IPA Project - Team 15
RISC-V Processor

Sarvesh Takbhate
2023102039
sarvesh.takbhate@students.iiit.ac.in

Vedant Pahariya
2023112012
vedant.pahariya@research.iiit.ac.in

Srihari Padmanabhan
2023102021
srihari.padmanabhan@students.iiit.ac.in

***Abstract*—RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture designed for efficiency, simplicity, and scalability. It supports various instruction formats and execution models, making it suitable for embedded systems, general-purpose computing, and high-performance applications. This document explores the RISC-V instruction sequencing and pipelining, detailing its stages and the role of the individual blocks ensuring smooth instruction flow within the pipeline.**

I. Introduction

RISC-V is a modular and extensible instruction set architecture (ISA) that has gained popularity due to its open-source nature and design flexibility.

We can implement the processor in two different ways which have their own pros and cons:

- Sequential Processing
- Pipelined Processing

Sequential Processing

- **Execution Method:** Instructions are executed one after the other in a linear fashion.
- **Speed:** Slower execution as only one instruction is processed at a time.
- **Throughput:** Low throughput since each instruction must complete before the next begins.
- **Resource Utilization:** A single stage implementation means hardware resources are often idle, leading to lower efficiency.
- **Latency:** A sequential implementation does not require balancing of stages and operates quickly for a small number instructions.
- **Complexity:** Simple control logic and easier to design and implement.
- **Use Case:** Suitable for simpler and non-time-critical applications.

Pipelined Processing

- **Execution Method:** Allows multiple instructions to be processed simultaneously in different stages of the pipeline.
- **Speed:** Faster execution by leveraging parallelism and overlapping instruction execution.
- **Throughput:** High throughput as new instructions can enter the pipeline before previous ones finish.
- **Resource Utilization:** More efficient utilization of hardware resources by keeping all pipeline stages active.
- **Latency:** The time taken per instruction is not improved by pipelined processing as compared to sequential processing - the advantage lies in the throughput.
- **Complexity:** More complex design due to the need for hazard handling and stage synchronization.
- **Use Case:** Ideal for high-performance computing and modern processors requiring fast processing rates.

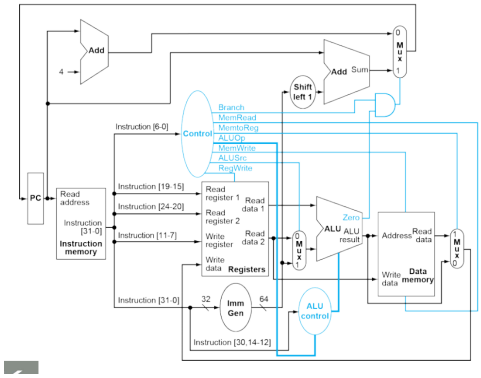

Fig. 1: Sequential Datapath with controls

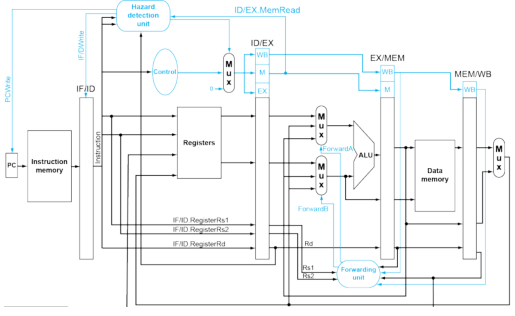

Fig. 2: Pipelined Datapath with Hazard Detection

The project follows a structured approach, beginning with the design of a sequential execution processor, followed by the implementation of a pipelined architecture. We use Icarus **Verilog** to implement our Architecture. in-depth discussion of each stage of development of the project.

II. Specifications

The processor design must meet the following specifications:

- A fundamental implementation of the processor architecture using a sequential design.
- An advanced implementation featuring a 5-stage pipelined processor architecture, incorporating mechanisms to handle pipeline hazards effectively.

Both implementations must support the execution of the following instructions from the RISC-V ISA: `add`, `sub`, `and`, `or`, `ld`, `sd`, and `beq`.

Additionally, we have implemented the `addi` instruction to facilitate loading values into registers, eliminating the need for manual register initialization.

For input we have created a python script in which we give **`RISC-V Language`** as input and then it converts it to hexcode which is stored in the instruction memory module.

Due to space constraints, we created separate documents for test results. The complete GTKwave simulations with testcases and explanations provided are linked below.

- **Sequential Doc:** Click here
- **Pipeline Doc:** Click here

Part 1: Sequential Implementation

III. Design of Sequential Processor

A. Architecture Overview

The sequential RISC-V processor follows a single-cycle execution model, where each instruction is fetched, decoded, executed, and written back within one clock cycle. This approach simplifies control logic but results in longer cycle times due to the complexity of executing all operations in one step.

The processor is composed of several key components that work together to fetch, decode, execute, and store instructions. These include:

- **Arithmetic Logic Unit (ALU)**: Performs arithmetic and logical operations based on control signals.
- **Control Unit**: Generates control signals required to execute instructions correctly.
- **Register File**: Holds a set of registers for temporary data storage.
- **Immediate Generator**: Extracts and sign-extends immediate values from instruction encoding.
- **Instruction Memory**: Stores the program instructions.
- **Data Memory**: Holds data values that are loaded and stored during execution.
- **Program Counter (PC)**: Maintains the address of the next instruction to be executed.

