ESD EX

M1Q1A) Define Embedded System. Classify an embedded system based on (i) generation (ii) Complexity (iii) Triggering

Definition of Embedded System:

An **embedded system** is an **electronic/electro-mechanical system** designed to perform a **specific function** and is a **combination of hardware and firmware (software)**. It is highly specialized to its application domain and cannot be used for general-purpose computing.

Classification of Embedded Systems:

(i) Based on Generation:

Embedded systems can be classified into **four generations**, depending on technological advancements:

1. **First Generation:**
    - Built around **8-bit microprocessors** (e.g., Intel 8085, Zilog Z80).
    - Firmware written in **assembly language**.
    - Used in **digital telephones, motor controllers**.
2. **Second Generation:**
    - Based on **16-bit processors/microcontrollers**.
    - Featured better **instruction sets**.
    - Examples include **SCADA systems**, **Data Acquisition Systems**.
3. **Third Generation:**
    - Use of **32-bit processors** and **application/domain-specific ICs**.
    - Supported **pipelining, DSPs, ASICs**.
    - Used in **robotics, media devices, industrial control**.
4. **Fourth Generation:**

- Includes **System on Chips (SoC)**, **multi-core processors**, **reconfigurable processors**.
- Compact, high-performance.
- Seen in **smartphones, wearable devices, IoT systems**.

(ii) Based on Complexity and Performance:

1. **Small-Scale Embedded Systems:**
   - Simple tasks, low performance, often built on **8/16-bit microcontrollers**.
   - Example: **Electronic toys**.
2. **Medium-Scale Embedded Systems:**
   - Moderate complexity and performance.
   - Built on **16/32-bit processors**, may include **RTOS**.
   - Example: **Digital Cameras**.
3. **Large/Complex Embedded Systems:**
   - Highly complex and performance-oriented.
   - Use of **32/64-bit RISC processors**, **multi-core**, **RTOS**, **co-processors**.
   - Examples: **Flight control systems**, **encryption systems**.

(iii) Based on Triggering:

This is mainly applicable to **reactive embedded systems**, particularly in control applications.

1. **Event-Triggered Systems:**
   - Triggered by external **events** (e.g., sensor input, user commands).
   - Example: **Fire alarm systems**, **remote controls**.
2. **Time-Triggered Systems:**
   - Operations initiated based on **clock signals** or **timers**.
   - Example: **Periodic data loggers**, **sampling systems**.

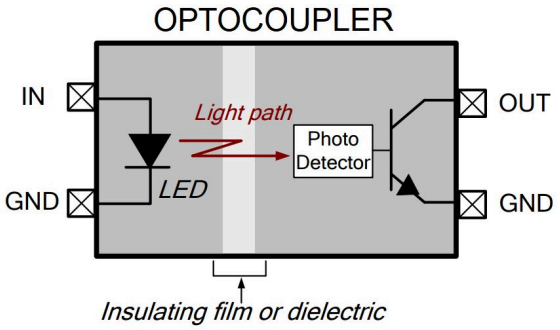
M1Q1B) Explain the following: (i) Optocoupler (ii) Zig-Bee (iii) Wi-Fi (iv) SPI bus (v) USB

(i) Optocoupler:

An **optocoupler**, also known as an **opto-isolator**, is an electronic component that **transfers electrical signals** between two isolated circuits using **light**.

- It consists of a **light-emitting diode (LED)** and a **photosensitive device** like a phototransistor.
- When the input side activates the LED, light is emitted, which is detected by the output-side phototransistor, **allowing signal transfer without physical contact**.

Purpose: Electrical isolation to protect low-voltage systems from high-voltage transients or spikes.

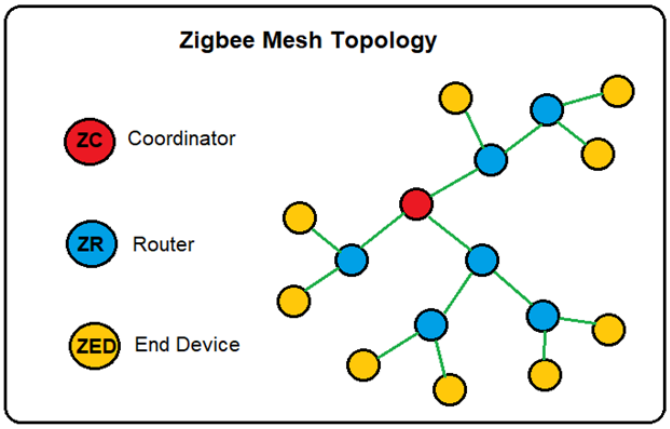


(ii) ZigBee:

ZigBee is a **low-power, low-data-rate** wireless communication technology based on the **IEEE 802.15.4** standard.

- Used for short-range wireless personal area networks (WPANs).
- Supports **mesh networking**, which enhances communication range and reliability.
- Data rate: **20 to 250 kbps**, range: typically **10–100 meters**.

Applications: Home automation, industrial control, smart lighting.



(iii) Wi-Fi:

Wi-Fi is a high-speed wireless communication technology based on **IEEE 802.11** standards.

- Enables devices to **communicate over a local area network (LAN)** or connect to the Internet wirelessly.
- Operates in **2.4 GHz or 5 GHz** bands with data rates ranging from **11 Mbps to several Gbps**.

Applications: Internet access in homes, offices, public places; multimedia streaming.

(iv) SPI Bus (Serial Peripheral Interface):

SPI is a **synchronous serial communication protocol** used to connect **microcontrollers with peripherals**.

- Uses **4 wires**: MISO (Master In Slave Out), MOSI (Master Out Slave In), SCLK (Serial Clock), and SS (Slave Select).
- It offers **full-duplex communication**, is faster than I^2C, and is suitable for **short-distance** communication.

Applications: Flash memory, SD cards, sensors.

(v) USB (Universal Serial Bus):

USB is a **standard interface** used to connect peripheral devices like keyboards, mice, printers, and storage devices to a host system like a PC.

- Supports **plug-and-play**, **hot-swapping**, and **multiple speed modes** (e.g., USB 2.0: 480 Mbps, USB 3.0+: up to 5 Gbps).
- It uses a **tiered star topology** and supports up to **127 devices**.

Applications: Data transfer, charging, communication with embedded systems.

M1Q2A) List the features of the following : (i) I2C Bus (ii) IrDA (iii) 1-wire Interface (iv) keyboard (v) UART

(i) I^2C Bus (Inter-Integrated Circuit):

- **Two-wire interface**: SDA (Serial Data), SCL (Serial Clock).
- Supports **multiple masters and slaves** on the same bus.
- **Synchronous serial communication**.
- Devices are identified by **unique addresses**.
- Data rates: **100 kbps (standard), 400 kbps (fast), up to 3.4 Mbps (high-speed)**.
- Suitable for **short-distance, intra-board communication**.

(ii) IrDA (Infrared Data Association):

- **Wireless communication** protocol using **infrared light**.
- Suitable for **short-range** line-of-sight communication (up to 1 meter).
- Data rate: typically **up to 4 Mbps**.
- Used in **remote controls, mobile phones, medical devices**.
- Immune to electrical noise but **requires clear LOS (line of sight)**.

(iii) 1-Wire Interface:

- **Single-wire bidirectional communication** (plus ground).
- Invented by **Dallas Semiconductor (now Maxim Integrated)**.
- Supports both **power and data transfer** over the same line.
- Used for **low-speed communication**, typically <16 kbps.
- Common in **temperature sensors, ID chips, RTCs**.
- **Simple and low-cost** connection system.

(iv) Keyboard:

- **Input device** typically arranged in a **matrix of rows and columns**.

- Key press detected by **scanning rows and columns**.
- Can be **mechanical, membrane-based, capacitive**, etc.
- Interfaces through **GPIO lines** or **keyboard controllers**.
- Usually **debounced in software** to avoid multiple detections.

