^{-6}} = 2000 = 07D0H$$

$\leftarrow 1\text{kHz}, T = \frac{1}{1\text{kHz}}$

$\uparrow 2\text{MHz} \Rightarrow T = \frac{1}{2\text{MHz}}$.

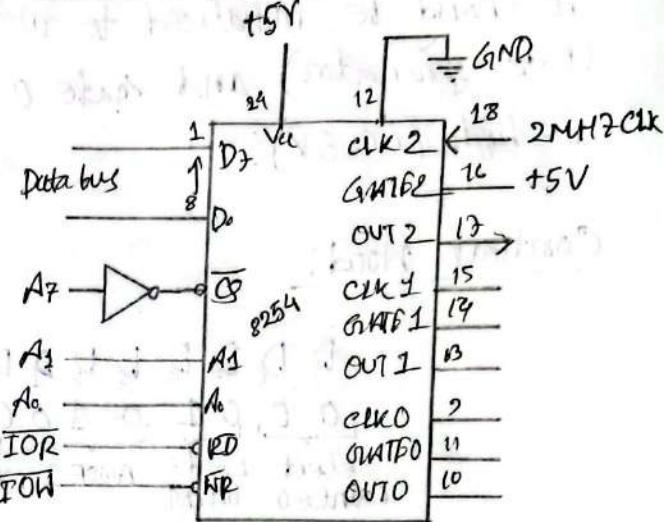


Figure: 15.26.

Instruction:

SQWAVE : MVI A,01110110B ; control word for mode 3 counter 1
 OUT 83H ; To control registers.
 MVI A,D0H ; Lower order byte of the count.
 OUT 81H ; Load counter 1 with lower order byte.
 MVI A,07H ; High order byte of the count.
 OUT 81H ; Load counter 1 with high order byte.
 HALT ; Stop program execution.



**Example
15.5**

Write a subroutine to generate an interrupt every 1 sec. Refer to Figure 15.26 for counter addresses.

The clock frequency given is 2MHz; +5V
To obtain a delay of 1sec.

The count should be,

$$1 \times 2^M = 2^M$$

$$= 2 \times 10^6$$

$$= 001E8480.$$

But these counters are 16 bit counter and the value 2×10^6 is too high for 16 bit register.

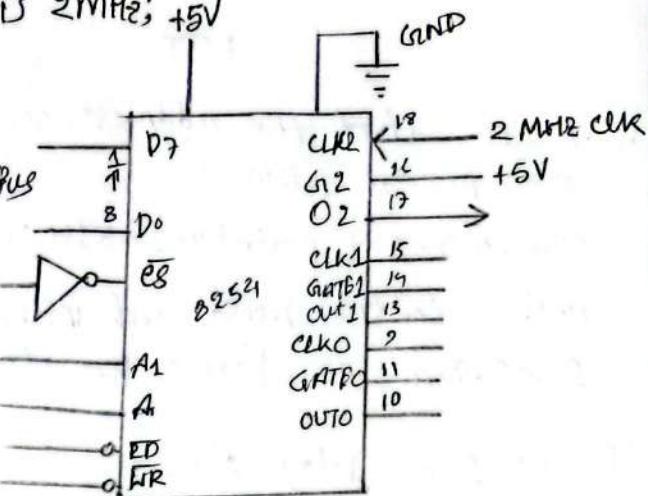


figure 15.26

Thus we can divide $2^M = 50k \times 40 = 50000 \times 40$. Thus we can use counter 1 for 50000 and counter 2 for next 40 .

The output of counter 1 can be used as a clk to counter2. Control word:

	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
Counter1 :	0	1	1	1	0	1	0	0
counter2 :	1	0	0	1	0	1	0	0

select counter Load count MODE-2 Binary count

Instructions:

CNT1LO
CNT1HI
COUNT2
SECOND

EQU 50H ; lower part of C350H is defined for count 1 Low.
EQU C3H ; Higher part of C350H is defined for count 1 High.
EQU 40 ; Count2 is defined in decimal.
MVI A, 01110100B ; Control word mode 2 counter 1.
OUT 83H ; Write in 8254 counter Register.
MVI A, 10010100B ; Control word mode 2 counter 2.
OUT 83H ; Write in 8254 counter register.
MVI A, CNT1LO ; Low order byte of count 50000.
OUT 81H ; Load Counter1 with low order byte.
MVI A, CNT1HI ; High order byte of count 50000.
OUT 81H ; Load Counter1 with high-order byte.

```

MVI A, COUNT2 ; count for counter 2
OUT 82H        ; load counter 2
RET

```

- Assume that the address of counters and control registers of 8254 PIT as follows:
counter 0 = 80H; counter 1 = 81H; counter 2 = 82H; control register = 88H
Write control word and instruction to generate a pulse every 50ms from counter 0. The clock input is 2 MHz.

