ESD IA3

M4Q1A) Explain the four major design rules that define the RISC philosophy. (7M)

The **RISC (Reduced Instruction Set Computer)** design philosophy emphasizes simplicity in hardware and instruction execution. It is based on the principle that performance can be optimized by simplifying processor operations and relying more on efficient software and compilers. The four major rules that define this philosophy are:

1. Simple Instructions (Reduced Instruction Classes)

RISC processors implement a **small, limited set of instruction types**, each performing a very simple operation that can usually execute in a **single clock cycle**.

- Complex operations (like multiplication or division) are constructed by combining these simple instructions in software.
- Instructions are of **fixed length**, simplifying **instruction decoding and pipelining**.

Example: Instead of a single complex multiply instruction, RISC would use shift and add instructions to achieve the same result.

2. Pipelining

The instruction execution is **divided into stages** such as fetch, decode, and execute.

- Each stage is processed in a **pipeline**, allowing multiple instructions to be executed in **parallel**, improving throughput.
- The pipeline ideally advances by one stage **per clock cycle**, maximizing instruction throughput.

Advantage: Allows simple hardware implementation and high clock speeds without complex microcoding.

3. Large Register Set

RISC architectures provide a **large number of general-purpose registers**.

- All registers can hold either data or addresses.
- **Registers are faster** to access than memory and are used extensively for computation and temporary storage.

Benefit: Minimizes memory access, thus speeding up computation.

4. Load-Store Architecture

Only **load** and **store** instructions access memory.

- **All operations** are performed on data in **registers**.
- This separation of computation and memory access leads to a **cleaner and faster execution pipeline**.

Contrast with CISC: In CISC processors, instructions can operate directly on memory, which increases complexity and execution time.

M4Q1B) Describe the concept of pipleline instruction execution in ARM processor. Explain the five-stage pipeline used in ARM9 architecture with a neat labeled diagram. (8M)

Concept of Pipelining in ARM Processors

Pipelining is a technique used in RISC architectures like ARM to **increase instruction throughput** by overlapping the execution of multiple instructions.

Instead of waiting for one instruction to finish before starting the next, pipelining breaks instruction execution into multiple **stages**, with each stage processed in **parallel**.

- This improves **instruction throughput** and allows **one instruction per clock cycle** in ideal conditions.
- Each instruction passes through **several stages**, and multiple instructions are processed simultaneously at different stages.

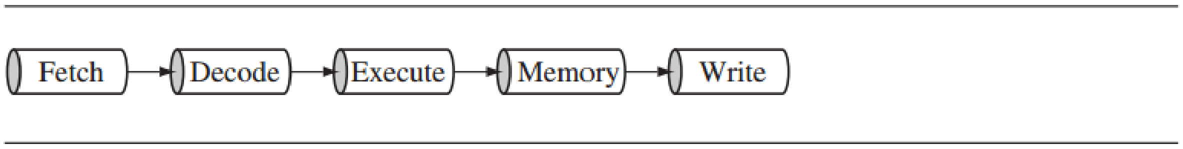
Five-Stage Pipeline in ARM9 Architecture

The **ARM9** processor uses a **five-stage pipeline**, which is more efficient than the three-stage pipeline used in ARM7.

Pipeline Stages:

Stage	Name	Function
1	**Fetch (F)**	Fetches the instruction from memory using the Program Counter (PC).
2	**Decode (D)**	Decodes the instruction and reads operand registers.
3	**Execute (E)**	Performs ALU operation or address calculation.
4	**Memory (M)**	Accesses memory for load/store instructions.
5	**Write-back (WB)**	Writes the result back to the destination register.

Pipeline Diagram:



ARM9 five-stage pipeline.

```
Clock Cycle →   1    2    3    4    5    6
Instruction 1:  F  →  D  →  E  →  M  →  WB
Instruction 2:       F  →  D  →  E  →  M  →  WB
Instruction 3:            F  →  D  →  E  →  M  →  WB
Instruction 4:                 F  →  D  →  E  →  M  →  WB
```

Each row shows a new instruction entering the pipeline. By the 5th cycle, **five instructions are being processed simultaneously** in different stages.