The overall architecture can be seen in the block diagram below:

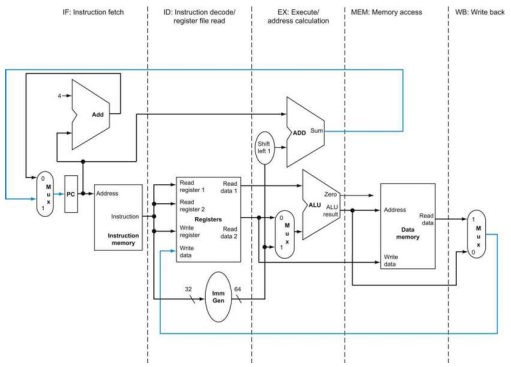

Fig. 3: Block diagram of the sequential processor

B. Supported Instruction Set

This processor supports a basic subset of the RISC-V instruction set, covering arithmetic, logical, memory, and control operations. The following instructions are implemented:

- **R-type Instructions**: `add, sub, and, or, xor, sll, srl, slt`
- **I-type Instructions**: `addi, andi, ori, xori, slli, srli, slti, lw`
- **S-type Instructions**: `sw`
- **B-type Instructions**: `beq, bne, blt, bge`

The instruction encoding follows the standard RISC-V format, with fields for opcode, register addresses, function codes, and immediates.

C. Sequential Execution Flow

The processor follows a straightforward execution model, where each instruction undergoes the following stages in a single clock cycle.

1) **Instruction Fetch (IF)**: The instruction memory provides the instruction at the address specified by the Program Counter (PC).
2) **Instruction Decode (ID)**: The opcode and function bits are extracted to determine the instruction type. The register addresses are also extracted, and values are read from the register file.
3) **Execution (EX)**: The ALU performs the necessary arithmetic or logical operation. For branch instructions, the target address is computed.
4) **Memory Access (MEM)**: Load and store instructions interact with data memory. Other instructions skip this stage.

5) **Write Back (WB)**: The result is written back to the destination register, if applicable.

Since the sequential processor executes one instruction per clock cycle, **the performance is constrained by the slowest instruction**, as all operations must complete before the next cycle begins. It is also important to note that the register/memory gets written into on the **subsequent clock edge**.

IV. Module-wise Implementation

A. *IF: Instruction Fetch*

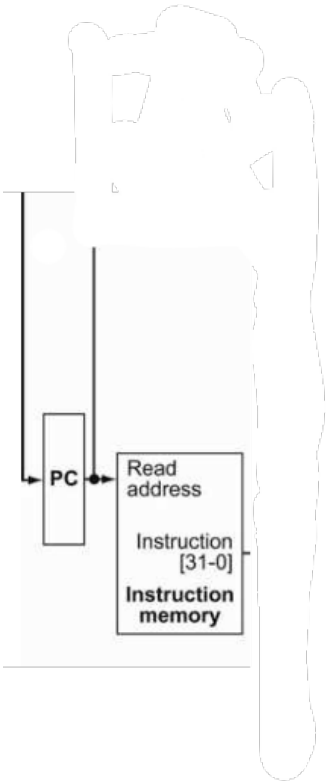

Fig. 4: Block diagram of the IF Stage

The given block diagram represents a simple instruction fetching mechanism in a processor. The main components involved are:

- Program Counter (PC)
- Instruction Memory

The PC is responsible for keeping track of the instruction address, and the instruction memory fetches the corresponding instruction.

- Program Counter (PC): The module `p_c.v` implements the PC logic.
  - On a positive clock edge, if `next_addr` is valid (non-negative), it gets loaded into the PC register.
  - `curr_addr` is the address available at the PC output.
- Instruction Memory : The `insmem` module represents an instruction memory that fetches instructions based on an 8-bit address.
  - Instructions are stored in a little-endian format in an external file `Instructions.mem`.
  - The module extracts the following instruction fields:
    * Opcode (7-bit control field)
    * Destination register (rd, 5 bits)
    * Source registers (rs1, rs2, each 5 bits)
    * Full 32-bit instruction
  - Depending on the opcode, the module displays instruction details such as R-format, I-format, Load, Store, and Branch operations.

B. *ID: Instruction Decode Stage*

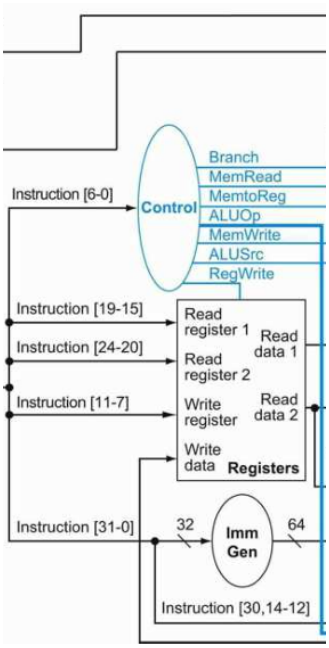

Fig. 5: Block diagram of the ID Stage

- Control Module
  The `control.v` module is responsible for generating control signals based on the instruction opcode. The input is a 7-bit opcode, and the module outputs the following control signals:
  - `branch` - Determines if the instruction is a branch instruction.
  - `RegWrite` - Enables writing to registers.
  - `MemtoReg` - Controls whether data comes from memory or ALU.
  - `MemRead` - Enables reading from memory.

- `MemWrite` - Enables writing to memory.
- `alu_src` - Determines if ALU input comes from register or immediate.
- `alu_op` - Defines the ALU operation type.