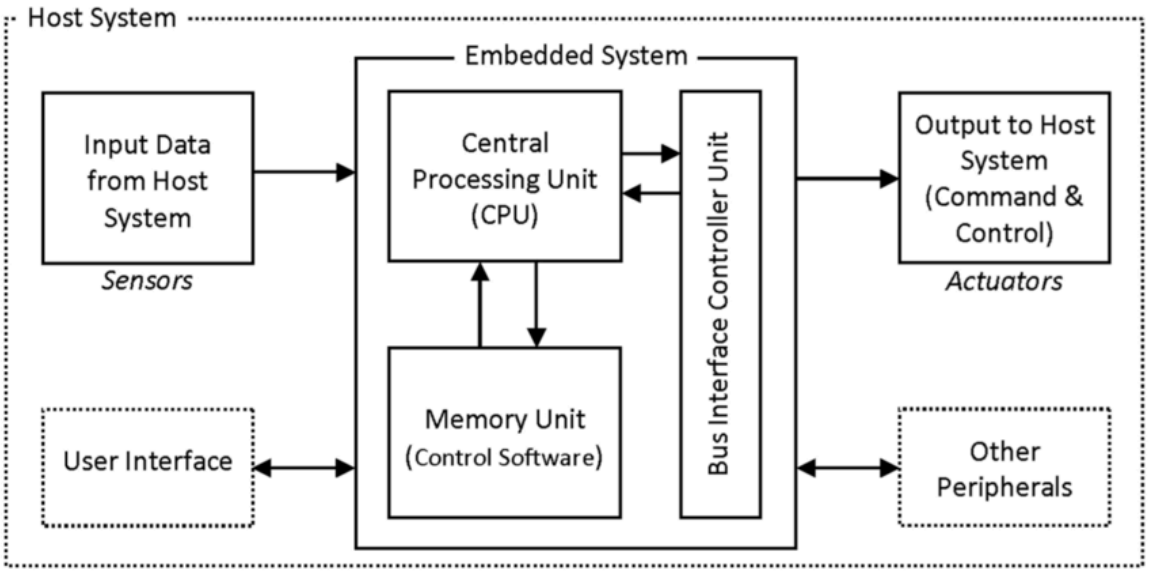
(v) UART (Universal Asynchronous Receiver Transmitter):

- **Asynchronous serial communication** protocol.
- Transmits data using **TX (Transmit)** and **RX (Receive)** lines.
- Common data format: **8 data bits, 1 start bit, 1 stop bit, optional parity**.
- Speeds typically up to **115.2 kbps** (can go higher).
- No clock line required; **uses start-stop timing**.
- Widely used for **console communication, debugging**, and device interfacing.

M1Q2B) Illustrate the architectural block diagram of embedded system and mention the components used.

1. Architectural Block Diagram of an Embedded System:

Below is a typical block diagram of an embedded system:



2. Components Used in an Embedded System:

Embedded systems are built with the following key components:

i) Processing Unit (Controller/Processor):

- Can be a **Microcontroller (e.g., 8051, PIC, AVR)**, **Microprocessor**, **SoC**, or **DSP**.
- Handles all the computational and decision-making tasks.

ii) Memory:

- **RAM**: Used for temporary data storage during execution.
- **ROM/Flash**: Stores firmware permanently.
- May include **EEPROM** for non-volatile variable storage.

iii) Sensors and Actuators:

- **Sensors**: Convert physical quantities to electrical signals (e.g., temperature, pressure).
- **Actuators**: Perform actions based on control signals (e.g., motors, LEDs, relays).

iv) I/O Interfaces:

- **Digital I/O, Analog I/O, Timers, Counters**, etc.
- Used to interact with external devices or users (e.g., switches, displays).

v) Communication Interfaces:

- **Serial protocols** like UART, SPI, I^2C, CAN, USB, ZigBee, Wi-Fi, etc.
- Enables communication with other systems or devices.

vi) Embedded Firmware:

- The **software** part programmed into ROM.
- Controls hardware, processes input, and generates appropriate outputs.

vii) Other System Components:

- **Power supply units, Reset circuits, Oscillators (Clocks)**, etc.

- Provide essential support for stable operation.

M2Q3A) Discuss the Operational and Non-Operational Quality Attributes of an Embedded System.

1. Operational Quality Attributes

These attributes describe how well the system behaves **during its normal operation**. They directly impact **user experience**, **system responsiveness**, and **performance**.

a) Response Time

- Time taken by the system to respond to an input.
- Crucial in **real-time systems** like automotive airbags or industrial control systems.

b) Throughput

- Number of tasks or operations a system can complete within a specified time.
- Higher throughput implies better performance.

c) Reliability

- Probability that the system performs correctly under specified conditions for a defined period.
- Often measured using **Mean Time Between Failures (MTBF)**.

d) Maintainability

- Ease with which the system can be maintained, debugged, and updated.

e) Security

- Protection of system data and operations from unauthorized access or modification.
- Especially important in **IoT** and **network-connected** embedded systems.

f) Safety

- Ensures that the system does not cause any harm to users or the environment.
- Essential in **medical**, **automotive**, and **aerospace** applications.

2. Non-Operational Quality Attributes

These relate to the system's **design, development, testing, and deployment** phases. They help in making the product more flexible, cost-effective, and market-ready.

a) Testability

- Ease with which system faults can be detected and corrected.
- Involves built-in self-tests, diagnostic software, etc.

b) Debuggability

- Refers to how easily bugs or errors in the system can be identified and fixed.
- Supported by **debug ports, simulation tools**, etc.

c) Evolvability

- Ability of the system to accommodate changes or enhancements in functionality.

d) Portability

- Degree to which the system or its software can be **adapted to different platforms or environments** with minimal changes.

e) Time-to-Prototype and Time-to-Market

- **Time-to-prototype**: Time taken to build an initial working model.
- **Time-to-market**: Total time taken from concept to commercial product launch.
- Faster development gives **competitive advantage**.

f) Per Unit Cost and Revenue

- Cost per unit should be optimized to maximize market reach and profits.
- Plays a critical role in **commercial viability**.

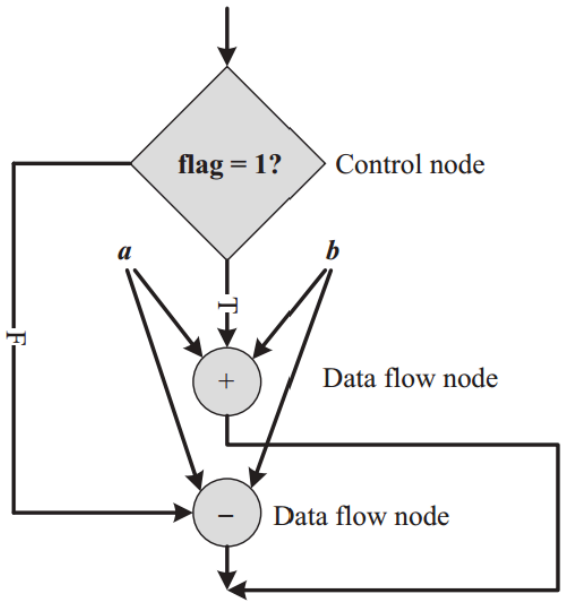
M2Q3B) Compare (i) DFG and CDFG models with an example.
(ii) C v/s Embedded C (iii) Compiler v/s Cross-Compiler.