Advantages of ARM9 Pipeline:

- **Improved performance** through instruction-level parallelism.
- **Reduced execution time** per instruction.
- **Efficient resource utilization**, as each stage is active every cycle.

M4Q2A) With a neat block diagram explain the architecture of an ARM-based microcontroller. (7M) [Refer ESD EX.pdf]

Overview

An ARM-based microcontroller combines a high-performance **ARM processor core** (such as Cortex-M3) with **memory, peripherals, and interfaces** on a single chip. The architecture is optimized for embedded control, real-time responsiveness, and low power consumption.

Block Diagram

You may draw a simplified version of the following:

```
+-----------------------------+
|      ARM Core (CPU)      |
| - ALU                  |
| - Register File         |
| - Program Counter (PC)    |
+-------------+---------------+
              |
+-------------v---------------+
|  NVIC (Interrupt Ctrl)     |
+-------------+---------------+
              |
+-------------v---------------+
|  Memory System            |
| - Flash Memory            |
| - SRAM                  |
| - EEPROM (optional)       |
+-------------+---------------+
              |
+-------------v---------------+
|  Bus Interface Unit       |
|  (AHB/APB Bus Matrix)      |
+-------------+---------------+
              |
+-------------v---------------+
|  Peripherals              |
| - Timers/Counters         |
| - GPIO                  |
│ - UART, SPI, I2C, ADC, etc.│
+-----------------------------+
```

Explanation of Components

1. ARM Core (CPU):

Executes instructions and performs data processing. Cortex-M cores support Thumb-2 instructions, pipelining, and hardware divide.

2. NVIC (Nested Vectored Interrupt Controller):

Manages hardware interrupts with programmable priorities. Supports nested and low-latency interrupt handling.

3. Memory System:

- **Flash**: Stores program code (non-volatile).
- **SRAM**: For variables and stack (volatile).
- **EEPROM** (optional): Used for non-volatile variable storage.

4. Bus Interface Unit (AHB/APB):

Enables communication between CPU, memory, and peripherals. AHB used for high-speed transfers; APB for low-speed peripherals.

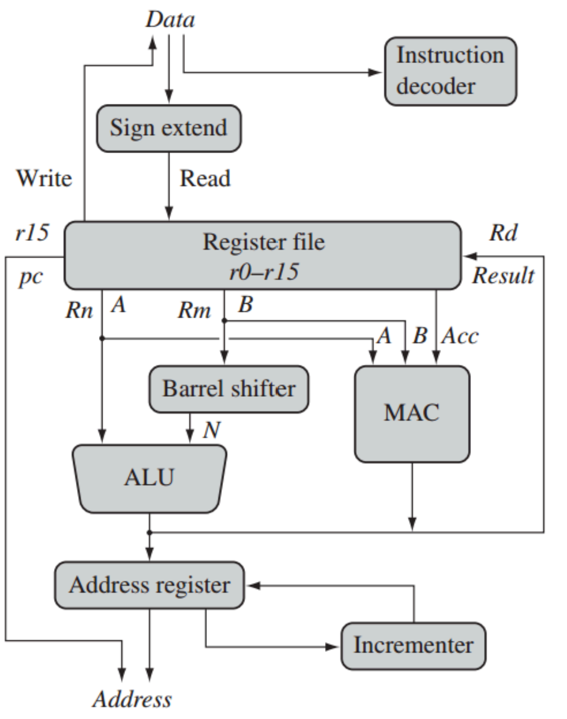
5. Peripherals:

Built-in modules like Timers, GPIO, ADC, UART, SPI, I2C, PWM, used for hardware control and communication.

M4Q2B) Describe ARM core dataflow model with a neat and labeled diagram. Explain how data flows through various components of ARM core during instruction execution. (8M)

The **ARM core dataflow model** represents how data moves inside the processor during the execution of instructions. It visually outlines how registers, buses, ALU, shifters, and control logic interact to carry out data operations efficiently.

Labeled Diagram of ARM Core Dataflow Model



ARM core dataflow model.

Explanation of Dataflow Components

1. Instruction Decoder:

Receives instruction from memory. Decodes the operation type, operands, and control signals.

2. Register File (r0–r15):

Stores operands and results. Operands are fetched via **internal buses A and B** for processing.