The behavior of this module depends on the opcode values:

- R-format instructions (opcode = 0110011) perform register-to-register operations.
- I-format instructions (opcode = 0010011) involve immediate values.
- Load (opcode = 0000011) reads data from memory.
- Store (opcode = 0100011) writes data to memory.
- Branch (opcode = 1100011) performs conditional jumps.

- Immediate Generator Module
  The `imm_gen.v` module extracts the 12 bit immediate from the instruction and sign-extends it to 64 bits. The extraction depends on the instruction type:
  - I-format: Immediate is extracted from bits [31:20].
  - Load instructions: Similar to I-format.
  - Store instructions: Immediate is constructed from bits [31:25] and [11:7].
  - Branch instructions: Immediate is built from scattered bits [31, 7, 30:25, 11:8].

- Register Module
  The `register.v` module simulates a set of registers in a processor. It includes:
  - Read operations: Two registers are read based on the instruction.
  - Write operations: Data is written to a register on the positive clock edge if `RegWrite` is enabled.
  - Using the **addi** function we store the value of data in the registers.

The register file is initialized using the `Registerlog.mem` file.

C. *EX: Execute Stage*

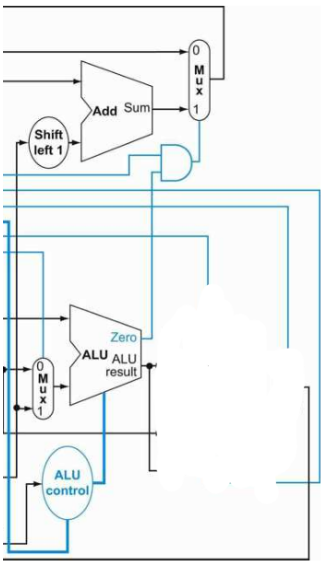

Fig. 6: Block diagram of the EX Stage

The execution (EX) stage is responsible for performing arithmetic and logical operations, computing branch target addresses, and selecting the correct operand for execution. It consists of several key components:

- **adder.v:** This unit calculates the next instruction address (`PC+4`) and the branch target address (`PC + 2 * target`).We **and** the `zero` and the `branch` output to obtain `PCsrc` .If it is 1 the branch is taken, otherwise the sequence of operations continues as normal.
- **alu.v (Arithmetic Logic Unit):** The ALU performs arithmetic and logical operations such as addition, subtraction, AND, OR, shift operations, and comparisons. It takes two input operands and executes the required operation based on the control signals.
- **alu_control.v:** This block generates the ALU operation signals based on the instruction type. It interprets the opcode and function bits to determine whether the ALU should perform addition, subtraction, bitwise operations, or shifts.

ALUOp	funct7	funct3	Instruction	ALU Control Output
00	-	-	Load/Store	0010 (ADD)
01	-	-	Branch	0110 (SUB)
10	0000000	000	ADD	0010 (ADD)
10	0100000	000	SUB	0110 (SUB)
10	0000000	100	XOR	0011 (XOR)
10	0000000	110	OR	0001 (OR)
10	0000000	111	AND	0000 (AND)

TABLE I: ALU Control Signal Mapping Based on funct7, funct3, and ALUOp

Here, `ALUOp` acts as the select signal, determining whether the instruction is a load/store, branch, or R-type operation. The `funct7` and `funct3` fields further specify the ALU operation. The default is set to **and** operation .

- **Immediate Multiplexer (imm_mux.v):** This multiplexer selects between the second register operand (`rs2`) and an immediate value (`imm`) based on the instruction type.The `ALU_src` is the select line for the mux which comes from control block. If the instruction uses an immediate value, the multiplexer forwards it to the ALU instead of `rs2`.

D. MEM: Memory Stage

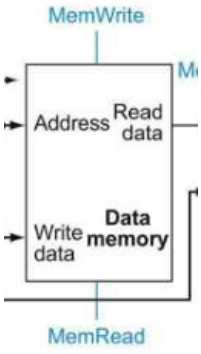

Fig. 7: Block diagram of the MEM Stage

The data memory module manages memory read and write operations for load and store instructions.

- **Memory Unit:** A 1024-entry memory that stores 64-bit signed values. It is initialized using a memory file (`Memorylog.mem`) and supports both read and write operations.
- **Memory Read Operation:** If `MemRead` is enabled, the data at the specified `address` is read from memory and forwarded to the next stage.
- **Memory Write Operation:** If `MemWrite` is enabled, the value from `data_in` is stored at the specified `address` in memory. The updated memory contents are saved back to `Memorylog.mem` for persistence.
- **Address Output Register:** The module includes an `address_out` register to help visualize memory updates in GTKWave.
- **Clock Dependency:** Memory write operations occur on the rising edge of the clock (`posedge clk`), ensuring synchronous updates, while read operations are combinational.