(i) DFG vs CDFG (Data Flow Graph vs Control Data Flow Graph)

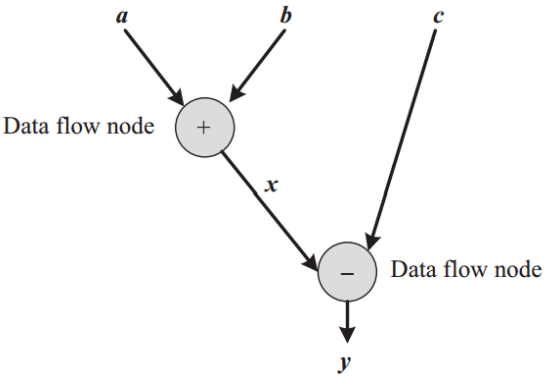
Feature	**DFG (Data Flow Graph)**	**CDFG (Control and Data Flow Graph)**
Represents	Only **data dependencies** among operations	Both **data** and **control dependencies**
Nodes	Operations or computations	Includes **operations + control constructs** (like if, loops)
Edges	Show flow of data	Show both **data flow** and **control flow**
Control Conditions	Not represented	Clearly represented
Suitability	Best for **pure data-driven applications**	Suitable for **real-world embedded programs**

Example:

For the equation: `z = a + b; x = z * c;`



Control Data Flow Graph Model



Data Flow Graph (DFG) Model

(ii) C vs Embedded C

Feature	C	Embedded C
Purpose	General-purpose programming language	Designed for **programming microcontrollers**
Standard Library Use	Fully supports std I/O, file handling, etc.	Often **limited or no standard libraries**
Target System	Works on PCs, laptops	Works on **resource-constrained embedded devices**
Hardware Access	Not designed for direct hardware access	Supports **direct port/memory access**, interrupts
Compiler Support	Uses standard compilers (e.g., GCC)	Uses **cross-compilers** and **device-specific compilers**

(iii) Compiler vs Cross-Compiler

Feature	Compiler	Cross-Compiler
Definition	Converts source code to machine code **for same platform**	Converts source code for **a different target platform**
Host = Target?	**Yes**	**No** – Target is different from host

Feature	**Compiler**	**Cross-Compiler**
Use Case	Used in PC application development	Used in **embedded system development**
Example	GCC for x86 running on a PC	ARM cross-compiler generating code for microcontroller
Hardware Dependency	Minimal	Highly **hardware-specific**

M2Q4A) Explain the assembly language based embedded firmware development with a diagram and mention its advantages and disadvantages.

Assembly Language-Based Firmware Development

In embedded systems, **firmware** is the low-level software written to control hardware. One way to develop firmware is using **Assembly Language**, which offers **direct control over hardware** and is specific to the target processor.

Assembly language uses **mnemonics** to represent machine instructions, allowing programmers to write **efficient**, **hardware-optimized** code for microcontrollers and processors.

1. **Editor Program**
    - **Purpose**: To write source code in assembly language.
    - **Output**: `myfile.asm` (the assembly source file).
    - This file contains human-readable instructions for the microcontroller.
2. **Assembler Program**
    - **Purpose**: Converts the assembly code into machine-readable object code.
    - **Inputs**: `myfile.asm`
    - **Outputs**:
        - `myfile.obj` (object file): Contains machine code.
        - `myfile.lst` (list file): Optional file showing source code along with addresses and opcodes for debugging.
3. **Linker Program**

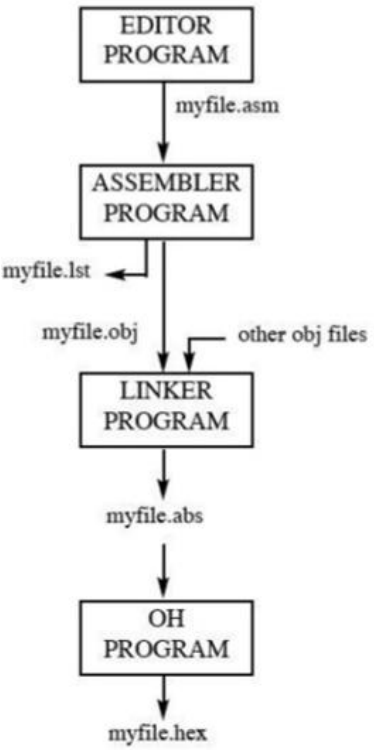
- **Purpose**: Combines the object file with any other necessary object files or libraries.
- **Inputs**:
  - `myfile.obj`
  - `other obj files` (if needed)
- **Output**: `myfile.abs` (absolute file): A fully linked machine code file with fixed addresses.

4. **OH Program (Object to Hex Converter)**
   - **Purpose**: Converts the absolute file into a hex file.
   - **Input**: `myfile.abs`
   - **Output**: `myfile.hex` — This file is in Intel HEX format, used to program the microcontroller.

Block Diagram of Assembly Language-Based Development Flow

Advantages of Assembly Language Firmware

1. **Highly efficient and fast** – Optimized for execution speed and memory usage.
2. **Full control over hardware** – Direct access to processor instructions and peripherals.
3. **Minimal memory footprint** – Useful for memory-constrained systems.
4. **Deterministic behavior** – Important for real-time systems.



Disadvantages of Assembly Language Firmware

1. **Difficult to learn and debug** – Low-level syntax and lack of abstraction.
2. **Poor portability** – Code is tied to a specific processor architecture.
3. **Time-consuming** – Longer development time compared to high-level languages like Embedded C.
4. **Hard to maintain** – As code size grows, readability and manageability decrease.

M2Q4B) Demonstrate coin operated telephone system with a FSM, function of states and state transition diagram.

(i) FSM (Finite State Machine) for Coin Operated Telephone System

An FSM consists of:

- **States** (A to I)
- **Events** that trigger transitions
- **Actions** that occur during transitions

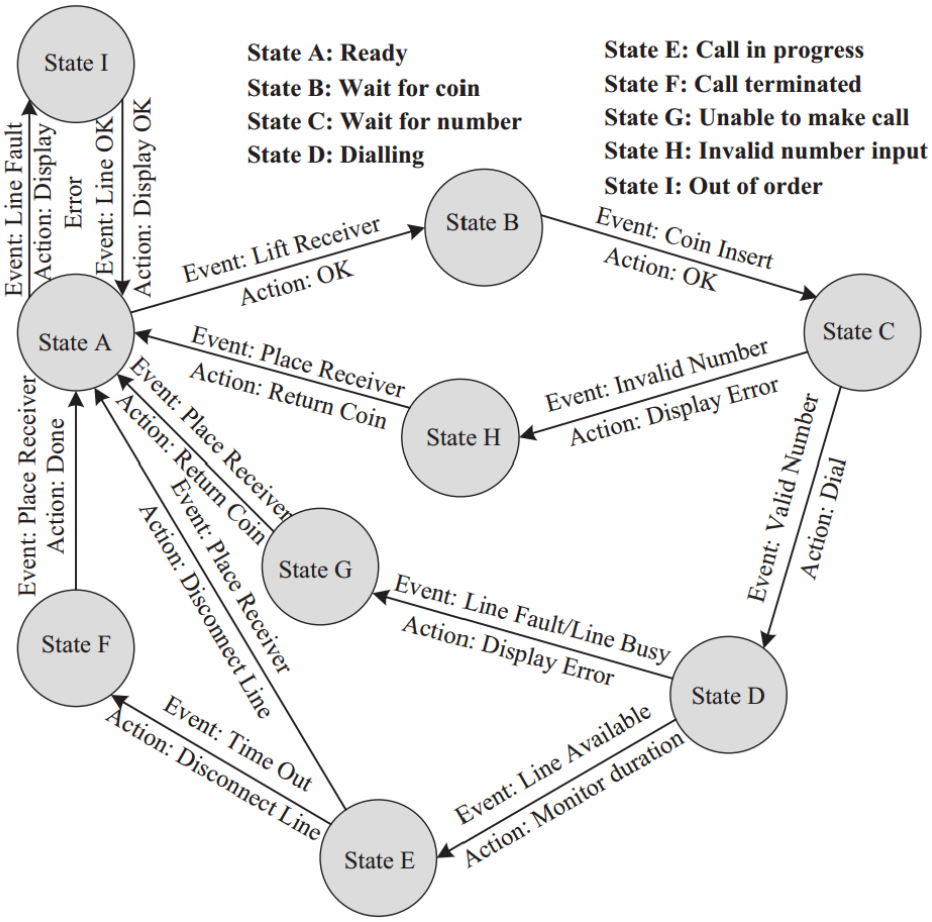
The system transitions from one state to another based on user interactions and system conditions such as coin insertion, dialing a number, or call termination.

(ii) Function of States

State	Name	Function
A	Ready	System is idle. Waiting for receiver to be lifted.
B	Wait for Coin	After receiver is lifted, system waits for a coin to be inserted.
C	Wait for Number	After coin insertion, system waits for user to dial a number.
D	Dialing	Number is being dialed; system checks its validity.
E	Call in Progress	Valid number; call is connected and monitored.
F	Call Terminated	Call ended either by user or timeout. System prepares to reset.

