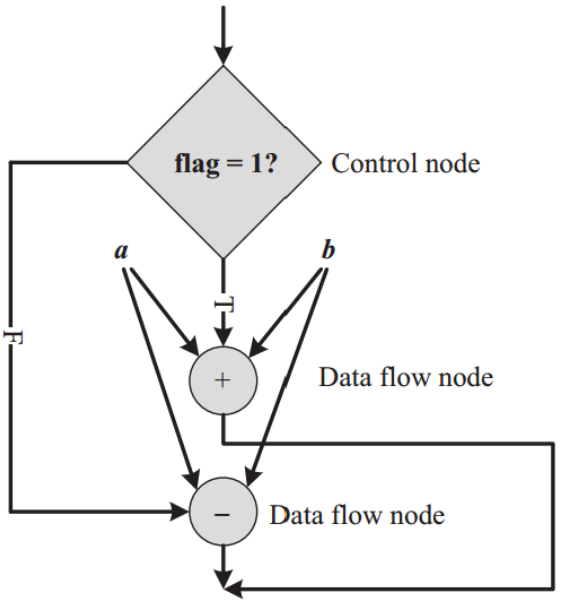
ESD IA2

M2Q1) Control Data Flow Graph/Diagram (CDFG)

A **Control Data Flow Graph (CDFG)** is a program modeling approach used in embedded system design that combines both control and data aspects of computation. It is particularly effective for applications involving **conditional execution**, where program behavior depends on certain conditions.

Key Features:

- **Includes both Data Flow and Control Flow**:
  - **Data Flow Nodes** represent operations on data (like addition, subtraction, etc.).
  - **Control Nodes** represent decision points (such as `if`, `else`, `while`), typically shown as **diamonds**.
- **Based on DFG (Data Flow Graph)**: It builds upon the DFG by adding conditional logic, allowing modeling of complex program behavior.



Control Data Flow Graph Model

- **Used for modeling concurrent process execution**.
- **Real-world application**: Digital cameras where user settings decide formats (JPEG, TIFF, etc.) is an example of a system modeled using CDFG.

The CDFG would show:

- A **control node** checking the value of `flag`.
- A **true path** leading to a data flow node performing `x = a + b`.
- A **false path** leading to a data flow node performing `y = a - b`.

M2Q2) Data Flow Graph (DFG) Model

A **Data Flow Graph (DFG)** is a graphical representation used to model the data dependencies between operations in a system or program. It is commonly used in **embedded system design**, especially for **applications with regular data processing and no conditional execution**.

Key Features of DFG:

- **Nodes (Vertices):** Represent **operations** or **functional units** (e.g., addition, multiplication).
- **Edges (Arrows):** Represent the **flow of data** (i.e., dependencies between operations).
- **No Control Flow:** DFGs model *only* data movement—not control structures like `if`, `else`, or loops.
- **Suitable For:** Systems that require continuous or repetitive data processing, such as **DSP (Digital Signal Processing)** applications.
- **Concurrency:** DFGs naturally model **parallelism**, allowing simultaneous operation of independent nodes.

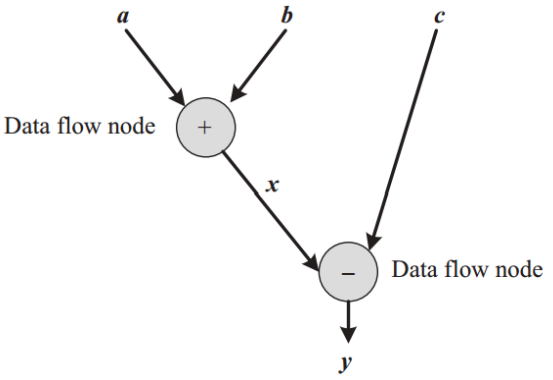
Example:

To model the operation:

`x = (a + b) * c`

The DFG would look like:

- `a` and `b` are inputs to the **addition** node.



Data Flow Graph (DFG) Model

- The result `temp` is passed to a **multiplication** node along with `c`.
- The output is `x`.