E. WB: Write-Back Stage

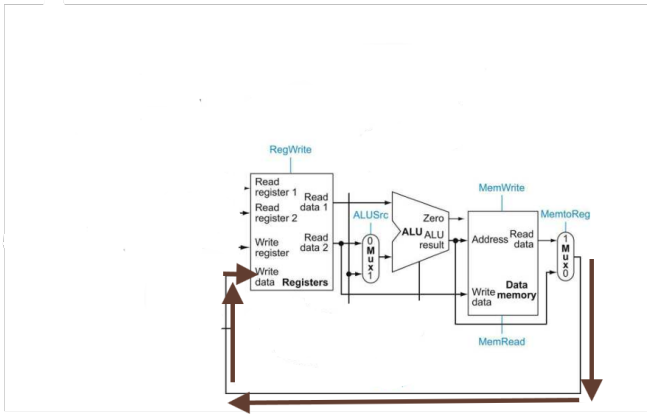

Fig. 8: Block diagram of the WB Stage

The Write-Back (WB) stage is the final stage of the pipeline, where the computed results are written back to the register file. It consists of the following components:

- **Write-Back MUX:** The multiplexer (`wb_mux`) selects whether the value written to the register file comes from the ALU result or memory, based on the `MemtoReg` control signal.
- **ALU Result Input:** If `MemtoReg` is 0, the result from the ALU (`alu_out`) is forwarded to the write-back register.
- **Memory Output Input:** If `MemtoReg` is 1, the data read from memory (`mem_out`) is written back to the register file.
- **Combinational Logic:** The selection occurs within an always block, meaning the write-back value is updated as soon as any input changes.
- **Final Register Update:** The selected value is assigned to the output `writeback`, ensuring the correct value is stored in the destination register.

Part 2: Pipeline Implementation

Pipelining is a technique used in processor design to improve instruction throughput by executing multiple instructions simultaneously in different stages. Instead of executing one instruction at a time (as in a sequential processor), pipelining divides instruction execution into multiple stages, allowing new instructions to enter the pipeline before the previous ones have finished. Following are the advantages of using Pipelining over Sequential.

- **Increased Throughput:** By overlapping execution, multiple instructions are completed per unit time.
- **Efficient Resource Utilization:** Each stage of the processor is kept busy rather than remaining idle while waiting for previous instructions to complete.
- **Faster Execution:** The overall latency for a single instruction remains the same, but the number of instructions executed per cycle increases.
- **Minimizing Execution Bottlenecks:** Helps in reducing the delays caused by instruction dependencies.

V. Pipeline Stages and Pipeline Registers

Pipelining divides the instruction execution into multiple stages, each performing a part of the instruction execution. Pipeline registers are added between stages to store intermediate results and ensure smooth execution flow. As we are running multiple operations in our processor, there is a chance for us to get hazards and problems. To handle that we add two extra blocks called as `Hazard_Unit` and `Forwarding_Unit` .

A. Stages in a Pipelined Processor

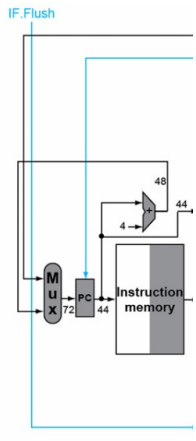

Fig. 9: Block diagram of the IF Pipeline Stage

1) Instruction Fetch (IF) Stage:

- Fetches the instruction from memory using the Program Counter(PC).
- Increments PC to point to the next instruction.
- Stores the instruction in the IF/ID pipeline register for the next stage.
- In this implementation PC also has a select line called `PC_write` . We need it for stalling whenever we detect a hazard .
- We just store the instruction in the register but we do the slicing of the instructions i.e registers ,immediate, control signals after the IF/ID pipeline register i.e in ID stage.
- The mux takes input as `PC = PC + 4` or `PC = PC + 2*target` depending on the select line `PC_src`

Pipeline Register: `IF/ID Register :-`

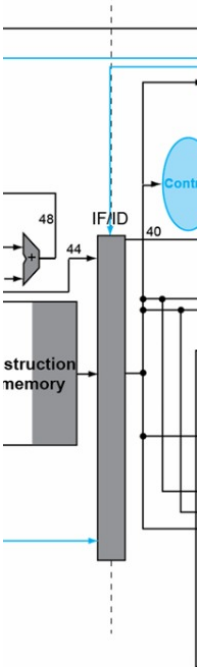

Fig. 10: Block diagram of the IF/ID Register

The **IF/ID pipeline register** serves as the intermediate storage between the **Instruction Fetch (IF) stage** and the **Instruction Decode (ID) stage** in a pipelined processor. It holds the fetched instruction and the updated program counter (PC) value to be used in the next stage. Functionality of this block is as follows :-

- **Stores Fetched Instruction and PC:** When a new instruction is fetched from memory, it is stored in `instruction_out`.
  The program counter (PC) is updated and stored in `pc_out`.
- **Extracts Key Fields from the Instruction:**
  - **Opcode (`ctrl`):** Extracts the 7-bit opcode from bits $[6 : 0]$ of the instruction, which helps determine the instruction type (R, I, S, B, etc.).
  - **Destination Register (`rd`):** Extracts bits $[11 : 7]$ that specify the destination register.
  - **Source Registers (`rs1` and `rs2`):** Extracts bits $[19 : 15]$ and $[24 : 20]$ respectively, which identify

the registers used in the instruction.

- **Controlled Write Operation:** The pipeline register updates only when `IF_ID_Write` is high, preventing unintended overwriting when stalling. The flush functionality of `IF_Flush` is achieved through `PCsrc`.

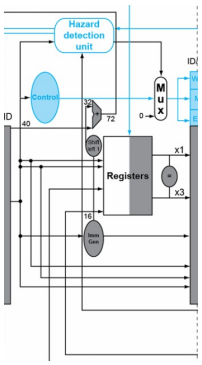

Fig. 11: Block diagram of the ID Pipeline Stage

2) Instruction Decode (ID) Stage: The **Instruction Decode (ID) Stage** processes the instruction fetched in the previous stage. It extracts critical components required for execution and determines necessary control signals.