State	Name	Function
G	Unable to Make Call	Call cannot be placed (e.g., line fault or busy); coin returned.
H	Invalid Number Input	Number dialed is invalid; error displayed and coin returned.
I	Out of Order	System fault (e.g., line fault at initial state); error displayed.

(iii) State Transition Diagram



FSM Model for Coin Operated Telephone System

M3Q5A) Explain monolithic and micro kernels with suitable example for each. (6M)

1. Monolithic Kernel:

A monolithic kernel includes **all essential operating system services within the kernel space**. This means that all the kernel modules run in the same address space, usually under a **single kernel thread**.

- **Features:**
  - All services like memory management, file system, process scheduling, device drivers, etc., operate inside the kernel.
  - Tightly integrated components allow **efficient hardware utilization**.
  - However, any fault in one module can crash the entire system, making it **less robust**.
- **Advantages:**
  - Faster execution due to direct system call access.
  - Better performance in systems where **efficiency is critical**.
- **Disadvantages:**
  - Difficult to modify or extend.
  - Low fault tolerance—errors in one component can impact the whole system.
- **Examples:** Linux, Solaris, MS-DOS.

2. Microkernel:

In a microkernel architecture, only the **essential services** such as **memory management, process management, interrupt handling, and timer system** are kept in the kernel space. Other services like file systems, device drivers, and network stacks are implemented in **user space as separate programs** called *servers*.

- **Features:**
  - Promotes **modularity** and **system stability**.
  - If a service fails, only the specific server needs restarting—not the entire system.
- **Advantages:**
  - **Highly robust** and secure—crashes in user space don't affect kernel space.

    - Supports easier debugging and dynamic configuration.
- **Disadvantages:**
    - Slightly slower due to **inter-process communication (IPC)** between user space and kernel space.
- **Examples:** QNX, Minix 3, Mach.

M3Q5B) Discuss the terms tasks, process and threads. (8M)

1. Process

A **process** is an **independent program in execution**, with its own memory space and system resources.

- Each process has:
    - A **process control block (PCB)** that stores its state and attributes.
    - Its own **address space**, **code**, **data**, and **stack**.
- Processes do not share memory or resources directly with each other (unless via IPC).

Example:

In an embedded system running a camera application, one process may handle **image capture**, while another handles **image storage**.

2. Task

In **embedded and real-time systems**, a **task** is a lightweight unit of execution similar to a process, but often used **interchangeably with "thread"** depending on the RTOS.

- Tasks:
    - Are **scheduled** by the RTOS.
    - Can have **priorities**.
    - Often share the same **code space**, and may share **data**.
- Tasks are central to RTOS-based systems where multiple tasks run concurrently to handle I/O, processing, etc.

Example:

In a robotic control system:

- **Task 1**: Monitor sensor input.
- **Task 2**: Motor control.
- **Task 3**: Communication with a host PC.

3. Thread

A **thread** is the smallest unit of CPU execution within a process.

- Multiple threads can run **within a single process**.
- Threads **share**:
  - The same **memory space**.
  - Code, data, and system resources of the parent process.
- Each thread has its own **program counter**, **stack**, and **registers**.

Advantages of Threads:

- Efficient communication (shared memory).
- Lower overhead than processes.
- Faster context switching.

Example:

A media player might have:

- One thread for **decoding audio**,
- Another for **updating the UI**,
- Another for **fetching data from storage**.

Comparison Table:

Feature	**Process**	**Task**	**Thread**
Memory Space	Separate	Shared or independent (RTOS-specific)	Shared with parent process
Overhead	High	Medium	Low
Communication	Through IPC	Message queues/semaphores	Shared memory

Feature	**Process**	**Task**	**Thread**
Usage	General-purpose OS	Real-time embedded systems	Multi-threaded applications

M3Q5C) Three processes with process IDs P1, P2 and P3 with estimated completion time 10,5,7 ms respectively enter the ready queue together in order P1, P2 , P3. Calculate waiting time and turn around time for each process and average waiting time and TAT (assume there is no I/O waiting for the processes). (6M)

M3Q6A) Write a note on IAP [In Application Programming] and In System Programming. (4M)

1. In-Application Programming (IAP):

- **IAP** refers to the process of **modifying or upgrading the firmware** of a microcontroller **while the application is running**.
- The system has built-in routines that allow the application to **reprogram its own Flash memory**.
- No need to remove the chip or halt system execution.
- Often used in systems requiring **remote firmware updates** (e.g., via UART, USB, Ethernet).

Example:

An IoT device receiving a firmware patch over-the-air (OTA) and writing it to Flash while still running its communication stack.

2. In-System Programming (ISP):

- **ISP** allows firmware to be programmed **without removing the chip** from the system board.

- Programming is done using an external device (e.g., a **serial programmer**, **JTAG**, or **USB bootloader**) connected to a **dedicated interface**.
- Typically used during development or production testing.

Example:

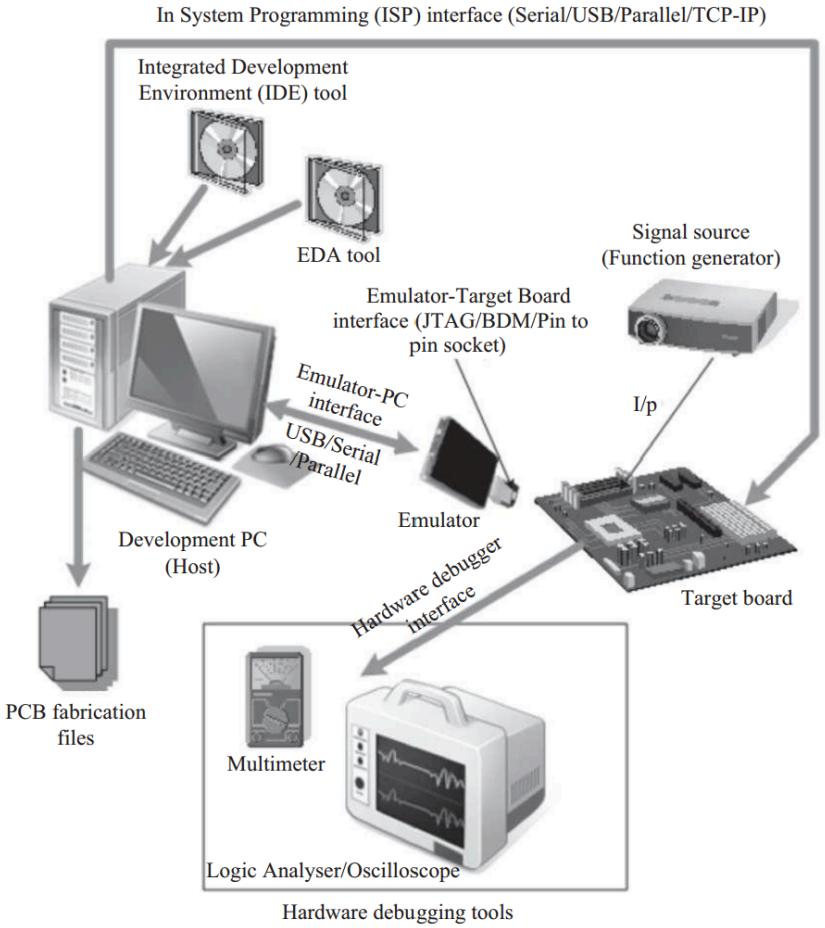
Using a USB bootloader to update firmware of an 8051 microcontroller on the production board.

Difference at a Glance:

Feature	**IAP**	**ISP**
Performed by	Application itself	External programmer
Chip Removal	Not required	Not required
System Running?	Yes, firmware updated while system runs	Typically done in boot/reset mode
Use Case	Remote updates, field upgrades	Initial programming, production setup

M3Q6B) Demonstrate a block schematic of IDE environment for embedded system design and explain their functions in brief. (8M)

An **Integrated Development Environment (IDE)** is a software tool used to **develop, compile, simulate, debug, and program embedded firmware**. It brings together all development tools under one graphical interface, streamlining embedded system development.