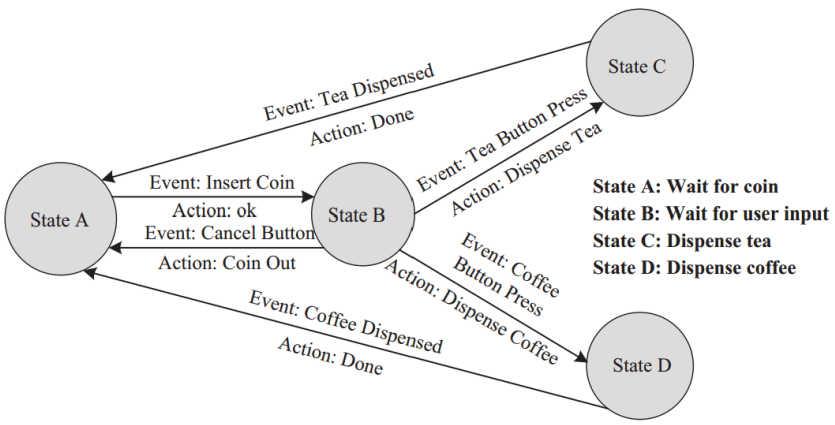
M2Q3) Automatic tea/coffee vending machine

An **automatic tea/coffee vending machine** is a real-life example of an **application-specific embedded system**. It automates the process of

dispensing beverages based on user input and is designed for efficiency, speed, and user-friendliness.

Basic Components:

1. **Input Interface:**
   - Push buttons or touch panel to select the drink (tea, coffee, black coffee, hot water, etc.)
   - Coin or card reader for payment
2. **Control Unit (Microcontroller):**
   - Takes user input and controls the sequence of operations
   - Interfaces with sensors and actuators
3. **Sensors:**
   - Water level sensor
   - Cup presence sensor
   - Temperature sensor
4. **Actuators:**
   - Valves for dispensing hot water and milk
   - Motors for powder dispensing (tea/coffee/sugar)
5. **Output Devices:**
   - LCD/LED display for instructions and status
   - Dispenser outlets for beverage delivery



FSM Model for Automatic Tea\Coffee Vending Machine

Working Process:

1. User selects a drink.
2. Payment is verified.
3. Controller checks the availability of resources (water, cup, ingredients).
4. Water is heated and ingredients are mixed as per the selected drink.
5. Final beverage is dispensed into the cup.
6. Status is updated on display (e.g., “Please collect your drink”).

Embedded System Characteristics:

- **Application-specific:** Designed only for vending beverages
- **Reactive & Real-Time:** Must respond quickly to button presses and sensor inputs
- **Autonomous:** Operates with minimal user interaction
- **User-friendly Interface**

M2Q4) FSM model for automatic seat belt warning system

An **FSM (Finite State Machine)** is a computational model used to design control logic in embedded systems. It is ideal for modeling systems that operate in **well-defined states** with **clearly defined transitions**—like an **automatic seat belt warning system** in vehicles.

Objective of the System:

To alert the driver (with a buzzer/light) if the **seat belt is not fastened** while the **ignition is ON**.

FSM Components:

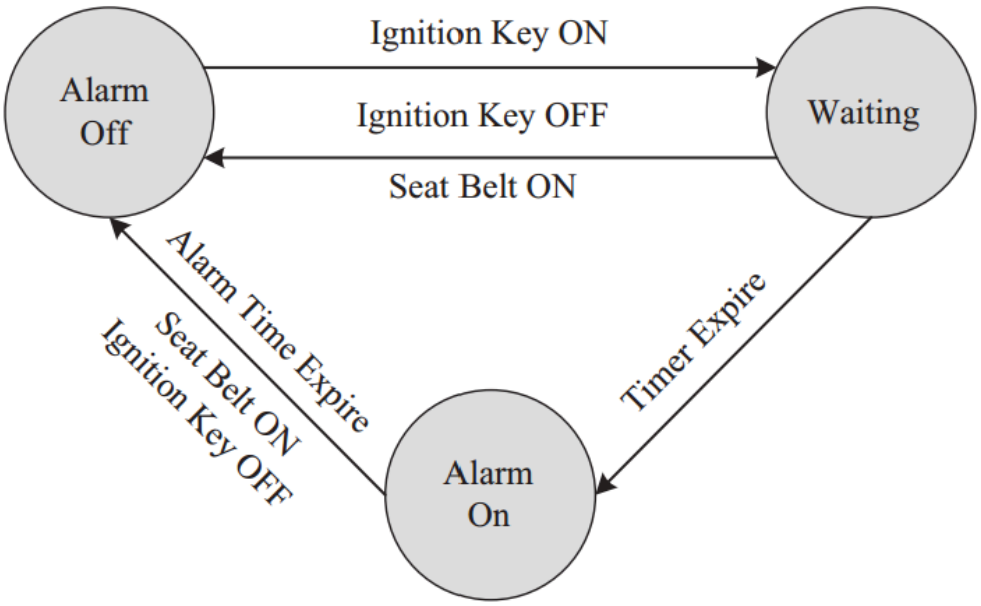
1. **States:**
    - `Idle`: Ignition OFF
    - `Seat Unbuckled`: Ignition ON & seatbelt not fastened
    - `Warning`: Buzzer ON