The 50ms delay gives.

$$\text{Count} = \frac{50 \times 10^6}{0.5 \times 10^6} = 100000 \approx 186AOH$$

since 186AOH is in 20 bits, counter 0 and counter 1 are used in cascade, where

→ Use counter 0 as the LSB counter in MODE 2.

Use counter 1 as the MSB counter in MODE 2.

Connect output of counter 0 to the CLK input of counter 1.

Control Words:

	$D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0$
counter 0	0 0 1 1 0 1 0 0 = 34H
counter 1	$\underbrace{0 1}_{\text{select counter}} \underbrace{1 1}_{\substack{\text{LSB} \\ \text{then} \\ \text{MSB}}} \underbrace{0 1}_{\text{MODE-2}} \underbrace{0 0}_{\text{Binary}}$

Instructions:

PULSE :

MVI A, 34H ; Control word for counter 0
OUT 88H ; send to control register.

MVI A, A0H ; LSB of counter 0
OUT 80H ; Load to counter 0

MVI A, 00H ; MSB of counter 0
OUT 80H ; Load to counter 0

MVI A, 74H ; control word for counter 1
OUT 88H ; send to control register

MVI A, 86H ; LSB of counter 1
OUT 81H ; Load to counter 1

MVI A, 01H ; MSB of counter 1
OUT 81H ; Load to counter 1

HALT ; End program.

8259A

The 8259A is a programmable interrupt controller designed to manage hardware interrupts for Intel microprocessors like the 8085, 8086 & 8088. It allows handling upto eight interrupt inputs, replacing the processor's single interrupt pin with more efficient management.

Features:

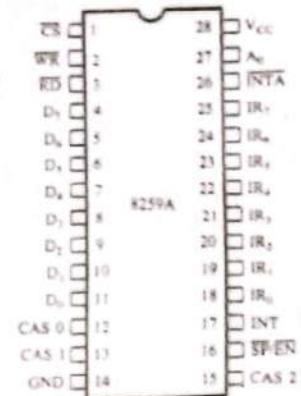
- Manages 8 hardware interrupt via control registers.
- Vectorizes interrupt requests anywhere in memory.
- Supports multiple priority modes (nested, automatic specific)
- Allows individual masking of interrupts.
- Displays status of pending, in service and masked interrupts.
- Supports both level triggered and edge-triggered interrupts.
- Expandable upto 64 interrupt lines by cascading multiple 8259As.
- Compatible with both 8085 and 8086/8088 microprocessor modes.
- Upward-compatible with 8259 and adds more advanced features.

Significance: It enhances interrupt handling, improves processing efficiency and supports multitasking in complex systems.

Applications:

- Used in microprocessor systems (8085, 8086/8088) to manage multiple interrupt signals efficiently.
- Supports priority based interrupt handling, essential in real time systems.
- Crucial in embedded systems, industrial controllers and computer systems.
- Reduces CPU overhead by handling interrupt management externally.

Pin Configuration



Pin Names	
D-D ₀	Data Bus (Bidirectional)
RD	Read Input
WR	Write Input
A ₀	Command Select Address
CS	Chip Select
CAS ₋ CAS ₊	Cascade Lines
SPEN	Slave Program/Enable Buffer
INT	Interrupt Output
INTA	Interrupt Acknowledge Input
IR ₋ IR ₊	Interrupt Request Inputs

FIGURE 15.29
The 8259A Block Diagram

Block Diagram of 8259A:

Data bus buffer: Interfaces with the microprocessor's data bus for command and data transfer.

Read/Write Logic: Controls internal read write operations using RD, WR and CS signals.

Control Logic: Coordinates interrupt requests and microprocessor responses.

Interrupt request Register (IRR):

Holds incoming interrupt requests.

In service Register (ISR): Tracks interrupts being serviced.