Embedded System Development Environment

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.

M3Q6C) Illustrate the concept of ‘deadlock’ with a neat diagram. Mention the different conditions which favors a deadlock situation. (8M)

What is Deadlock?

A **deadlock** is a situation in a **multiprogrammed or multitasking system** where two or more tasks are **permanently blocked**, waiting for resources **held by each other**, and none can proceed.

It occurs in systems that allow **concurrent access to shared resources**, such as memory, semaphores, files, or I/O devices.

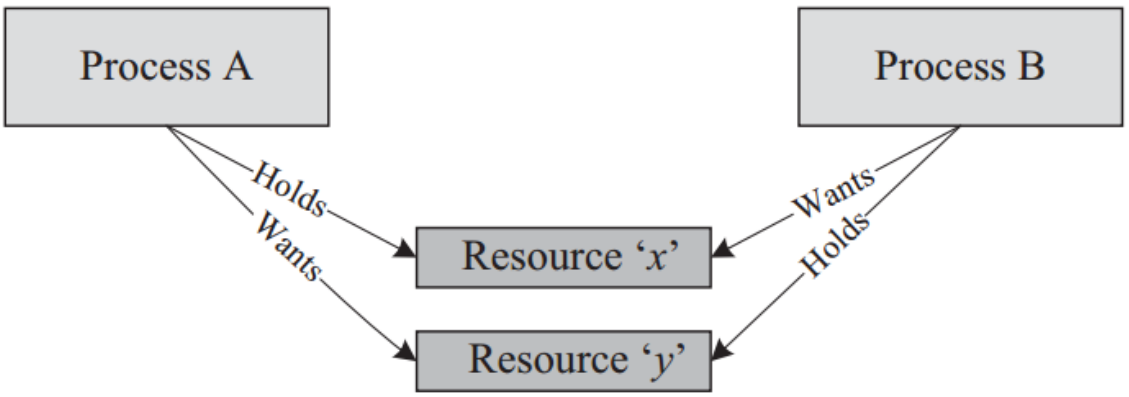
Example Scenario:

Let’s say:

- **Task A** holds **Resource 1** and waits for **Resource 2**.
- **Task B** holds **Resource 2** and waits for **Resource 1**.

Both tasks are now in a state of **circular wait**, and the system **cannot recover without intervention**.

Deadlock Diagram:



Scenarios leading to deadlock

Conditions Favoring Deadlock (Coffman's Conditions):

1. **Mutual Exclusion**

   At least one resource must be held in a **non-sharable** (exclusive) mode.

2. **Hold and Wait**

   A task is holding at least one resource and **waiting for additional resources** held by other tasks.

3. **No Preemption**

   Resources **cannot be forcibly removed**; they are released **voluntarily** by the task holding them.

4. **Circular Wait**

   A set of processes exists such that each process is **waiting for a resource held by the next**, forming a **cycle**.

M4Q7A) List the different registers of ARM CORTEX – M3 and mention their use. Explain the use of R13, R14 and R15 registers. (8M)

Registers in ARM Cortex-M3:

ARM Cortex-M3 uses a **register-based architecture** with **16 general-purpose registers** (R0–R15) and several **special registers**.

General-Purpose Registers (R0 to R12):

Register	Use
R0–R3	General-purpose registers for temporary data and parameter passing.
R4–R11	Callee-saved registers (preserved across function calls).
R12	Intra-procedure-call scratch register (can be used freely).

Special Registers:

Register	Name	Use
R13	**SP (Stack Pointer)**	Points to the current top of the stack. Two stack pointers exist: **MSP (Main Stack Pointer)** and **PSP (Process Stack Pointer)**.
R14	**LR (Link Register)**	Holds the **return address** of a function call. Used by `BL` (branch with link) instruction.
R15	**PC (Program Counter)**	Holds the **address of the next instruction** to be executed. Auto-increments after each fetch.

Explanation of R13, R14, and R15:

R13 – Stack Pointer (SP):

- Points to the **top of the current stack**.
- Used for **function calls**, **interrupt handling**, and **context switching**.
- Cortex-M3 supports:
  - **MSP** (Main Stack Pointer): Used in Handler mode (e.g., interrupts).
  - **PSP** (Process Stack Pointer): Used in Thread mode (e.g., user applications).

R14 – Link Register (LR):

- Stores the **return address** when a function is called using `BL`.
- On function exit, `BX LR` is used to return to the caller.
- In exception handling, LR holds special values to determine the return behavior.

R15 – Program Counter (PC):

- Always holds the **address of the current instruction** being executed.
- Auto-increments (by 2 or 4 bytes depending on instruction size).
- Used in branching, jumping, and interrupt vector fetching.

Summary Table: Ref.

Register	Alias	Function
R13	SP	Points to top of stack (MSP/PSP)
R14	LR	Holds return address for function calls
R15	PC	Points to next instruction to execute

M4Q7B) Summarize the CPSR configuration. Illustrate how to access different subdivisions of PSR. (6M)

What is CPSR?

CPSR stands for **Current Program Status Register**. It holds key information about the current **state of the processor**, including:

- **Flags** (Negative, Zero, Carry, Overflow)
- **Interrupt status**
- **Processor mode**
- **Execution state**

CPSR Configuration (Bit Fields):

Bit(s)	Field	Description
31	**N (Negative)**	Set if the result of an operation is negative.
30	**Z (Zero)**	Set if the result is zero.
29	**C (Carry)**	Set if there's a carry out or borrow.
28	**V (Overflow)**	Set if arithmetic overflow occurs.
27–8	**Reserved**	Reserved for future use.
7	**I (IRQ Disable)**	Disables IRQ interrupts when set.
6	**F (FIQ Disable)**	Disables FIQ interrupts when set.

Bit(s)	Field	Description
5	**T (Thumb State)**	Set if processor is in Thumb instruction set.
4–0	**M[4:0] (Mode)**	Specifies the processor mode (e.g., user, FIQ, IRQ).

Accessing PSR Fields

In **Cortex-M3**, PSR fields can be accessed using **system instructions or inline assembly**.

Example in ARM Assembly:

```
MRS R0, APSR  ; Read APSR (application status)
MRS R1, IPSR  ; Read IPSR (interrupt number)
MRS R2, EPSR  ; Read EPSR (execution status)
```

- `MRS`: Move Register from System Register (reads PSR value)
- You can also write to PSR using `MSR` (Move to System Register)

M4Q7C) Explain exceptions and interrupts of ARM CORTEX – M3. (6M)

Definition:

- In **ARM Cortex-M3**, **exceptions** are events that interrupt the normal execution flow of a program.
- **Interrupts** are a type of exception triggered by **hardware events** (e.g., timer overflow, GPIO change).
- All exceptions are handled through a **vector table** and controlled by the **NVIC** (Nested Vectored Interrupt Controller).

Categories of Exceptions:

System Exceptions (Fixed)

These have fixed exception numbers and are built into the core.

Exception Name	Purpose
Reset	Triggered during power-on or reset signal.
NMI (Non-Maskable Interrupt)	High-priority, cannot be disabled.
Hard Fault	Critical fault like illegal memory access.
MemManage Fault	Memory protection violation.
Bus Fault	Failed memory access (e.g., fetch/store).
Usage Fault	Caused by undefined instructions or divide-by-zero.
SVCall	Software-triggered system service call.
PendSV	Used for context switching in RTOS.
SysTick	System timer interrupt for periodic tasks.