1) **Register Read:** The source registers (`rs1` and `rs2`) are read from the register file based on the instruction fields.

2) **Immediate Generation:** The `ImmGen` unit extracts the immediate value from the instruction for operations requiring constants.

3) **Control Signal Generation:** The control unit decodes the opcode and generates necessary control signals for execution, memory access, and write-back stages.

4) **Hazard Detection:** The **Hazard Detection Unit** prevents data hazards caused by load-use dependencies. A load-use hazard occurs when an instruction in the execution stage (EX) needs data that has not yet been written back into the relevant register.
   - **Detecting Load-Use Hazard:** The unit checks if the destination register (`rd`) of an instruction currently in the EX stage is the same as one of the source registers (`rs1` or `rs2`) in the ID stage.
   - **Stalling the Pipeline:** If a hazard is detected, the pipeline is stalled by preventing updates to the PC and the IF/ID pipeline register.
   - The unit compares `IF_ID_rs1` and `IF_ID_rs2` with `ID_EX_rd`.
   - If there is a match and the instruction in the EX stage is a memory read (`ID_EX_MemRead = 1`), a stall is inserted.
   - The control signals `PC_Write` and `IF_ID_Write` are set to zero to stop the fetch and decode stages from progressing.

The logic used for the above unit is as follows

$$\text{If } (IF\_ID\_rs1 == ID\_EX\_rd) \text{ OR} \quad (1)$$
$$(IF\_ID\_rs2 == ID\_EX\_rd) \quad (2)$$
$$\text{AND } (ID\_EX\_MemRead) \quad (3)$$
$$\text{AND } (ID\_EX\_rd \neq 0) \quad (4)$$
$$\text{Then: } stall = 1, \quad (5)$$
$$PC\_Write = 0, \quad IF\_ID\_Write = 0 \quad (6)$$

This stage ensures that the necessary data and control signals are available for execution in the next stage.

Pipeline Register: `ID/EX Register :-`

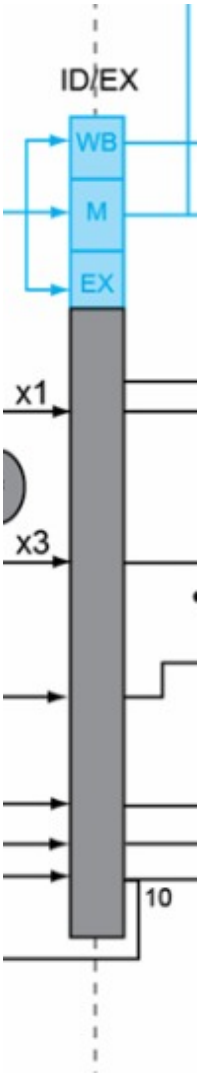

Fig. 12: Block diagram of the ID/EX Pipeline Register

The **ID/EX pipeline register** acts as an interface between the Instruction Decode (ID) and Execution (EX) stages, preserving instruction details and control signals.

- **Register Data:** Stores the values of source registers (`rs1_data`, `rs2_data`) and the destination register (`rd_data`).

- **Immediate Value:** Holds the computed immediate from the `ImmGen` unit.
- **Program Counter (PC):** Keeps the updated PC value for branch computations.
- **Control Signals:** Preserves control signals (`MemtoReg`, `RegWrite`, `Branch`, `MemRead`, `MemWrite`, `ALUSrc`, `ALUOp`) for later stages.
- **Instruction Fields:** Stores register identifiers (`rs1`, `rs2`, `rd`) for forwarding unit usage.

This register plays a crucial role in maintaining smooth instruction execution in the pipeline.

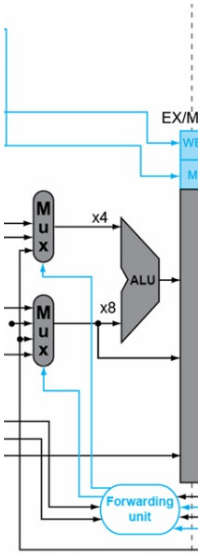

Fig. 13: Block diagram of the EX Pipeline Stage

3) Execution (EX) Stage: The Execution (EX) stage is responsible for performing arithmetic and logical operations using the ALU. Additionally, this stage handles operand selection through multiplexers and resolves data hazards using the forwarding unit.

- **ALU :**The ALU receives two inputs, which are selected through multiplexers:
  - The first operand is chosen between the register value (`rs1`) and a forwarded value.
  - The second operand is chosen between the register value (`rs2`), an immediate value, or a forwarded value.
- **Forwarding Unit :** The forwarding unit determines whether to use the values from previous pipeline stages (`EX/MEM`, `MEM/WB`) to avoid stalls.This is used to handle the Data hazard conditions.

