2019-20

8

A.a) Short notes on various types of jumping instruction: [6] i) Signed Conditional Jumps ii) Unsigned Conditional Jumps iii) Single Flag Jumps

Here is a **short note** on the **various types of jumping instructions** in assembly language (particularly in x86 architecture):

---

i) Signed Conditional Jumps

- These jumps are used **when working with signed numbers** (positive or negative).
- They depend on the **sign (SF), overflow (OF)**, and **zero (ZF)** flags.
- Common signed jump instructions:
    - `JG` / `JNLE` – Jump if greater (ZF = 0 and SF = OF)
    - `JL` / `JNGE` – Jump if less (SF ≠ OF)
    - `JGE` / `JNL` – Jump if greater or equal (SF = OF)
    - `JLE` / `JNG` – Jump if less or equal (ZF = 1 or SF ≠ OF)

---

ii) Unsigned Conditional Jumps

- These are used **when working with unsigned numbers** (only positive).
- They depend on the **carry (CF)** and **zero (ZF)** flags.
- Common unsigned jump instructions:
    - `JA` / `JNBE` – Jump if above (CF = 0 and ZF = 0)
    - `JB` / `JNAE` – Jump if below (CF = 1)

- JAE / JNB – Jump if above or equal (CF = 0)
- JBE / JNA – Jump if below or equal (CF = 1 or ZF = 1)

---

iii) Single Flag Jumps

- These jumps are based on the **value of a single flag**.
- Common single-flag jump instructions:
  - JE / JZ – Jump if equal / zero (ZF = 1)
  - JNE / JNZ – Jump if not equal / not zero (ZF = 0)
  - JC – Jump if carry (CF = 1)
  - JNC – Jump if no carry (CF = 0)
  - JS – Jump if sign (SF = 1)
  - JNS – Jump if no sign (SF = 0)
  - JO – Jump if overflow (OF = 1)
  - JNO – Jump if no overflow (OF = 0)

Let me know if you need **examples** or a **diagram** for better understanding!

Here are **examples** for each type of jumping instruction:

✅ i) Signed Conditional Jump Example

```
MOV AX, 5
CMP AX, -3
JG greater_label    ; Jump if greater (signed comparison)

greater_label:
; AX is greater than -3 (signed), this code executes
```

📌 Here, JG (Jump if Greater) considers **signed values**. Since 5 > -3, it jumps.

✅ ii) Unsigned Conditional Jump Example

```
MOV AX, 5
CMP AX, 250
JA above_label      ; Jump if above (unsigned comparison)

above_label:
; AX is above 250 (unsigned), but since 5 < 250, it won't jump
```

📌 JA checks **unsigned** comparison. Since 5 < 250 in **unsigned**, it doesn't jump.

✅ iii) Single Flag Jump Example

```
MOV AX, 10
SUB AX, 10          ; AX = 0, so Zero Flag (ZF) is set
JE equal_label      ; Jump if equal (ZF = 1)

equal_label:
; Executes because AX == 0, so ZF = 1
```

📌 JE (Jump if Equal) checks **only the Zero Flag (ZF)**.

Let me know if you'd like to simulate this with **step-by-step flag changes or memory status**!

B.) Write an assembly language program for comparing two string and concatenating two string

Here are two simple **8086 Assembly Language Programs** using **MASM/TASM syntax**:

✅ 1. Program to Compare Two Strings

This program compares two strings and sets a flag if they are equal.

```
.MODEL SMALL
.STACK 100H

.DATA
   STR1 DB 'HELLO$', 0
   STR2 DB 'HELLO$', 0
   MSG_EQUAL DB 'Strings are equal$'
   MSG_NOTEQUAL DB 'Strings are not equal$'

.CODE
MAIN:
   MOV AX, @DATA
   MOV DS, AX

   LEA SI, STR1     ; Point SI to first string
   LEA DI, STR2     ; Point DI to second string

COMPARE_LOOP:
   MOV AL, [SI]
   MOV BL, [DI]
   CMP AL, '$'      ; End of string?
   JE END_COMPARE
   CMP AL, BL
   JNE NOT_EQUAL
   INC SI
   INC DI
   JMP COMPARE_LOOP

NOT_EQUAL:
   LEA DX, MSG_NOTEQUAL
   JMP DISPLAY

END_COMPARE:
   LEA DX, MSG_EQUAL

DISPLAY:
   MOV AH, 09H
   INT 21H

   MOV AH, 4CH
   INT 21H
END MAIN
```