External Interrupts (IRQs)

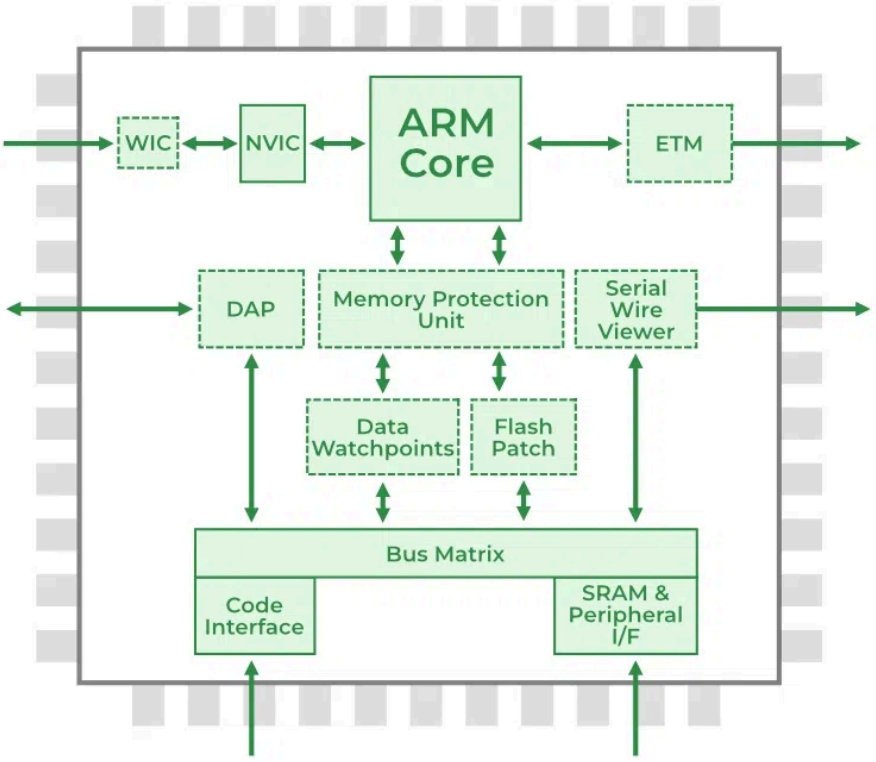
- Triggered by **external peripherals**.
- Cortex-M3 supports up to **240 IRQs**.
- Controlled and prioritized via **NVIC**.

Interrupt Handling Process:

- When an exception or interrupt occurs:
  1. **Processor saves the current context** (registers) on the stack.
  2. **Jumps to ISR** (Interrupt Service Routine) from vector table.
  3. Executes ISR.
  4. Returns using special instruction (`BX LR`), restoring context.

M4Q8A) Describe with a block schematic, explain the function of various units in ARM Cortex M3 processor architecture in brief. (10M) [OLD MQP]

Block Diagram of ARM Cortex-M3 Architecture:



Key Units and Their Functions:

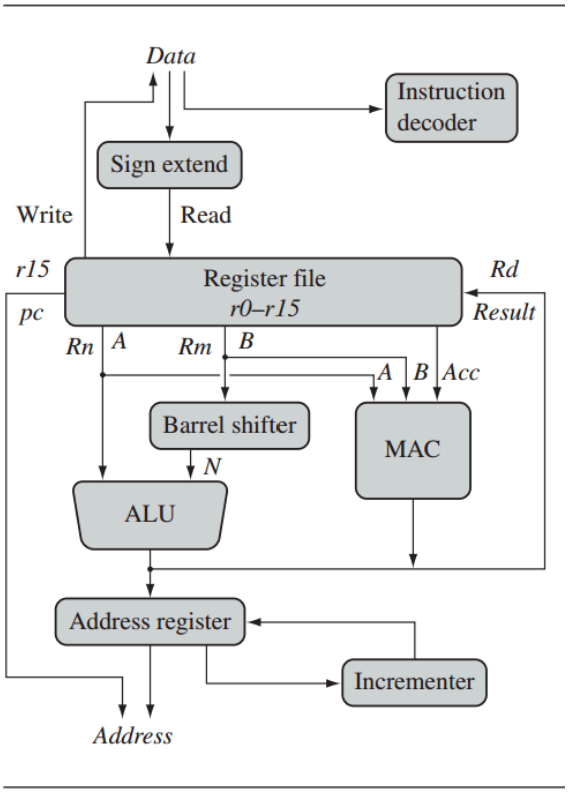
Unit	Function
1. Cortex-M3 Core	The central unit of the processor that executes instructions using a 3-stage pipeline (Fetch, Decode, Execute).
2. ALU (Arithmetic Logic Unit)	Performs all arithmetic and logical operations. Integrated with a **barrel shifter** for efficient bitwise operations.
3. Register File	Contains **16 general-purpose registers (R0–R15)** and **special registers** (e.g., Program Counter, Stack Pointer).
4. Control Unit	Manages the instruction cycle and pipeline execution. Controls program flow, branching, and exception handling.
5. Program Counter (R15)	Holds the address of the current or next instruction to be executed.
6. Stack Pointers (R13)	Cortex-M3 supports both **Main Stack Pointer (MSP)** and **Process Stack Pointer (PSP)** for context switching and task management.
7. NVIC (Nested Vectored Interrupt Controller)	Handles up to 240 interrupts with **hardware-prioritized nesting** and **low-latency response**.

Unit	Function
8. Bus Interface Unit	Connects the processor to **AMBA buses (AHB, APB)** to communicate with memory and peripherals.
9. Memory Interface	Facilitates access to Flash, RAM, and peripheral memory regions via the **Harvard Architecture** (separate instruction/data paths).
10. Debug Unit	Provides debugging capabilities using **JTAG or Serial Wire Debug (SWD)** interfaces. Supports **breakpoints and watchpoints**.

M4Q8A) Describe ARM core dataflow model with a neat and labeled diagram. Explain how data flows through various components of ARM core during instruction execution. (10M)

The ARM core dataflow model illustrates the **path taken by data** as it moves through the processor during **instruction execution**. This model highlights the interaction between the **register file**, **ALU**, **barrel shifter**, **MAC unit**, and **memory interfaces**, which are essential in understanding ARM's efficiency and pipelined operation.

Diagram



ARM core dataflow model.

Dataflow Explanation During Instruction Execution

1. **Instruction Decode:**
   - Instructions fetched from memory are sent to the **Instruction Decoder**, which identifies the opcode, operands, and operation type.
2. **Operand Fetch (Register File):**
   - The source operands `Rn` and `Rm` are read from the **register file (r0–r15)**.
   - `Rn` is sent directly to the **ALU**.
   - `Rm` is sent to the **Barrel Shifter** or **MAC Unit** based on the instruction.
3. **Barrel Shifter (optional path):**
   - Performs shifts or rotates on `Rm` in the same instruction cycle before it enters the ALU.
   - Output `N` is sent to the ALU.
4. **Arithmetic Logic Unit (ALU):**
   - Receives `Rn` and the output from the barrel shifter.

- Executes arithmetic/logical operations and updates status flags.
- Sends the result to:
  - The **destination register Rd**, or
  - The **Address Register**, if it's a memory instruction.

5. **Multiply-Accumulate (MAC) Unit:**
   - Accepts inputs `A`, `B` and an optional accumulator `Acc`.
   - Produces multiply-accumulate results used in DSP operations.
6. **Sign Extend Unit:**
   - Used in instructions involving immediate values or load/store with offsets.
   - Converts smaller width values to 32-bit for correct operation.
7. **Address Register and Incrementer:**
   - Used for computing and incrementing memory addresses during load/store instructions.
   - Final address is passed to memory via the address bus.
8. **Result Write-back:**
   - Final results are written back to the **destination register Rd** in the register file.

M4Q8B) Discuss any 5 applications of ARM cortex M3 based on its features. (5M)

The **ARM Cortex-M3** processor is widely used in embedded systems due to its **low power consumption**, **high performance**, **rich instruction set**, and **real-time capabilities**. Below are five key applications:

1. Industrial Automation Systems

- Used in **motor control**, **sensor interfacing**, and **process control** systems.
- Cortex-M3 supports **real-time responsiveness**, **priority-based interrupts**, and **low interrupt latency** via NVIC, making it ideal for automation tasks.

2. Consumer Electronics

- Found in **digital cameras**, **printers**, **washing machines**, and **smart TVs**.
- Its **low power** and **high integration** reduce overall system cost and improve efficiency.

3. Automotive Applications

- Used in **airbag systems**, **engine control units (ECUs)**, **infotainment systems**, and **dashboard electronics**.
- The processor's **fault handling**, **low interrupt latency**, and **robust exception management** are critical for safety and reliability.