- `Seat Buckled`: Seatbelt fastened

2. **Inputs:**
   - Ignition status (`Ignition = ON/OFF`)
   - Seat occupancy (`Seat = Occupied/Empty`)
   - Seat belt status (`Belt = Buckled/Unbuckled`)
3. **Outputs:**
   - Buzzer (ON/OFF)
   - Indicator light (ON/OFF)

FSM Transitions:

Current State	Input Condition	Next State	Output
Idle	Ignition = OFF	Idle	Buzzer OFF
Idle	Ignition = ON, Seat occupied, Belt unbuckled	Warning	Buzzer ON
Warning	Belt buckled	Seat Buckled	Buzzer OFF
Seat Buckled	Ignition OFF	Idle	Buzzer OFF
Seat Buckled	Belt unbuckled	Warning	Buzzer ON
Warning	Seat becomes empty	Idle	Buzzer OFF



FSM Model for Automatic seat belt warning system

Embedded System Relevance:

- **Event-driven** design—perfect use case for FSM
- **Reactive system** that continuously monitors input changes
- Ensures **driver safety compliance**

M2Q5) High Level Language based Embedded firmware development technique.

In **embedded systems**, firmware is the software that runs on the microcontroller or processor to control the system's behavior.

Definition:

High-level language-based firmware development involves writing code using languages like **C**, **C++**, or **Python** (rare, for interpreted environments) instead of low-level Assembly, and compiling it using a **cross-compiler** to generate machine code for the target hardware.

Why Use High-Level Languages?

- **Improved Readability & Maintainability**
- **Faster Development Time**
- **Easier Debugging and Testing**
- **Portability across platforms**
- **Availability of Libraries & Toolchains**

Common High-Level Languages:

- **Embedded C**: Most widely used in microcontroller-based systems.
- **C++**: Used when object-oriented features are needed.
- **Python**: Limited to high-end embedded Linux systems (like Raspberry Pi).

Development Flow:

1. **Code Writing**: Firmware written in C using an IDE (e.g., Keil µVision, MPLAB X, Eclipse).

2. **Compilation**: Source code is compiled using a **cross-compiler** (e.g., GCC for ARM).
3. **Linking**: All object files are linked to create a final binary (e.g., `.hex` or `.bin`).
4. **Downloading**: Firmware is flashed onto the microcontroller using a programmer/debugger.
5. **Testing & Debugging**: Simulators or hardware debuggers are used to verify operation.

Example Code (Embedded C):

```
#include <reg51.h>

void delay() {
    int i;
    for (i = 0; i < 30000; i++);
}

void main() {
    while(1) {
        P1 = 0xFF;  // Turn ON all LEDs on Port 1
        delay();
        P1 = 0x00;  // Turn OFF all LEDs
        delay();
    }
}
```

Tools Commonly Used:

- **Keil µVision** (8051/ARM)
- **MPLAB X IDE** (PIC)
- **IAR Embedded Workbench**
- **AVR Studio**

M3Q1) Discus the Process with multi-threads with a neat diagram. Discuss the important functional and

non-functional requirements that need to be analyzed while selecting an RTOS for an embedded system design.

Process with Multi-Threads (with Diagram)

In an **RTOS-based embedded system**, a **process** can be divided into multiple **threads**, where each thread represents a unit of execution. This allows **parallelism**, **responsiveness**, and better **CPU utilization**.