✅ 2. Program to Concatenate Two Strings

```
.MODEL SMALL
.STACK 100H

.DATA
   STR1 DB 'HELLO ', 0
   STR2 DB 'WORLD$', 0
   CONCAT DB 20 DUP('$')   ; Empty buffer to store concatenated result

.CODE
MAIN:
   MOV AX, @DATA
   MOV DS, AX
   MOV ES, AX              ; Use ES for output buffer

   LEA SI, STR1
   LEA DI, CONCAT

COPY_STR1:
   MOV AL, [SI]
   CMP AL, 0
   JE COPY_STR2
   MOV [DI], AL
   INC SI
   INC DI
   JMP COPY_STR1

COPY_STR2:
   LEA SI, STR2

COPY_LOOP2:
   MOV AL, [SI]
   CMP AL, '$'
   JE END_DISPLAY
   MOV [DI], AL
   INC SI
   INC DI
   JMP COPY_LOOP2

END_DISPLAY:
   MOV BYTE PTR [DI], '$'
   LEA DX, CONCAT
   MOV AH, 09H
   INT 21H

   MOV AH, 4CH
```

INT 21H
END MAIN

💡 Notes:

- $ is used as a **string terminator** for DOS INT 21H function 09H.
- Strings must be null-terminated or $-terminated based on how you plan to process them.
- These programs work in **8086 real mode**, suitable for emulators like **DOSBox** or **EMU8086**.

Would you like these programs explained step-by-step?

7. a) What is Microcontroller? What are the advantages and disadvantages of Microcontroller? [4]

Sure! Here's a **comprehensive and detailed explanation** of the question:

✅ 7(a) What is a Microcontroller? What are the Advantages and Disadvantages of Microcontroller?

🔹 What is a Microcontroller?

A **Microcontroller** (often abbreviated as **MCU** – Microcontroller Unit) is a **compact integrated circuit (IC)** designed to govern a specific operation in an embedded system. It is a **self-contained system** with a processor core, memory, and programmable input/output peripherals.

It is sometimes referred to as a **"computer on a chip"** because it contains all essential elements of a computer on a single IC. Microcontrollers are typically used in **automated systems**, such as:

- Home appliances (washing machines, microwave ovens)

- Consumer electronics (TV remote, digital camera)
- Automobiles (airbags, anti-lock braking system)
- Industrial control systems
- Robotics
- IoT (Internet of Things) devices

🔸 Basic Components of a Microcontroller:

1. **CPU (Central Processing Unit)** – Executes instructions and controls operations.
2. **RAM (Random Access Memory)** – Temporary memory for data storage during program execution.
3. **ROM/Flash** – Non-volatile memory to store the program.
4. **I/O Ports** – Interface with external devices such as sensors, switches, motors, displays.
5. **Timers and Counters** – For timing operations and event counting.
6. **Serial Communication Interface** – UART, SPI, I2C for external communication.
7. **ADC/DAC (Analog to Digital / Digital to Analog Converters)** – Interface with analog signals.
8. **Interrupt System** – To handle events like keypress or sensor input efficiently.

🔹 Advantages of Microcontroller:

1. ### 🔸 Compact and Cost-Effective:
    - Combines CPU, memory, and I/O on a single chip.
    - Reduces size and cost of the system.
    - No need for additional external components for most tasks.

2. 🔸 **Low Power Consumption:**
    - Ideal for battery-powered or energy-sensitive devices.
    - Many microcontrollers have power-saving modes (sleep/idle).
3. 🔸 **Real-Time Performance:**
    - Can respond quickly to input/output changes in real-time systems.
    - Interrupt-driven processing allows quick response to critical events.
4. 🔸 **Easy to Interface with Peripherals:**
    - Built-in GPIO (General Purpose Input/Output) pins and communication protocols make it easy to connect with external devices like sensors, LEDs, motors, etc.
5. 🔸 **Reliable for Repetitive Tasks:**
    - Excellent for controlling repetitive or fixed-function applications like motor control or signal monitoring.
6. 🔸 **Wide Availability and Range:**
    - Available in many variants (8-bit, 16-bit, 32-bit).
    - Manufacturers like Intel, Microchip, Atmel, STMicroelectronics provide a wide variety of MCUs.