4. Medical Devices

- Deployed in **portable health monitors**, **insulin pumps**, and **smart inhalers**.
- Offers **low power operation** and **deterministic behavior**, essential for life-critical embedded applications.

5. Internet of Things (IoT)

- Used in **smart home devices**, **wearables**, and **connected sensors**.
- The Cortex-M3's **compact code footprint**, **energy efficiency**, and **support for wireless stacks** make it ideal for **IoT edge nodes**.

M4Q8C) Explain all the processor modes of ARM Cortex M3 along with a diagram. (5M)

Processor Modes in ARM Cortex-M3:

ARM Cortex-M3 operates in **two main processor modes**, used to separate **privileged system-level tasks** from **application-level tasks**. This improves security, stability, and task management in embedded systems.

1. Thread Mode

- **Default mode** after reset.
- Used to execute **application-level code**.
- Can run in:
  - **Privileged mode** (full access)

- **Unprivileged mode** (restricted access)

- Mode can be changed using control register (`CONTROL`).

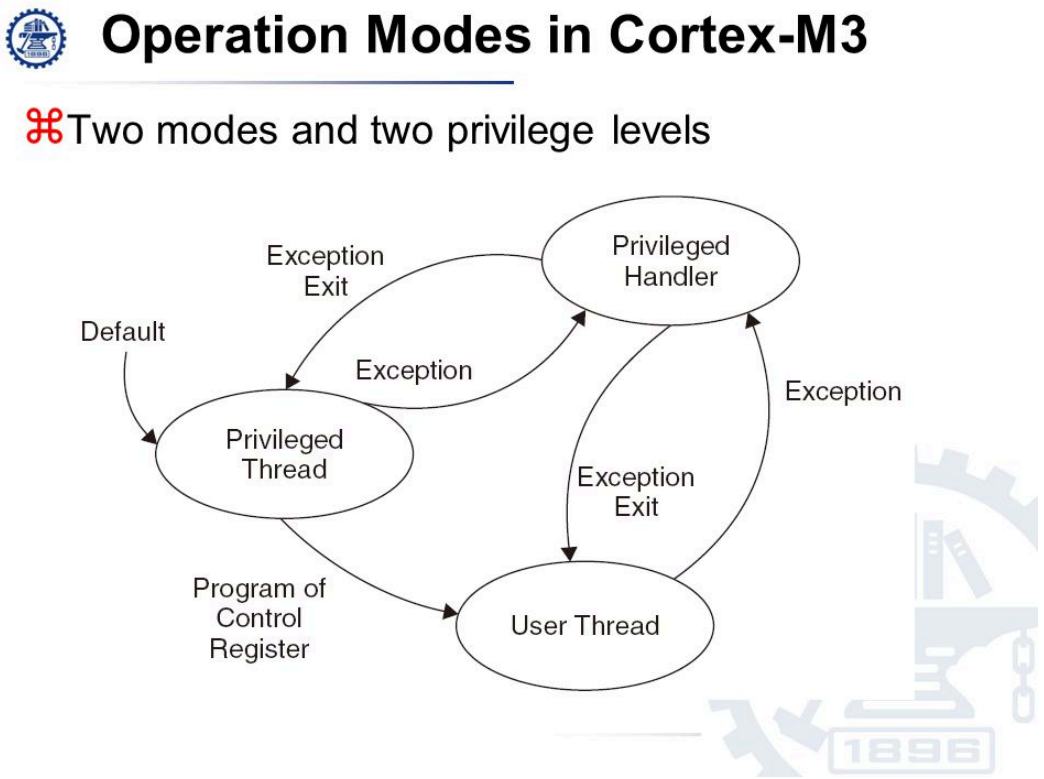
Use Case: Running user tasks in an RTOS.

2. Handler Mode

- Entered automatically when an **exception or interrupt occurs**.
- Always runs in **privileged mode**.
- Executes **Interrupt Service Routines (ISRs)** and **fault handlers**.
- Uses **Main Stack Pointer (MSP)** by default.

Use Case: Handling SysTick, IRQs, or system faults like hard faults.

Diagram: ARM Cortex-M3 Mode Switching



M5Q9A) Explain the following 32 bit instructions with an example for each : ADC, BIC, LSL and PUSH. (8M)

1. ADC – Add with Carry

- **Function:** Adds two values and the **carry flag**.
- Commonly used in **multi-word arithmetic** (e.g., 64-bit addition using 32-bit registers).

Syntax:

```
ADC Rd, Rn, Rm
```

- `Rd = Rn + Rm + Carry`

Example:

```
ADC R1, R2, R3
```

- Adds `R2 + R3 + Carry` and stores the result in `R1`.

Use Case:

Multi-byte number addition when previous operation sets the Carry flag.

2. BIC – Bit Clear (AND with NOT)

- **Function:** Performs **bitwise AND** between the first operand and the **bitwise NOT of the second operand**.
- Used to **clear specific bits** in a register.

Syntax:

```
BIC Rd, Rn, Rm
```

- `Rd = Rn AND NOT(Rm)`

Example:

```
BIC R0, R0, #0x01
```

- Clears bit 0 of `R0`.

Use Case:

Used in masking operations where specific bits need to be reset.

3. LSL – Logical Shift Left

- **Function:** Shifts bits of a register value to the **left by a given number of bits**, filling with zeros on the right.
- Used for **multiplying by powers of two**.

Syntax:

```
LSL Rd, Rm, #shift_amount
```

- `Rd = Rm << shift_amount`

Example:

```
LSL R1, R2, #2
```

- Shifts the value in `R2` left by 2 bits and stores in `R1`.

Use Case:

Efficient multiplication by 2, 4, 8, etc.

4. PUSH – Push Register(s) to Stack

- **Function:** Stores multiple registers onto the stack in one instruction.
- Stack grows **downward** (i.e., towards lower memory addresses).

Syntax:

```
PUSH {R0-R3, LR}
```

- Saves R0, R1, R2, R3, and LR on the stack.

Example:

```
PUSH {R4, R5}
```

- Pushes contents of R4 and R5 onto the stack.

Use Case:

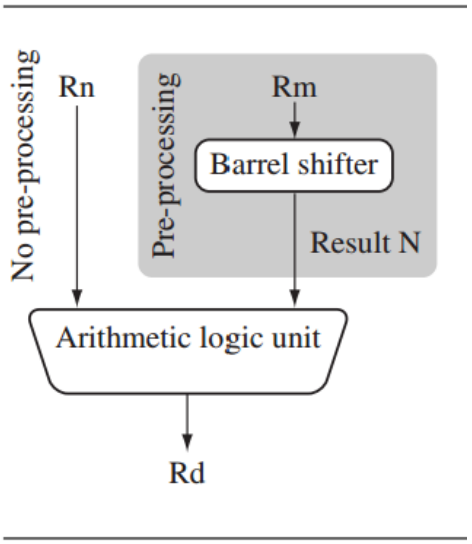
Used at the beginning of subroutines or ISRs to save context.

M5Q9B) Describe the barrel shifter and ALU unit with diagram and explain the barrel shifter operations. (7M)

In ARM architecture, the **Barrel Shifter** and **Arithmetic Logic Unit (ALU)** work closely together to perform data processing operations. This design allows for **flexible and efficient computation** within a single instruction cycle, especially in **arithmetic, logical**, and **move** operations.

Diagram: Data Path with Barrel Shifter and ALU

- **Rn** goes directly to the ALU.
- **Rm** is optionally passed through the **barrel shifter** before entering the ALU.
- The ALU performs the operation and sends the **result to destination register Rd**.



Barrel shifter and ALU.

Role of Barrel Shifter

- The **Barrel Shifter** can **preprocess** operand Rm by applying a shift or rotate operation **within the same instruction cycle**.
- Supports five types of shifts:
  - **LSL** – Logical Shift Left
  - **LSR** – Logical Shift Right
  - **ASR** – Arithmetic Shift Right

- **ROR** – Rotate Right
- **RRX** – Rotate Right with Extend (using carry)

Example:

```
MOV r7, r5, LSL #2   ; r7 = r5 << 2
```

If `r5 = 5`, result is `r7 = 20` (since 5 × 2^2 = 20).