Interrupt Mask Register (IMR): Masks/unmasks individual interrupt.

Priority Resolver: Determines priority of active requests.

Cascade Buffer Comparator: Enables cascading for more than 8 interrupts.

Control Words: Initialize and configure operations via ICWs and OCWs.

Q Explain the interrupt operation of 8259A. Refer to the figure above.

• 8259A receives interrupt signals from external devices via IR.

- Stores all interrupt requests in the interrupt request register (IRR).
- Priority Resolver checks and selects the highest priority interrupt.
- If not masked (checked via IMR), it sends an interrupt signal to the CPU via control register.
- After the CPU acknowledges via INTA, ISR stores the inservice interrupt info.
- Sends the corresponding interrupt vector to the CPU via data bus.
- Once the CPU finishes servicing, it issues End Of Interrupt (EOI) command.

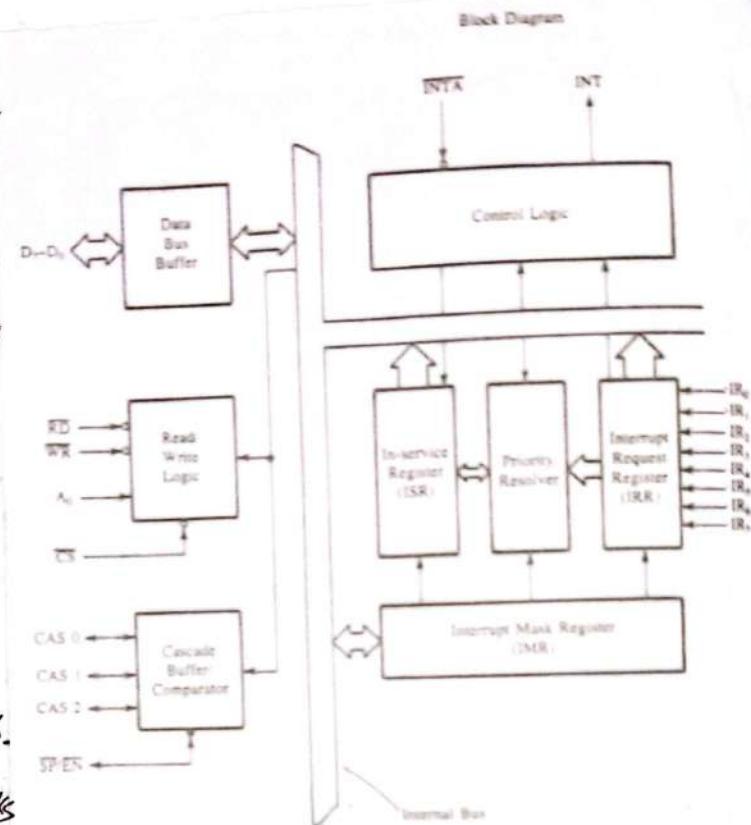


Figure: Block Diagram of 8259A

The ISR clears the respective bit, and the next highest priority request is processed.

Write short notes on

- (i) Fully nested mode &
- (ii) Special fully nested mode of 8259A.

(i) fully nested mode.

- Default mode for 8259A with fixed priorities from IR₀ (highest) to IR₇ (lowest).
- Higher priority interrupt can preempt lower ones.
- No lower priority interrupts can be serviced until the higher one is completed.
- Ensures orderly interrupt handling.
- Best for single or master only configuration.

(ii) Special fully nested mode.

- Used in cascaded systems with master and slave 8259As.
- Allows a slave to interrupt during a master's IR.
- Overrides standard blocking to allow same-level interrupts from slaves.
- Useful in complex systems with multiple levels of interrupts.
- Enables better multitasking by expanding nested handling.

Determine the value of ICW1 for 8086 mode, if it is triggered by rising edge trigger and there is a single PIC. Also use interval 4 and note that ICW4 is needed.

ICW1

A ₀	A ₇	A ₆	A ₅	1	LTIM	ADI	SNGL	IC4
0	0	0	0	1	0	1	1	1

Address bit A₀=0,
for ICW1

Don't care for
8086

Fixed
always

Edge
Triggered

Intervals,

single
mode

ICW4
needed

Differentiate between 8254 timer and 8259A interrupt controller

8259 A	V/S	8254
<ul style="list-style-type: none">Used for managing hardware interrupts.Can handle 8 interrupts; expandable to 64.Controls priorities and handling of multiple IRQs.Has multiple priority modes.Reduces CPU load by handling interrupts efficiently.Uses initialization (ICW) and operation (OCW) words.		<ul style="list-style-type: none">Used for generating accurate time delays.Has 3 independent 16-bit counters.Controls timing in systems like clocks.No interrupt priority handling.Interacts with CPU for timing tasks.Operates through control words.