🔹 Disadvantages of Microcontroller:

1. 🔸 **Limited Processing Power:**
    - Generally not designed for heavy computation or multitasking.
    - Not suitable for running complex operating systems or applications.
2. 🔸 **Restricted Memory:**

- RAM and ROM are limited, which restricts the size of code and data handling capabilities.

3. 🔸 **Limited Number of I/O Pins:**
   - A fixed number of I/O ports makes it less flexible for complex I/O-heavy applications.
4. 🔸 **Fixed Architecture:**
   - Most MCUs come with a fixed set of peripherals and architecture.
   - Difficult to upgrade hardware features once selected.
5. 🔸 **Less Flexible than Microprocessors:**
   - Microcontrollers are good for specific tasks but not general-purpose computing.
6. 🔸 **Slower Clock Speeds:**
   - To reduce power consumption, they typically operate at lower clock speeds than microprocessors.

🔹 Difference Between Microcontroller and Microprocessor (Brief):

Feature	Microcontroller	Microprocessor
Components	CPU, RAM, ROM, I/O on one chip	Only CPU (others external)
Application	Embedded/control systems	General-purpose computing
Power Consumption	Low	Higher
Size	Small	Larger (requires peripherals)
Cost	Low	Higher

🔹 Popular Microcontroller Families:

- **8051 series** – Basic 8-bit MCUs, widely used for educational and control applications.

- **AVR (e.g., ATmega328)** – Used in Arduino boards.
- **PIC (by Microchip)** – Widely used in industrial and consumer products.
- **ARM Cortex-M** – 32-bit MCUs used in high-performance embedded systems (e.g., STM32 series).
- **MSP430 (by Texas Instruments)** – Known for ultra-low power consumption.

🔹 Applications of Microcontroller:

1. Washing Machines, Microwave Ovens
2. Air Conditioners and Refrigerators
3. Digital Cameras and Smart Watches
4. Smart Home Devices (IoT)
5. Car Engine Control Units (ECU), Airbags
6. Industrial Automation
7. Robotics
8. Medical Devices (ECG, glucose meters)
9. Mobile Phones and Remote Controllers
10. Elevators and Vending Machines

🔚 Conclusion:

Microcontrollers are **essential building blocks** of modern electronic systems due to their **compactness, low power, and real-time control capabilities**. While they have **some limitations in processing power and memory**, their benefits in **embedded and dedicated tasks** far outweigh the drawbacks.

---

b) What is the difference between timer and counter of a microcontroller? List the factors to be considered for selection of microcontroller for particular application.

Here is a complete and detailed answer for **Question 7(b)**:

✅ (b) What is the Difference Between Timer and Counter of Microcontroller?

🔹 Timer:

- A **timer** is a device that **counts internal clock pulses** to measure time intervals.
- It is used to **generate delays**, control real-time events, or measure time duration.
- Works with the **microcontroller's internal clock frequency** (e.g., 1 MHz, 16 MHz).

Example Use: Delay generation, time measurement, periodic interrupt generation.

🔹 Counter:

- A **counter** is a device that **counts external events** or pulses applied to an input pin.
- It is used for **event counting**, such as the number of external pulses, sensor activations, or revolutions of a motor shaft.
- Works with **external signals** rather than internal clock.

Example Use: Counting how many times a button is pressed, RPM measurement, frequency counting.

🔸 Difference Table:

Feature	Timer	Counter
Input Source	Internal clock	External event (via pin)
Main Use	Measure time/delay	Count occurrences of events
Example Application	Generating delays, real-time clocks	Frequency/rate measurement

Trigger Source	Clock cycles	External pulses (rising/falling edge)
Controlled by	Internal programming	External hardware signals

✅ Factors to be Considered for Selection of Microcontroller for a Particular Application:

When choosing a microcontroller for a specific task, the following factors should be considered:

🔸 1. Application Requirements:

- Determine whether it needs **data processing**, **control**, **communication**, or **sensor interfacing**.
- Choose 8-bit, 16-bit, or 32-bit MCU based on task complexity.

🔸 2. Number of I/O Pins:

- Based on how many sensors, switches, displays, or other devices you want to connect.