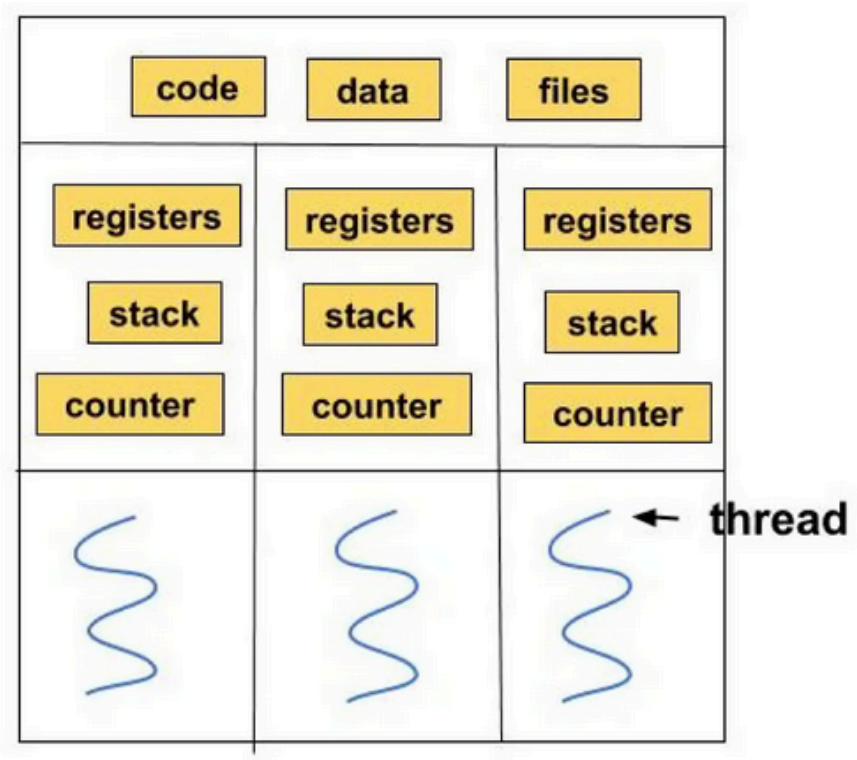
- **Process**: A running instance of a program.
- **Thread**: A lightweight unit of execution within a process. Threads share the same memory space of the process.

Advantages of Multithreading:

- Enables **concurrent execution** of tasks
- Simplifies design using **modular threads**
- Enhances **responsiveness** in real-time applications

Example:

- Thread 1: Handle audio playback
- Thread 2: Monitor touchscreen input
- Thread 3: Manage Bluetooth connectivity



Process with multi-thread

RTOS Selection Requirements

To choose the right **RTOS (Real-Time Operating System)**, both **functional** and **non-functional** requirements must be analyzed:

Functional Requirements:

1. **Task Scheduling:** Support for preemptive, priority-based, or round-robin scheduling
2. **Task Synchronization:** Semaphores, mutexes, and condition variables
3. **Inter-task Communication:** Queues, mailboxes, shared memory
4. **Interrupt Handling:** Real-time and nested interrupt support
5. **Timers & Clocks:** For time-based task execution

Non-Functional Requirements:

1. **Determinism:** Predictable response time under all conditions
2. **Memory Footprint:** Suitability for low-resource systems
3. **Portability:** RTOS should support multiple hardware platforms
4. **Scalability:** Ability to scale up for future requirements

5. **Reliability and Safety:** Crucial for mission-critical systems
6. **Toolchain and Debug Support:** Good IDE, debugger, simulator availability
7. **Licensing and Cost:** Open source vs proprietary, royalty fees