Write down the algorithm for initialization of 8259 PIT

The algorithm is provided below.

- Send ICW1: Start initialization, specify edge/level trigger, and if ICW4 is needed.
- Send ICW2: set interrupt vector address.
- Send ICW3: set Master/Slave configuration if cascaded.
- Send ICW4: Define mode (8086, 8088 auto EOI, etc).
- Send OCW5: mask/unmask interrupts, set priority control operations.

Q Why DMA is required? Explain the operation of DMA controller with suitable diagram.

A DMA—Direct Memory Access is required to transfer data between memory and peripherals without involving the CPU, improving system efficiency and speed.

Operation of DMA Controller:

- If I/O wants to send data to memory, then request DRQ is sent.
- To take control of system data buses DMA will send HOLD signal to MPU.
- To give control of address and data, MPU will give HLDA [Hold Ack] which indicates that now DMA is master of Buses. So now buses will be managed by DMA for memory and I/O.
- DMA places memory address and control signals, Data is transferred between memory and I/O.
- Once HOLD signal goes low, MPU will take control of system buses and then MPU becomes master.

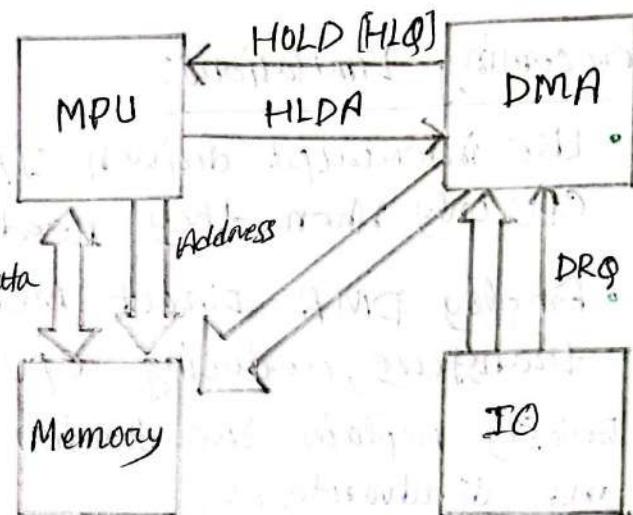


Figure: Operation of DMA controller.

Q What is polled approach? Describe its limitations and how to overcome those.

A In the polled approach the CPU continuously checks the status of a device to determine if it needs service. It does not rely on interrupts, leading to constant CPU involvement.

Limitations:

- Wastes CPU time by checking devices repeatedly, even when idle.
- Not efficient for real time or multitasking system.
- Slower response time to I/O requests.

Overcoming Limitations:

- Use interrupt driven I/O, where devices notify the CPU only when they need service.
- Employ DMA- Direct Memory Access for large data transfers, reducing CPU load.

Q Briefly Explain the function of loaders and its advantages and disadvantages.

A Loaders are system programs that load executable files into memory and prepare them for execution by the CPU.

Advantages:

- Simplifies program execution.
- supports dynamic loading and linking.
- Efficient memory management.
- Reduces user responsibility for memory handling.

Disadvantages:

- Adds overheads to system startup time.
- Complexity increases in multi-module or dynamic linking.
- Debugging becomes harder with dynamic relocations.

Assembly code

QNA

1. How does assembly language get translated into machine language?

Assembly language is converted into machine language by a program called an **assembler**. The assembler translates mnemonic instructions (e.g., `MOV`, `ADD`) into binary opcode and operands understood by the CPU. The result is an **object file** containing machine code.