🔸 3. Memory Requirements:

- **RAM** (for variables) and **Flash/ROM** (for code).
- Larger programs or real-time tasks require more memory.

🔸 4. Clock Speed (Performance):

- Higher clock = faster processing.

- Select based on whether your task is **time-sensitive** (e.g., motor control, signal processing).

🔸 5. Power Consumption:

- Important for **battery-powered** devices.
- Look for low-power or ultra-low-power MCUs (e.g., MSP430, ATmega).

🔸 6. Timers, Counters, ADC/DAC, PWM:

- Select MCU based on peripherals needed:
    - **ADC** for analog sensors
    - **PWM** for motor or LED brightness control
    - **UART/SPI/I2C** for communication

🔸 7. Cost and Availability:

- Choose one that fits your budget and is **easily available** in the market.

🔸 8. Development Tools and Community Support:

- Availability of **compilers, simulators, IDEs (like Arduino IDE, MPLAB)**.
- Strong community or documentation helps with debugging and learning.

🔸 9. Packaging and Size:

- For compact devices, choose smaller packages (e.g., SMD).

- Depends on physical space constraints of the system.

🔸 10. Operating Voltage and Temperature Range:

- Must match with sensors and devices used.
- Important for **industrial or outdoor** applications.

✅ Summary (For Revision):

Factor	Consideration
Bit width (8/16/32-bit)	Task complexity
I/O ports	Number of devices to connect
Memory	Code/data storage needs
Clock speed	Performance requirement
Peripherals	ADC, PWM, communication ports, etc.
Power	Battery-operated systems?
Cost & availability	Budget and supply
Tools & support	IDE, libraries, examples
Package type	Size and mounting constraints
Environment suitability	Voltage, temperature limits

Let me know if you want this translated into **Bengali**, formatted into a **PDF**, or explained with **diagrams/examples**!

c. Differentiate between Memory mapped IO and IO mapped IO with reference to 8085 microprocessors.

Here is a detailed differentiation between **Memory-Mapped I/O** and **I/O-Mapped I/O** with reference to the **8085 microprocessor**:

✅ Memory-Mapped I/O vs I/O-Mapped I/O in 8085 Microprocessor

Feature / Criteria	Memory-Mapped I/O	I/O-Mapped I/O (Isolated I/O)
1. Addressing Space Used	Uses **memory address space**	Uses **separate I/O address space**
2. Instruction Set Used	All **memory-related instructions** (e.g., `LDA`, `STA`, `MOV M, A`) can access I/O devices	Only specific **I/O instructions** (`IN` and `OUT`) are used
3. Address Lines Used	All **16 address lines (A0–A15)** are used	Only **8 address lines (A0–A7)** are used
4. Addressable Range	64 KB devices (as large as memory space)	256 I/O devices (2^8 = 256 ports)
5. Control Signals Used	Uses **MEMR** and **MEMW** control signals	Uses **IOR** and **IOW** control signals
6. Data Transfer Flexibility	Can perform arithmetic/logical operations directly on I/O data	Cannot perform such operations directly on I/O data
7. Speed	Slightly **slower**, as treated like memory	**Faster**, as dedicated I/O instructions are used
8. Memory Interference	Reduces available memory space	Does **not affect** memory space
9. Interface Circuit	More complex circuitry required	Simpler interface due to dedicated signals

---

🔹 Example (8085 Assembly)

🔸 Memory-Mapped I/O:

```
LDA 2000H   ; Load data from I/O device mapped at memory location 2000H
MOV B, A    ; Move to register B
```

🔸 I/O-Mapped I/O:

```
IN 05H      ; Read data from I/O port number 05H
OUT 06H     ; Send data to I/O port number 06H
```

🔹 Conclusion:

- **Memory-mapped I/O** is powerful when you need a large number of I/O devices and wish to use all memory instructions.
- **I/O-mapped I/O** is simpler and efficient when the number of I/O devices is limited (within 256).

Let me know if you'd like a **diagram**, **Bengali version**, or **MCQ/viva questions** on this topic!

6. a) What is recursion? Write an assembly language program for calculating the factorial of 5. [4] b) Write short notes on: i) Logical Shift ii) Arithmetic Shift iii) ROL iv) ROR [8]

Here is a **complete answer** for **Question 6(a)**:

✅ 6(a) What is Recursion?