---

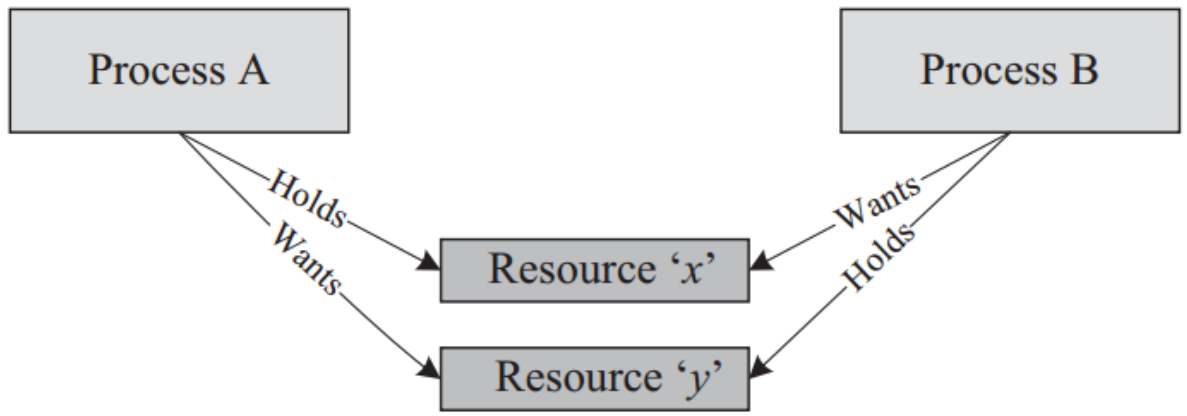
M3Q2) Explain concept of ‘deadlock’ with diagram and different conditions which favors deadlock situation.

A **deadlock** is a situation in **concurrent systems** (like RTOS-based embedded systems) where **two or more tasks** are **waiting indefinitely** for resources held by each other. As a result, none of the tasks can proceed—causing the system to halt or hang.

It typically occurs in **multi-threaded** environments where **resource sharing** and **synchronization** are involved.

Example Scenario:

- **Task A** holds **Resource 1** and waits for **Resource 2**.
- **Task B** holds **Resource 2** and waits for **Resource 1**.
- Neither can continue, resulting in a **deadlock**.



Scenarios leading to deadlock

Situations favouring deadlocks

1. **Mutual Exclusion**
    - At least one resource must be held in a **non-shareable** mode (only one task at a time).

2. **Hold and Wait**
    - A task is holding at least one resource and **waiting** to acquire additional resources held by others.
3. **No Preemption**
    - A resource **cannot be forcibly taken** away from a task. It must be **released voluntarily**.
4. **Circular Wait**
    - A **closed chain of tasks** exists, where each task holds one resource and waits for another held by the next task in the chain.

Prevention Techniques:

- **Avoid circular wait** (e.g., impose resource ordering)
- **Use timeout-based locking**
- **Deadlock detection and recovery**
- **Avoid hold-and-wait** (acquire all resources at once)

M3Q3) Embedded System Development Environment with diagram.

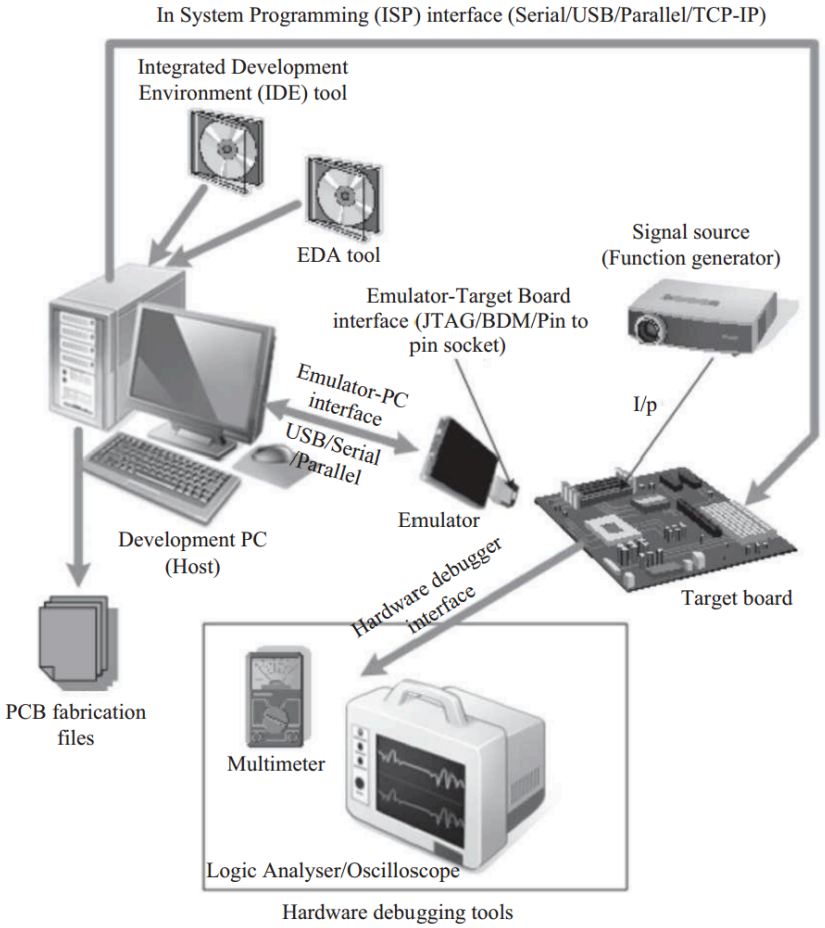
The **Embedded System Development Environment** consists of **tools and processes** required to write, compile, debug, and deploy embedded firmware onto a target hardware platform.