$$
\begin{aligned}
&\text{If } (EX\_MEM\_regWrite \text{ AND} && (7)\\
&\quad (EX\_MEM\_rd \neq 0) \text{ AND} && (8)\\
&\quad (EX\_MEM\_rd = ID\_EX\_rs1)) && (9)\\
&\quad \text{Then: forward A} = 10 && (10)\\
&\text{Else if } (MEM\_WB\_regWrite \text{ AND} && (11)\\
&\quad (MEM\_WB\_rd \neq 0) \text{ AND} && (12)\\
&\quad (MEM\_WB\_rd = ID\_EX\_rs1)) && (13)\\
&\quad \text{Then: forward A} = 01 && (14)\\
&\text{Else: forward A} = 00 && (15)\\
& && (16)
\end{aligned}
$$

$$
\begin{aligned}
&\text{If } (EX\_MEM\_regWrite \text{ AND} && (17)\\
&\quad (EX\_MEM\_rd \neq 0) \text{ AND} && (18)\\
&\quad (EX\_MEM\_rd = ID\_EX\_rs2)) && (19)\\
&\quad \text{Then: forward B} = 10 && (20)\\
&\text{Else if } (MEM\_WB\_regWrite \text{ AND} && (21)\\
&\quad (MEM\_WB\_rd \neq 0) \text{ AND} && (22)\\
&\quad (MEM\_WB\_rd = ID\_EX\_rs2)) && (23)\\
&\quad \text{Then: forward B} = 01 && (24)\\
&\text{Else: forward B} = 00 && (25)\\
& && (26)
\end{aligned}
$$

The forwarding unit ensures that the latest values from previous instructions are directly forwarded to the ALU, minimizing pipeline stalls.This includes both the MEM and EX hazard.

- The ALU performs computations like addition, subtraction, bitwise operations, and logical comparisons.

To resolve data hazards, the forwarding unit ensures that the most recent data is used in execution without waiting for it to be written back.

Pipeline Register: `EX/MEM Register`

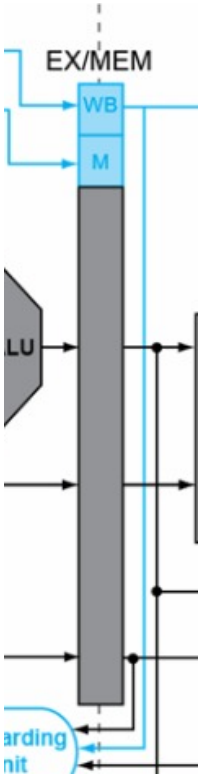

Fig. 14: Block diagram of the EX/Mem Pipeline Register

The **EX/MEM** pipeline register serves as a bridge between the Execution (EX) stage and the Memory (MEM) stage. It stores the results of computations from the ALU, branch information, and control signals needed for the subsequent memory access and write-back stages.

The register receives data from multiple sources:

- **ALU Result:** The computed result from the ALU, which may be used for memory access or written back to a register.
- **Branch Target Address:** The target address computed for branch instructions, coming from the adder unit.
- **Zero Flag:** A signal from the ALU indicating whether the result is zero, which is essential for branch decision-making.
- **Register Data:** The value stored in `rd`, forwarded from the ID/EX stage, which may be used for store instructions.
- **Destination Register:** The destination register (`Rd`), needed for write-back and forwarding decisions.

Additionally, the control signals are forwarded to ensure proper memory access and write-back functionality.

- **Memory Control Signals:**
  - `MemRead`: Specifies if a memory read operation is required.
  - `MemWrite`: Determines if data should be written to memory.
  - `Branch`: Indicates if a branch instruction is being executed.
- **Write-Back Control Signals:**
  - `MemtoReg`: Controls whether the result from memory or the ALU is written back.
  - `RegWrite`: Determines if a register write operation should be performed.

On every positive edge of the clock, the register updates its stored values:

$$\text{ALU\_data\_out} \leftarrow \text{ALU\_data} \tag{27}$$
$$\text{rd\_data\_out} \leftarrow \text{rd\_data} \tag{28}$$
$$\text{branch\_target\_out} \leftarrow \text{branch\_target} \tag{29}$$
$$\text{zero\_out} \leftarrow \text{zero} \tag{30}$$
$$\text{MemtoReg\_out} \leftarrow \text{MemtoReg} \tag{31}$$
$$\text{regwrite\_out} \leftarrow \text{regwrite} \tag{32}$$
$$\text{branch\_out} \leftarrow \text{branch} \tag{33}$$
$$\text{MemRead\_out} \leftarrow \text{MemRead} \tag{34}$$
$$\text{MemWrite\_out} \leftarrow \text{MemWrite} \tag{35}$$
$$\text{EX\_MEM\_rd} \leftarrow \text{Rd} \tag{36}$$

The `EX_MEM_rd` output is essential for the forwarding unit, allowing it to resolve data hazards by ensuring the most recent values are used in execution. This value is also sent to the MEM/WB register for later write-back operations.

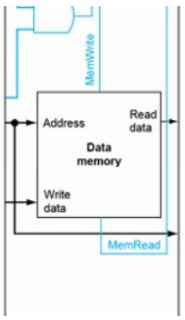

Fig. 15: Block diagram of the Mem Pipeline Stage

4) MEM :Memory Stage: The **Memory (MEM) stage** is responsible for accessing data from memory for `load` and `store` instructions. The operations in this stage are:

- If the instruction is a **load** (`ld`), the value from the computed memory address is read.
- If the instruction is a **store** (`sd`), the value from the register is written to memory.
- For arithmetic/logical instructions, this stage simply passes the ALU result forward.

The key components of the MEM stage are:

- **Data Memory:** Stores and retrieves values for load/store instructions.

- **Branch Logic:** Uses the zero flag and branch condition to decide whether to update the PC.

- **Pipeline Register (EX/MEM):** Stores intermediate values before passing them to the next stage.