🔹 Definition:

Recursion is a programming technique where a function **calls itself** either **directly or indirectly** to solve a problem.
Each recursive call handles a **smaller sub-problem**, and the solution is built from **base case(s)** upwards.

> 🔁 It continues until it reaches a **base condition**, which stops further calls.

🔹 Example (Factorial in C):

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

✅ Assembly Language Program to Calculate Factorial of 5

Since **8086 Assembly** doesn't support recursion natively like high-level languages, we simulate recursion using the **stack**.

Here's a simple **non-recursive version** using a loop to calculate factorial of 5:

🖥️ Program: Factorial of 5 in 8086 Assembly (Non-recursive)

```
.MODEL SMALL
.STACK 100H

.DATA
   FACTORIAL_RESULT DW ?

.CODE
MAIN:
   MOV AX, @DATA
   MOV DS, AX

   MOV CX, 5       ; Counter = 5 (5!)
   MOV AX, 1       ; Initialize result = 1

FACTORIAL_LOOP:
   MUL CX          ; AX = AX * CX
   LOOP FACTORIAL_LOOP

   MOV FACTORIAL_RESULT, AX ; Store result (120 for 5!)

   ; Display Result (optional for DOS, not shown here)

   MOV AH, 4CH
   INT 21H
END MAIN
```

🔸 Explanation:

- `MOV CX, 5`: sets up a loop to multiply 5 × 4 × 3 × 2 × 1.

- `MUL CX`: multiplies `AX` by `CX` in each iteration.
- `LOOP` decreases `CX` and jumps back until `CX = 0`.
- The result (120) is stored in `FACTORIAL_RESULT`.

If you want a **recursive version using stack operations** (advanced), I can provide that too. Would you like it?
Also, let me know if you need a **Bengali version** or **flowchart** for better understanding.

Here are **short notes** on the four shift and rotate operations in assembly (mainly for **8086** or similar microprocessors):

✅ i) Logical Shift

- **Definition:** Bits in a byte or word are shifted left or right, and the **vacated bit positions are filled with 0**.
- **Types:**
  - `SHL` (Shift Left Logical): Multiplies the number by 2.
  - `SHR` (Shift Right Logical): Divides the number by 2 (for unsigned values).
- **Use:** Used mainly for **unsigned numbers**.

Example:

```
MOV AL, 00010100B ; AL = 20
SHL AL, 1         ; AL = 00101000B = 40 (logical left shift)
```

✅ ii) Arithmetic Shift

- **Definition:** Similar to logical shift, but **preserves the sign bit (MSB)** during right shift.

- **Types:**
    - `SAL` (Shift Arithmetic Left): Same as `SHL` (multiplies by 2).
    - `SAR` (Shift Arithmetic Right): Divides signed number by 2, sign bit stays the same.
- **Use:** Used for **signed numbers**.

Example:

```
MOV AL, 11100000B ; AL = -32 (in 2's complement)
SAR AL, 1         ; AL = 11110000B = -16 (arithmetic right shift)
```

✅ iii) ROL (Rotate Left)

- **Definition:** Bits are rotated left; the **MSB (bit 7)** is moved to **LSB (bit 0)**.
- **Use:** For circular rotation, used in encryption or bit manipulation.

Example:

```
MOV AL, 10010011B
ROL AL, 1         ; AL = 00100111B
```

✅ iv) ROR (Rotate Right)

- **Definition:** Bits are rotated right; the **LSB (bit 0)** is moved to **MSB (bit 7)**.
- **Use:** Useful in bitwise operations and cryptographic routines.

Example:

```
MOV AL, 10010011B
ROR AL, 1         ; AL = 11001001B
```

Let me know if you want a **diagram**, **truth table**, or **Bengali version** for these operations!

a) Draw the architecture of 8255. Describe the operating modes of 8255. [6] b) What do you mean by programmable peripheral interface (PPI)? Mention the role of 8255 PPI. [3] c) Write