Key Components:

1. **Host System:**
    - The PC or workstation used by the developer to write and compile the code.
2. **Editor (IDE):**
    - Graphical environment like **Keil µVision**, **MPLAB X**, or **IAR Embedded Workbench** used for code development.
3. **Compiler / Cross-Compiler:**

- Translates high-level source code (e.g., C) into **machine code** suitable for the **target architecture** (e.g., ARM, PIC, AVR).

4. **Assembler:**
   - Converts assembly language programs into object code.
5. **Linker:**
   - Links object files and libraries to create a final executable (e.g., `.hex` or `.bin` file).
6. **Loader/Programmer:**
   - Transfers the compiled code into the **target embedded device's memory** (e.g., via JTAG, ISP, or USB).
7. **Debugger:**
   - Allows **step-by-step code execution**, **breakpoints**, and **real-time variable inspection** using hardware or simulator.
8. **Simulator/Emulator:**
   - Simulates or emulates the target hardware to test firmware without needing actual physical hardware.
9. **Target Hardware:**
   - The actual embedded board or system where the firmware will run.



Embedded System Development Environment

M3Q4) Out-of-Circuit and In System Programming (ISP) techniques

Programming techniques for embedded systems refer to the methods used to load firmware into the memory (usually Flash or EEPROM) of a microcontroller or embedded chip.

Out-of-Circuit Programming

In this technique, the microcontroller or programmable chip is **physically removed** from the target circuit board and programmed using a **separate programmer device**.

Steps:

1. Remove the chip from the socket on the PCB.
2. Insert it into a dedicated programmer.

3. Program the chip using a PC-based tool.
4. Re-insert the programmed chip back into the circuit.

Pros:

- Useful for **mass production** using programming sockets.
- Works for **simple boards** with DIP or socketed ICs.

Cons:

- **Time-consuming** for large batches or SMT packages.
- **Mechanical wear and tear** on sockets/pins.
- Cannot program in the field or after assembly.

In-System Programming (ISP):

ISP allows the microcontroller or memory chip to be programmed **without removing it** from the circuit. Programming is done using **dedicated pins** (like SPI, JTAG, or UART) while the chip remains on the PCB.

Common Interfaces:

- **SPI-based ISP** (common in AVR, some PIC)
- **JTAG/SWD** (used in ARM Cortex-M)
- **UART/Bootloader** methods

Steps:

1. Connect a programmer/debugger to the board (e.g., USBasp, ST-Link, or Atmel-ICE).
2. Provide power to the circuit.
3. Use software tools to upload firmware via ISP interface.

Pros:

- Suitable for **field updates**, **prototyping**, and **mass production**.

Cons:

- Requires **proper ISP header/interface** on the PCB.
- Programming might fail if the microcontroller's configuration is

- **No need to remove chip** from the board.
- Works well with **surface-mount devices (SMDs)**.

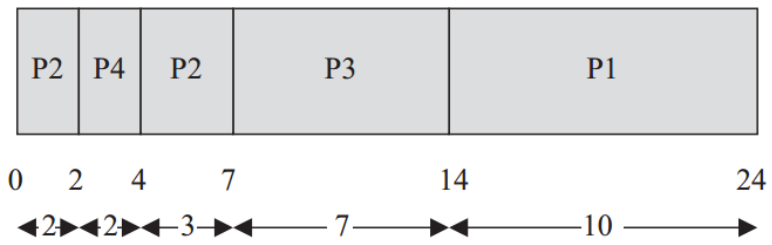
incorrect (e.g., wrong fuse bits).