The following operations occur in the MEM stage:

$$\text{Memory Address} = \text{ALU Output} \tag{37}$$
$$\text{Write Data} = \text{Register Value (for store)} \tag{38}$$
$$\text{Read Data} = \text{Memory[Memory Address]} \tag{39}$$
$$\text{(if load instruction)} \tag{40}$$

The memory operations are controlled by:

- **MemRead:** Enables reading from memory.

- **MemWrite:** Enables writing to memory.

The outputs of the MEM stage are:

- **ALU Result** (forwarded to WB stage for arithmetic instructions).

- **Memory Data** (read from memory for load instructions).

- **Control Signals** (forwarded to the MEM/WB register).

Pipeline Register: `MEM/WB Register`

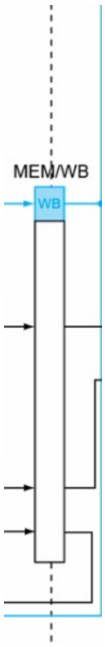

Fig. 16: Block diagram of the MEM/WB Pipeline Register

The **MEM/WB register** is the final pipeline register that holds the necessary data and control signals before the **Write-Back (WB) stage**. It ensures that data is correctly written to the register file.

The inputs to this register come from the MEM stage and include:

- **ALU Data** - The result of an arithmetic or logical operation.

- **Read Data** - Data loaded from memory (if the instruction is a load).

- **RegWrite** - Determines if a register needs to be written.

- **MemtoReg** - Selects whether the WB stage writes memory data or ALU result.

- **Destination Register (rd)** - The register to which the result will be written.

On each clock cycle, the MEM/WB register transfers its inputs to outputs:

$$\text{MEM\_WB\_read\_Data\_out} \leftarrow \text{read\_Data} \tag{41}$$
$$\text{MEM\_WB\_ALU\_data\_out} \leftarrow \text{ALU\_data} \tag{42}$$
$$\text{MEM\_WB\_MemtoReg\_out} \leftarrow \text{MemtoReg} \tag{43}$$
$$\text{MEM\_WB\_regwrite\_out} \leftarrow \text{regwrite} \tag{44}$$
$$\text{MEM\_WB\_rd} \leftarrow \text{EX\_MEM\_rd} \tag{45}$$

The Write-Back stage uses the following logic:

- If MemtoReg = 1: The data from memory (MEM_WB_read_Data_out) is written back to the register file.

- If MemtoReg = 0: The ALU result (MEM_WB_ALU_data_out) is written back to the register file.

The destination register MEM_WB_rd ensures that the correct register is updated.

*5) **Write-Back (WB) Stage**:*

- The primary function of the WB stage is to **write the computed result**—either from the **ALU** or **memory**—back to the register file.

- The output from the memory stage is selected based on the **control signal** of the `Write_back` multiplexer.

- If **MemtoReg** is set to `1`, the operation is a **load**, and data from memory is written to the destination register. Otherwise, the ALU result is written back.

- The selected value is stored in two locations:
  - **Destination Register (`rd`)** – Holds the final result of the instruction execution.
  - **ID/EX Pipeline Register** – Used for forwarding and hazard resolution in subsequent cycles.

Pipeline Register: Since this is the **final stage** of the pipeline, no additional register is needed for further propagation.

VI. Hazards in Pipelining and Their Resolution

In our pipeline, we encounter three main types of hazards:

1) **Load-Use Hazard** – Solved by Stalling (Hazard Detection Unit)
2) **General Data Hazard (RAW - Read After Write)** – Solved by Forwarding (Forwarding Unit)
3) **Control Hazards (Branch Hazards)** – Solved using branch prediction and flushing methods.

A. Load-Use Hazard

1) When does it occur?: A load-use hazard occurs when an instruction tries to use a register value that is still being loaded from memory in a previous instruction.

```
ld x2, 0 (x3)
load the value into x2 (MEM stage) \\
add x4, x2, x5
needs x2, but x2 is not yet ready \\
```

Since `x2` is still being fetched from memory in the `MEM` stage, the next instruction cannot use it immediately in the `EX` stage.

2) Detection (Hazard Detection Unit): The following condition detects a load-use hazard in our Verilog code:

```
if (((IF_ID_rs1 == ID_EX_rd) ||
(IF_ID_rs2 == ID_EX_rd)) &&
ID_EX_MemRead && (ID_EX_rd != 0))
begin
    stall = 1;
    IF_ID_Write = 0;
    PC_Write = 0;
end
```

- If the `ID_EX` stage is performing a **memory read** (`MemRead = 1`) and - The destination register (`rd`) is required in the `IF/ID` stage, - Then we **stall the pipeline** (stop PC and all other registers are flushed).

3) Resolution (Stalling): To resolve the hazard, we insert a stall cycle:

1) Set `stall = 1`, stopping new instruction fetch.
2) Freeze PC (`PC_Write = 0`) and all register updates/flushes to 0.
3) Allow data to be written before continuing.

Cycle	LW (x2)	ADD (x4)	SUB	AND	OR
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	**Stall**	ID	IF	
5	WB	EX	ID	IF	
6		MEM	EX	ID	IF

Fig. 17: Load-Use Hazard Handling (Pipeline Stall)

B. General Data Hazard (RAW - Read After Write)

1) When does it occur?: A general data hazard occurs when an instruction depends on a register that is still being written by a previous instruction.

```
add x2, x3, x4
// x2 = x3 + x4  (EX stage)
sub x5, x2, x6
// Needs x2, but x2 is still in EX stage!
```

The value of `x2` is not yet written when the `sub` instruction is in the EX stage, leading to incorrect results.