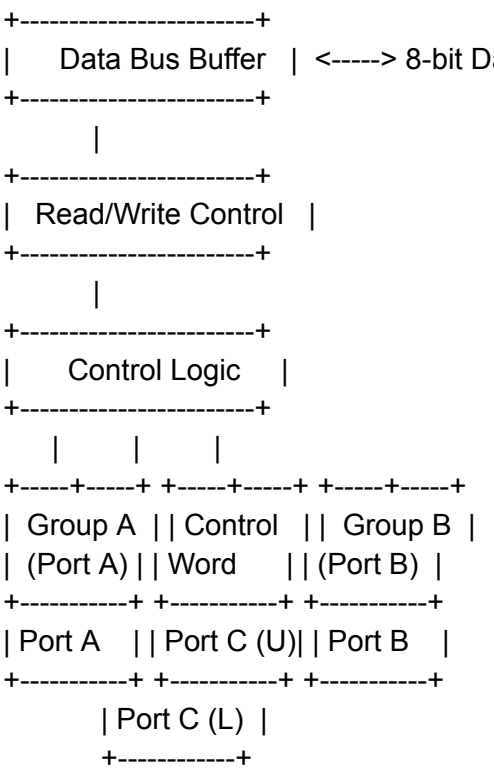
assembly code statement for each of the high level language assignments statements. [3] i) A=B*C-5/D ii) B=(A-C)*D/7

Here's a complete answer to your questions regarding the **Intel 8255 Programmable Peripheral Interface (PPI)** and corresponding assembly code translation:

---

a) Draw the Architecture of 8255 and Describe the Operating Modes

Architecture of 8255:



- **Port A**: 8-bit I/O port (can work in Mode 0, 1, or 2)
- **Port B**: 8-bit I/O port (can work in Mode 0 or 1)
- **Port C**: Split into upper (PC7–PC4) and lower (PC3–PC0); used as control/status or general I/O.
- **Control Word Register**: Stores control word to define modes/direction.
- **Read/Write Control Logic**: Interprets control signals like RD, WR, CS.

Operating Modes of 8255:

8255 has **3 main operating modes**:

1. **Mode 0 (Basic Input/Output):**
    - Simple I/O.
    - No handshaking.
    - Ports A, B, and C used for data.
    - Output latched, input not latched.
2. **Mode 1 (Strobed Input/Output):**
    - Handshaking I/O.
    - Uses Port A or B with corresponding PC lines for control and status.
    - Input: Data valid, strobe, acknowledge.
    - Output: Output buffer full, strobe, acknowledge.
3. **Mode 2 (Bidirectional Data Transfer):**
    - Only for **Port A**.
    - Bidirectional communication with handshaking.
    - Useful in inter-processor communication.

---

b) What is Programmable Peripheral Interface (PPI)? Role of 8255

Programmable Peripheral Interface (PPI):

- A **PPI** is an integrated circuit used to interface peripheral devices (like keyboards, displays, ADCs) with a microprocessor system.
- It allows the processor to **read input** from or **write output** to external devices in a programmable way.

Role of 8255 PPI:

- Acts as a **bridge between microprocessor and peripheral devices**.
- Offers **24 programmable I/O lines** grouped as Port A, Port B, and Port C.
- Supports **various data transfer modes** (simple, interrupt, handshake).
- **Reduces CPU overhead** in managing I/O operations.
- Widely used in **industrial control systems, instrumentation**, and **embedded systems**.

---

c) Assembly Code for High-Level Statements

We assume that you're writing for the **8085/8086** architecture, and all variables are stored in registers. Let's write pseudo-assembly considering general-purpose registers:

i) A = B * C - 5 / D

High-Level Statement:

```
A = B * C - 5 / D
```

Assume:

- `B → register B`
- `C → register C`
- `D → register D`
- `A → register A`

Assembly Code (8086-like Pseudocode):

```
MOV AL, B       ; Load B into AL
MUL C           ; AL = AL * C, result in AX
MOV BX, AX      ; Save result in BX

MOV AL, 5       ; Load 5
MOV AH, 0       ; Clear AH
DIV D           ; AL = 5 / D

SUB BX, AX      ; BX = (B*C) - (5/D)
MOV A, BL       ; Store final result to A
```

---

ii) B = (A - C) * D / 7

High-Level Statement:

```
B = (A - C) * D / 7
```

Assume:

- `A → register A`
- `C → register C`
- `D → register D`
- `B → register B`

Assembly Code (8086-like Pseudocode):

```
MOV AL, A       ; Load A
SUB AL, C       ; AL = A - C

MOV AH, 0       ; Clear AH for MUL
MUL D           ; AX = (A - C) * D

MOV BL, 7       ; Divisor 7
DIV BL          ; AL = result / 7

MOV B, AL       ; Store final result to B
```

Note:

- These are not exact instructions for any specific microcontroller. For 8085, MUL and DIV are not available—you would need software routines.
- This is simplified to show **logic flow** assuming high-level capabilities.