Comparison Table:

Feature	Out-of-Circuit Programming	In-System Programming (ISP)
Chip removal required	Yes	No
Suitable for SMD	No	Yes
Used in production	Yes (limited)	Yes (widely)
Field updates	No	Yes
Hardware required	External programmer	Programmer/debugger + ISP port

M3Q5) Three processes—P1 (10 ms), P2 (5 ms), P3 (7 ms)—enter the ready queue at time 0 ms. After 2 ms, a new process P4 (2 ms) arrives. Using Shortest Remaining Time (SRT) scheduling (preemptive SJF), determine the order of execution and completion time of each process.

Scheduling Timeline (Gantt Chart):



Step-by-step Execution:

1. **At** `t = 0` **:**
   - Ready queue: P1(10), P2(5), P3(7)
   - SRT picks **P2** (shortest) → starts executing
   - Waiting time for **P2** : 0ms

2. **At `t = 2`:**
    - P2 has 3 ms left
    - **P4 (2 ms)** arrives → has less remaining time than P2 → **P2 is preempted**, **P4 starts**
    - Waiting time for **P4**: 0ms
        - P4 starts executing by preempting P2 since the execution time for completion of P4 (2 ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3 ms))
3. **At `t = 4`:**
    - P4 finishes
    - P2 resumes (3 ms left)
4. **At `t = 7`:**
    - P2 finishes
    - Among remaining: P3 (7 ms), P1 (10 ms) → SRT picks **P3**
    - Waiting time for **P3**: 7ms
5. **At `t = 14`:**
    - P3 finishes
    - Only P1 left → executes till t = 24
    - Waiting time for **P1**: 14ms

Turnaround Time (Completion Time - Arrival Time):

Process	Arrival	Completion	Turnaround
P1	0	24	24
P2	2	7	7
P3	0	14	14
P4	2	4	2

Average Turnaround Time:

(Turn Around Time for all Processes) / No. of Processes

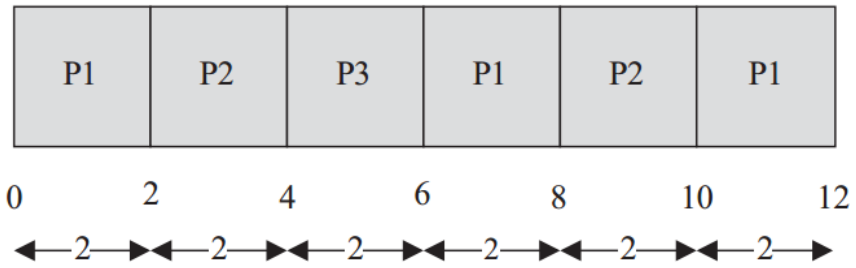
(7+2+14+24)/4 = 47/4 = 6 ms

M3Q6) Three processes—P1 (6 ms), P2 (4 ms), P3 (2 ms)—arrive at time 0 ms in the order P1, P2, P3.Using Round Robin scheduling with time quantum = 2 ms, calculate:

1) Waiting Time and Turnaround Time for each process
2) Average Waiting Time and Average Turnaround Time

(Assume only CPU-bound execution, no I/O)

Execution Timeline (Gantt Chart):



Explanation:

- **P1** runs from 0–2 (remaining 4)
- **P2** runs from 2–4 (remaining 2)
- **P3** runs from 4–6 (completes)
- **P1** runs from 6–8 (remaining 2)
- **P2** runs from 8–10 (completes)
- **P1** runs from 10–12 (completes)

Turnaround Time (TAT)

TAT = Completion Time – Arrival Time (0 for all)

Process	TAT = CT - AT
P1	12 – 0 = 12
P2	10 – 0 = 10
P3	6 – 0 = 6

Waiting Time (WT)

WT = Turnaround Time – Burst Time

Process	WT = TAT – BT
P1	12 – 6 = 6
P2	10 – 4 = 6
P3	6 – 2 = 4

Final Table:

Process	Burst Time	Completion	Turnaround Time	Waiting Time
P1	6 ms	12 ms	12 ms	6 ms
P2	4 ms	10 ms	10 ms	6 ms
P3	2 ms	6 ms	6 ms	4 ms

Averages:

- **Average Turnaround Time:** (12 + 10 + 6) / 3 = **9.33 ms**
- **Average Waiting Time:** (6 + 6 + 4) / 3 = **5.33 ms**

********** EOF **********