2) Detection (Forwarding Unit): Our forwarding unit checks:

```
if (EX_MEM_regWrite && (EX_MEM_rd != 0) &&
    (EX_MEM_rd == ID_EX_rs1))
     fwd_A_reg = 10; // Forward from EX/MEM stage
else if (MEM_WB_regWrite && (MEM_WB_rd != 0) &&
    (MEM_WB_rd == ID_EX_rs1))
     fwd_A_reg = 01; // Forward from MEM/WB stage
else
     fwd_A_reg = 00; // No forwarding needed
```

If the previous instruction is writing to a register and its destination (`rd`) matches the current instruction's **source** (`rs1` or `rs2`), forwarding is applied.

3) Resolution (Forwarding): Instead of waiting, we **forward** the latest value:

1) **Forward from EX/MEM stage** if available (`fwd_A = 10, fwd_B = 10`).
2) **Forward from MEM/WB stage** if needed (`fwd_A = 01, fwd_B = 01`).

Cycle	ADD (x2)	SUB (x5)	AND	OR	XOR
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX (**Forward x2**)	ID	IF	
5	WB	MEM	EX	ID	IF
6		WB	MEM	EX	ID

Fig. 18: Data Hazard Handling (Forwarding)

C. Control Hazard (Branch Hazard)

1) When does it occur?: A control hazard occurs when the pipeline does not know **whether a branch should be taken or not**, causing incorrect instruction execution.

```
beq x1, x2, LABEL
// Branch if x1 == x2
add x3, x4, x5
// This instruction is fetched before
//knowing the branch result
```

2) How do we solve it?: We use the **Flushing Logic**:

- The pipeline assumes that **no branch is taken** (fetches next sequential instructions).
- If the branch is taken, we **flush** the incorrect instruction i.e when we get our branch output at the MEM stage (in our implementation **and** `gate` is in MEM stage) and we get `PC_src = 1` , which means branch should be taken. But by that time the next instruction is being Decoded/Executed and we don't want it.
- So we flush our registers on the posedge of the next clock cycle (the clock cycle just after the cycle in which `PC_src = 1`) . The `IF/ID`,`ID/EX`,`EX/MEM` register's are set to 0.

3) Implementation in Our Code: In our implementation:

- If a branch is taken i.e `PC_src = 1` we **flush the registers and the PC**.
- We give `Pc_src` as input to`IF/ID`,`ID/EX`,`EX/MEM` register's and `PC_mux` as select line.
- This ensures that incorrect instructions do not proceed further.

$$\texttt{PCsrc}\ (\text{Flush}) \leftarrow \text{branch}\ \&\ \text{zero} \tag{46}$$

D. Comparison of Hazard Solutions

Hazard Type	Cause	Solution
Load-Use Hazard (Data Hazard)	Load instruction result not available in EX stage	**Pipeline Stall** (Hazard Detection Unit)
General Data Hazard (RAW)	ALU result not written before next instruction reads it	**Forwarding** (Forwarding Unit)
Control Hazard (Branch Hazard)	Branch outcome unknown at fetch stage	**IF Flush Mechanism** (Assume no branch, flush if needed)

TABLE II: Comparison of Pipeline Hazard Solutions

VII. Conclusion

In our Verilog implementation:

- The **Hazard Detection Unit** ensures correctness by stalling when required.
- The **Forwarding Unit** optimizes performance by reusing values without waiting.
- The **Flush Mechanism** efficiently handles control hazards by assuming no branch and flushing incorrect instructions if needed.
- These methods together **enable efficient pipelining**, minimizing stalls while ensuring correct execution.

VIII. Contributions of Team Members

The project was a collaborative effort of all three members. Most of the time, we worked on the same files, discussing logic and implementing it together. So, this makes it tough to claim anything any task as one member's work. However, based on the major contribution of one person in any task we can divide the tasks as follows:

- **Srihari**:
  - [Design Planning, Theory and Implementation matching, Sequential + Pipeline Debugging, Sequential Wrapper]
- **Vedant**:
  - [Pipeline registers, Pipeline Wrapper implementations, module integration]
- **Sarvesh**:
  - [Hazard Resolution Blocks, Sequential + Pipelined testcases, Project Report]

TABLE III: Comparison of Sequential Execution, Basic Pipelining, and Enhanced Pipelining with Forwarding and Hazard Detection

Stage	Sequential Execution	Basic Pipelining	Pipelining with Forwarding and Hazard Detection
Performance	Instructions execute one after another	Instructions overlap, improving throughput	Optimized for throughput
Dependency Handling	No dependencies	Dependencies cause stalls due to lack of data availability	Forwarding allows direct data transfer, reducing stalls
Data Hazard Management	Not applicable	Data hazards cause pipeline stalls	Forwarding minimizes data hazards; Hazard detection ensures correct execution
Control Hazard Management	No branch prediction needed	Branches cause stalls due to delay in decision-making	Some control hazards remain, but pipeline flushing and branch prediction help mitigate delays
Forwarding Multiplexers	Not needed	Multiplexers are needed	Added to resolve Read-After-Write (RAW) hazards by forwarding data
Hazard Detection Unit	Not needed	Pipeline stalls frequently due to missing values	Detects potential hazards and stalls pipeline only when necessary
Overall Throughput	Low	Improved, but frequent stalls limit efficiency	High, as forwarding minimizes stalls and hazard detection improves stability