Let me know if you need 8255 pin diagram or example in 8085-specific format.

4. a) Explain how LOOP Instruction works in assembly language? Write an assembly language program for For Loop to show the "CSE Department" as reverse string. [5] b) How instruction affect the Flags: SUB AX, BX [AX=90h, BX=30h] ADD AL, BL [AL=30h, BL=70h] [2]

Here's the detailed explanation and solution for your question on **LOOP instruction, reverse string printing, and flag effects in Assembly Language**:

---

4. a) LOOP Instruction & For Loop Program

How LOOP Instruction Works:

- `LOOP` is used to repeat a block of code a certain number of times.
- It **automatically decrements** the `CX` register and **jumps** to the given label if `CX ≠ 0`.

Syntax:

LOOP label

-

The instruction sequence:

```
 MOV CX, count      ; Initialize loop counter
label:
  ; Code to repeat
  LOOP label       ; Decrements CX and loops if CX != 0
```

-

Assembly Program: Display "CSE Department" in Reverse

Assume you're using **8086 Assembly** and the string is stored in memory.

Goal: Print `"CSE Department"` in reverse:
Output: `tnemtrapeD ESC`

```
.MODEL SMALL
.STACK 100H

.DATA
msg DB "CSE Department$"      ; '$' indicates end for DOS INT 21h AH=09h

.CODE
MAIN PROC
```

```
    MOV AX, @DATA
    MOV DS, AX

    ; Find string length manually (13 chars, including space)
    MOV SI, OFFSET msg
    MOV CX, 13              ; length of string
    ADD SI, CX              ; Point SI to end of string
    DEC SI                  ; Adjust to point to last character (not '$')

PRINT_LOOP:
    MOV DL, [SI]            ; Load character
    MOV AH, 02H             ; Function to display char
    INT 21H                 ; Call DOS interrupt
    DEC SI                  ; Move to previous character
    LOOP PRINT_LOOP         ; Loop until CX = 0

    ; Exit program
    MOV AH, 4CH
    INT 21H
MAIN ENDP
END MAIN
```

---

4. b) How Instructions Affect Flags

Instruction 1: `SUB AX, BX`

Given:

- AX = `90h`
- BX = `30h`
- Operation: `AX = AX - BX = 90h - 30h = 60h`

Affected Flags:

- **Zero Flag (ZF):** 0 (Result is not zero)
- **Sign Flag (SF):** 0 (60h is positive)
- **Carry Flag (CF):** 0 (No borrow)
- **Overflow Flag (OF):** 0 (No signed overflow)

- **Auxiliary Carry (AF):** May be affected, depending on low nibble borrow
- **Parity Flag (PF):** Depends on result bits; 60h = 01100000 → even parity → PF = 1

Instruction 2: `ADD AL, BL`

Given:

- AL = `30h`
- BL = `70h`
- Operation: `AL = AL + BL = 30h + 70h = A0h`

Affected Flags:

- **Zero Flag (ZF):** 0 (A0h ≠ 0)
- **Sign Flag (SF):** 1 (A0h has MSB = 1 → negative in signed context)
- **Carry Flag (CF):** 0 (No carry from MSB)
- **Overflow Flag (OF):** 1 (Signed overflow: +48 + +112 = -96 → invalid)
- **Auxiliary Carry (AF):** May be 1 (0x0 + 0x0 + carry from bit 3)
- **Parity Flag (PF):** 1 (A0h = 10100000 → even number of 1s)

Let me know if you'd like a version for **8085**, or an **explanation with flag tables**.

a) What is physical memory? A memory location has physical address (80FD2h). In what segment it have offset BFD2h? [4] b) Discuss how an array can be declared in 8086 processor with DUP. [4] c) Explain the concept of cache memory in microprocessors. What are its advantages, and how is it implemented in the system? [4] d) What do you mean by odd address bank and even address bank? How to address them using available prominent approach. [5]