3. Barrel Shifter:

Used for efficient data manipulation. Operates on one operand before it enters the ALU, allowing operations like logical shift, arithmetic shift, and rotate.

4. ALU (Arithmetic Logic Unit) / MAC (Multiply-Accumulate Unit):

Executes arithmetic or logical operations on input operands. Outputs result to be stored in a register or memory.

5. Result Register:

Holds the result temporarily before it's written back to the register file.

6. Address Register:

Holds memory addresses for load/store operations. Address is calculated by ALU.

7. Memory Interface:

Reads from or writes to memory during load-store operations, following address resolution.

Instruction Execution Flow Example

Let's consider a simple **ADD R1, R2, R3** instruction:

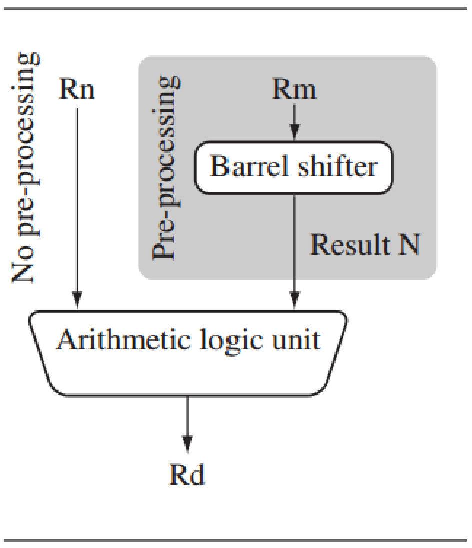
1. **Fetch:** Instruction is fetched from memory.
2. **Decode:** It's decoded to identify operation `ADD` with `R2` and `R3` as operands.
3. **Operand Fetch:** Values from R2 and R3 are fetched from the register file.
4. **Barrel Shifter:** If needed, shifts R3 (not in this simple ADD).
5. **ALU:** Performs `R2 + R3`.
6. **Result Register:** Stores result temporarily.
7. **Write Back:** Result is written into R1.

M5Q3A) Describe the roles of Barrel Shifter and ALU in an ARM processor with the help of neat diagram. (6M)

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 $5 \times 2^2 = 20$).

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`.

M5Q3B) Explain the following instructions with an example each: i) SMULL, ii) RSC, iii) TEQ. (6M)

i) SMULL – Signed Multiply Long

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

Operation: Performs a **signed 32-bit × 32-bit multiplication** of operands `Rn` and `Rm`, producing a **64-bit result**, with the lower 32 bits stored in `RdLo` and upper 32 bits in `RdHi`.

Example:

```
SMULL R4, R5, R2, R3
```

If `R2 = -10` and `R3 = 1000`, then the result is `-10000`.

- Lower 32 bits (`0xFFFFD8F0`) go to `R4`
- Upper 32 bits (sign extended, if needed) go to `R5`

Use Case: When high precision multiplication is required and the result may exceed 32 bits.

ii) RSC – Reverse Subtract with Carry

Syntax: `RSC Rd, Rn, Rm`

Operation: Performs `Rd = Rm - Rn - NOT(Carry)`, i.e., **reverses** the order of subtraction and includes the carry flag. It's the reverse of the `SBC` instruction.

Example:

```
RSC R1, R2, R3
```

If `R2 = 5`, `R3 = 10`, and Carry = 1, then:

`R1 = R3 - R2 - (1 - Carry)` = `10 - 5 - 0 = 5`

Use Case: Used in multi-word subtraction with borrow propagation.

iii) TEQ – Test Equivalence (Bitwise XOR, Updates Flags Only)

Syntax: `TEQ Rn, Rm`

Operation: Performs a **bitwise XOR** between `Rn` and `Rm`, **updates condition flags**, but **does not store the result**.

Example:

```
TEQ R2, R3
```

- If `R2 = 0xAA` and `R3 = 0xAA`, XOR = 0 → Zero flag is set.
- If they differ, Zero flag is cleared.

Use Case: Used in conditional branching when **equality of bit patterns** needs to be tested, without affecting register contents.