2. Write an assembly language program to execute the following operation and store the result in DX register.

Expression: $5 * AL - 6 * 3H + (BH / 8)$

```
asm                                         ⌂ Copy ⌂ Edit

MOV AL, 06H      ; Example value in AL
MOV BH, 40H      ; Example value in BH

; Calculate 5 * AL
MOV BL, AL
MOV CL, 05
MUL CL          ; AX = AL * 5

; Store result temporarily
MOV CX, AX

; Calculate 6 * 03H = 18 (decimal)
MOV AX, 06
MOV BL, 03H
MUL BL          ; AX = 18

; Subtract from previous result
SUB CX, AX

; Divide BH by 8
MOV AX, 0000
MOV AL, BH
MOV BL, 08
DIV BL          ; AL = BH / 8

; Add to previous result
ADD CL, AL
MOV DX, CX      ; Final result in DX
```

★ Assumes unsigned values and 8-bit registers. Adjust initial values as per need.

3. Using loop instructions, Write the assembly language program for the following series: $SUM = 5 + 6 + 8 + \dots + 110$

```
asm                                         ⌂ Copy ⌂ Edit

MOV CX, 15      ; 15 numbers in series
MOV AX, 5        ; First number
MOV BX, 0        ; SUM initialized
NEXT:
ADD BX, AX      ; Add current term to SUM
ADD AX, 1        ; Increment
ADD AX, 1        ; For skipping even value in 6, 8, etc.
LOOP NEXT
MOV DX, BX      ; Final result in DX
```

4. If AX = EF0EH, Write an assembly language program to:

- Mask rightmost 2 bits
- Set bit 8 and 9
- Invert 13th and 14th bits

```
asm                                     ⌂ Copy ⌂ Edit

MOV AX, 0EF0EH      ; Original value

; Mask rightmost 2 bits (bit 0 and 1)
AND AX, 0FFFC

; Set bit 8 and 9
OR AX, 0300H

; Invert bit 13 and 14
XOR AX, 6000H
```

5. Write an assembly language program to:

- Set leftmost 3 bits
- Invert 5th and 6th bit (bit 4 and 5)

```
asm                                     ⌂ Copy ⌂ Edit

MOV AX, 0000H

; Set leftmost 3 bits (bit 15,14,13)
OR AX, E000H

; Invert bit 4 and 5
XOR AX, 0030H
```

6. Determine the value of AX after execution of the following code. What is the purpose of this code?

Write a more efficient code to replace this one.

```
asm
```

```
CODE SEGMENT
ASSUME CS:CODE, DS:CODE
MOV AX, CS
MOV DS, AX

MOV BX, 0
MOV AX, 1
MOV CX, 6
L1: ADD AX, BX
    JMP L2
L2: ADD AX, BX
    NEG BX
    ADD BX, AX
    SUB AX, BX
    JMP L3
L3: LOOP L1
    HLT
```

Copy Edit

Answer:

- **Purpose:** This code calculates the sum of the first 6 natural numbers.
- **Final Value of AX:** 15 ($1+2+3+4+5+6$).
- **More Efficient Code:**

```
asm
```

```
MOV AX, 6
MOV BX, AX
INC BX
MUL BX
SHR AX, 1 ; AX = (n*(n+1))/2
```

Copy Edit



7. From the following program, determine how many times the program loop will execute? Also determine the values of AX, BX after the final execution.

```
asm
```

```
MOV AX, 30FFH
IF AL==FF && AH<33
    MOV BX, 0DEFH
    ADD AX, BX
END IF
```

Copy Edit

Answer:

- **Condition:** $AL = FF, AH = 30 \rightarrow 30 < 33 \Rightarrow \text{TRUE}$
- $BX = 0DEFH$
- $AX = 30FFH + 0DEFH = 3EEEH$
- **Loop Executions:** None, it's a conditional, not a loop.

12. Write down the function of the following pins of 8086 MP.

- i) TEST: Used with WAIT instruction; checked by processor.
 - ii) READY: Waits till slower devices respond.
 - iii) LOCK: Prevents other processors from accessing the bus.
 - iv) DEN: Enables external data transceivers.
 - v) HOLDA: Indicates CPU has released the bus.
-

13. Write down the functions of the following pins of 8086 microprocessor:

- i) BHE/S7: Bus High Enable; enables higher byte of data bus.
- ii) INTR: Maskable interrupt request input.
- iii) HOLD: Requests the CPU to release bus control.
- iv) RQ/GT1: Request/Grant pin for bus access in max mode.
- v) RQ/GT0: Another Request/Grant line with higher priority.

“And We made the sky a protected ceiling, but they, from its signs, are turning away”

(Quran 21:32)