Advantage: Saves instruction cycles by combining shift and operation in one step.

Role of ALU

- The **Arithmetic Logic Unit (ALU)** is responsible for performing:
  - **Arithmetic operations**: ADD, SUB, ADC, SBC
  - **Logical operations**: AND, ORR, EOR, BIC
  - **Comparison operations**: CMP, CMN
- Receives **Rn** directly and **Rm** via the barrel shifter (if needed).

Example:

```
ADD r0, r1, r2, LSL #1  ; r0 = r1 + (r2 << 1)
```

This adds `r1` to `r2` shifted left by one (i.e., `r2 * 2`), and stores the result in `r0`.

M5Q9C) Write an ALP to add the first 10 integer numbers using Cortex M3 processor. (5M)

Objective:

To write an **Assembly Language Program (ALP)** that computes the **sum of integers from 1 to 10**, i.e.,

`1 + 2 + 3 + ... + 10 = 55`

Assembly Program (Thumb-2, Cortex-M3 Syntax):

```
    AREA ADD_NUMBERS, CODE, READONLY
    ENTRY
```

```
        MOV     R0, #1          ; R0 = counter = 1
        MOV     R1, #0          ; R1 = sum = 0

LOOP    ADD     R1, R1, R0      ; sum = sum + counter
        ADD     R0, R0, #1      ; counter = counter + 1
        CMP     R0, #11         ; check if counter > 10
        BNE     LOOP            ; if not equal, repeat loop

        ; Result is in R1 = 55

        END
```

Explanation of Code:

Instruction	Purpose
`MOV R0, #1`	Initialize counter to 1
`MOV R1, #0`	Initialize sum to 0
`ADD R1, R1, R0`	Add current counter to sum
`ADD R0, R0, #1`	Increment counter
`CMP R0, #11`	Compare counter to 11 (loop until 10)
`BNE LOOP`	Branch if not equal (i.e., continue loop)

Result:

After the loop completes, **R1 = 55**, which is the sum of integers from 1 to 10.

M5Q10A) Explain the operation of following instruction with syntax and an example for each : (i) QDADD, (ii) LDMIA, (iii) BEQ, (iv) LSR, (v) UMULL (10M)

(i) QDADD – Saturated Double and Add

Syntax: `QDADD Rd, Rm, Rn`

Operation: Performs: `Rd = Rn + (Rm × 2)` with **saturation**. If doubling `Rm` or the final result overflows, it saturates to the max/min 32-bit signed integer.

Example:

```
QDADD R2, R0, R1
```

If `R0 = 0x40000000` and `R1 = 0x40000000`,

`R2 = 0x40000000 + 0x80000000 = 0xC0000000` (saturation not required).

But if doubling R0 overflows, result is saturated to `0x7FFFFFFF`.

Use Case: Useful in DSP applications requiring overflow protection.

(ii) LDMIA – Load Multiple Increment After

Syntax: `LDMIA Rn!, {Rlist}`

Operation: Loads multiple registers from consecutive memory locations starting from the address in `Rn`. After each load, `Rn` is incremented. The `!` writes back the updated address.

Example:

```
LDMIA R5!, {R0-R3}
```

Loads four words into R0–R3 from memory pointed to by R5. Then, R5 is incremented by 16 bytes.

Use Case: Efficient context restoration or multiple data loads.

(iii) BEQ – Branch if Equal

Syntax: `BEQ label`

Operation: Branches to the specified label **if the Zero flag (Z) is set**, typically after a comparison instruction.

Example:

```
CMP R1, R2
BEQ equal_handler
```

If R1 equals R2, control jumps to `equal_handler`.

Use Case: Used in conditional branching and decision-making.

(iv) LSR – Logical Shift Right

Syntax: `LSR Rd, Rm, #n`

Operation: Shifts the value in Rm **right by n bits**, inserting zeros into the most significant bits. No sign extension.

Example:

```
LSR R1, R2, #2
```

If R2 = `0x000000F0`, result: R1 = `0x0000003C`.

Use Case: Used for unsigned division by powers of 2 or bit masking.

(v) UMULL – Unsigned Multiply Long

Syntax: `UMULL RdLo, RdHi, Rn, Rm`

Operation: Performs an **unsigned 32 × 32-bit multiplication** and stores the 64-bit result in two registers: lower 32 bits in RdLo, upper 32 bits in RdHi.

Example:

```
UMULL R4, R5, R2, R3
```

If R2 = 0x00010000 and R3 = 0x00010000,

Product = 0x0000000100000000 →

R4 = 0x00000000 (Lo), R5 = 0x00000001 (Hi)

Use Case: Used in arithmetic operations requiring full 64-bit precision.

Summary Table:

Instruction	Operation Type	Description
QDADD	Saturating Arithmetic	`Rn + (2 × Rm)` with saturation
LDMIA	Load Multiple	Loads multiple registers from memory
BEQ	Conditional Branch	Branches if Z (zero) flag is set
LSR	Shift	Logical right shift, zero-filled
UMULL	Multiply	32-bit × 32-bit → 64-bit unsigned product

M5Q10B) Explain the SWAP, SWI and Program status register instructions of Cortex M3 with example for

each. (10M)

1. SWAP Instruction (SWP)

The SWAP instruction is an atomic operation that exchanges data between a register and a memory location. This ensures that no other bus access can interrupt the operation, which is critical in implementing synchronization mechanisms such as semaphores.

Syntax:

`SWP{B}{<cond>} Rd, Rm, [Rn]`

Operation:

- Reads the value from memory `[Rn]`
- Writes the value of `Rm` to memory `[Rn]`
- Stores the old value read from memory into `Rd`

Example:

```
PRE: mem32[0x9000] = 0x12345678
     r1 = 0x11112222
     r2 = 0x00009000
     r0 = 0x00000000

SWP r0, r1, [r2]

POST: mem32[0x9000] = 0x11112222
      r0 = 0x12345678
```

This instruction is commonly used for implementing mutual exclusion locks or semaphores in operating systems.

2. SWI Instruction (Software Interrupt)

The SWI instruction generates a software interrupt, which switches the processor to Supervisor (SVC) mode. It's typically used for invoking OS services from user mode.

Syntax:

`SWI{<cond>} SWI_number`

Operation:

- Forces processor into SVC mode
- Saves return address in `lr_svc` and CPSR in `spsr_svc`
- Jumps to vector at `0x00000008` or `0xFFFF0008`

Example:

```
SWI 0x123456
```

Before execution:

- `cpsr = USER`, `pc = 0x00008000`, `r0 = 0x12`

After execution:

- `pc = 0x00000008`, `cpsr = SVC`, `spsr = USER`, `lr = 0x00008004`, `r0 = 0x12`

Register `r0` is typically used for parameter passing to the SWI handler, and return values are also passed via registers.

3. Program Status Register (PSR) Instructions

The Program Status Registers include:

- CPSR (Current Program Status Register)
- SPSR (Saved Program Status Register)

These registers hold status and control information such as flags, processor mode, and interrupt masks.

Instructions:

- `MRS Rd, CPSR|SPSR` → Copy PSR to a register
- `MSR CPSR|SPSR_<fields>, Rm|#immediate` → Write to PSR

Example: Enabling IRQ by clearing the I mask:

```
MRS r1, cpsr       ; read CPSR into r1
BIC r1, r1, #0x80    ; clear I bit
MSR cpsr_c, r1       ; write back to CPSR
```

This allows IRQ interrupts by clearing bit 7 (I bit).

Fields in PSR:

- **Flags (N, Z, C, V)** – condition flags

- **Control (I, F, T, Mode)** – interrupt masks and processor mode
- **Extension/Status** – reserved.

Summary Table:

Instruction	Purpose	Example	Key Result
SWP	Atomic swap between memory and register	`SWP r0, r1, [r2]`	Synchronization/locking
SWI	Invoke system services in SVC mode	`SWI 0x123456`	OS routine call
MRS/MSR	Read/write CPSR/SPSR	`MRS r1, cpsr`, `MSR cpsr_c, r1`	Control flags/modes

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