Summary Table

Instruction	Operation	Key Use
`SMULL`	Signed 32×32 → 64-bit multiplication	High-precision math
`RSC`	Reverse subtract with carry	Multi-word subtraction
`TEQ`	Bitwise XOR, sets flags only	Conditional logic

M5Q4A) Discuss Single-Register-Transfer (SRT) and Multiple-Register-Transfer (MRT) in ARM architecture with suitable examples. (6M)

Single-Register-Transfer (SRT)

Single-register-transfer instructions are ARM instructions that operate on **one source and one destination register**. They are used for **data movement between register and memory**, or between two registers.

Key Instructions: `LDR`, `STR`, `MOV`, `MVN`, `LDRB`, `STRB`

Example 1 – Load Register:

```
LDR R1, [R2]
```

Loads the value from the memory address stored in `R2` into `R1`.

Example 2 – Move Register to Register:

```
MOV R0, R1
```

Copies the contents of `R1` into `R0`.

Features:

- Transfers a **single register** at a time.
- May involve immediate values, shifted register values, or memory addresses.

Multiple-Register-Transfer (MRT)

Multiple-register-transfer instructions allow **storing or loading multiple registers** from/to memory in a single instruction. These are highly efficient for **context saving**, **function calls**, or **stack operations**.

Key Instructions: `LDM`, `STM`, `PUSH`, `POP`

Example 1 – Load Multiple:

```
LDMIA R4!, {R0-R3}
```

Loads registers `R0–R3` from memory starting at address in `R4`, then increments `R4`.

Example 2 – Store Multiple:

```
STMFD SP!, {R4-R7, LR}
```

Stores registers `R4–R7` and `LR` on the stack and decrements `SP`.

Features:

- Operates on **register lists** enclosed in `{}`.

- Supports stack and frame management in procedure calls.
- Reduces instruction count and improves efficiency.

Summary Table

Feature	SRT (Single-Register-Transfer)	MRT (Multiple-Register-Transfer)
Registers involved	One at a time	Multiple at once
Use Case	Simple load/store or move	Stack ops, function prologue/epilogue
Examples	`MOV`, `LDR`, `STR`	`LDM`, `STM`, `PUSH`, `POP`

SRT is ideal for basic data transfers and immediate values, whereas MRT is optimized for block memory operations and register save/restore.

M5Q4B) Explain the operation of the following instructions with example each: i) ASR, ii) SBC, iii) BIC (6M)

i) ASR – Arithmetic Shift Right

Syntax: `ASR Rd, Rm, #n`

Operation: Shifts the bits in register `Rm` **right by n bits**, while **preserving the sign bit (MSB)**. This is equivalent to division by a power of two for signed integers.

Example:

If `R1 = 0xFFFFFFF8` (i.e., -8 in 2's complement),

```
ASR R2, R1, #2
```

The result in `R2` is `0xFFFFFFFE` (i.e., -2), because `-8 >> 2 = -2` with sign extension.

Use Case: Efficient signed division by powers of two.

ii) SBC – Subtract with Carry

Syntax: `SBC Rd, Rn, Rm`

Operation: Performs subtraction including the **inverse of the carry flag**:

`Rd = Rn - Rm - (1 - Carry)`

Example:

If `R2 = 15`, `R3 = 5`, and carry = 1,

```
SBC R1, R2, R3
```

Result: `R1 = 15 - 5 - 0 = 10`

If carry = 0, then:

`R1 = 15 - 5 - 1 = 9`

Use Case: Used in multi-word (64-bit) subtraction with borrow handling.

iii) BIC – Bit Clear (AND with NOT)

Syntax: `BIC Rd, Rn, Rm`

Operation: Performs bitwise operation: `Rd = Rn AND NOT(Rm)`

It **clears specific bits** in a register as per the mask defined in `Rm`.

Example:

If `R4 = 0xFFFF000F` and `R5 = 0x0000000F`,

```
BIC R6, R4, R5
```

Result: `R6 = 0xFFFF0000`, since the lower 4 bits are cleared.

Use Case: Used for clearing or masking specific bits.

Summary Table

Instruction	Operation	Use Case
`ASR`	Arithmetic right shift with sign	Signed division by powers of two
`SBC`	Subtract with carry-in	Multi-word subtraction
`BIC`	Bitwise AND with NOT (bit clear)	Bit masking/clearing

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