IPA Project Report

Team Members:
Soham Jahagirdar
2023102046

Aryan Agrawal
2023102050

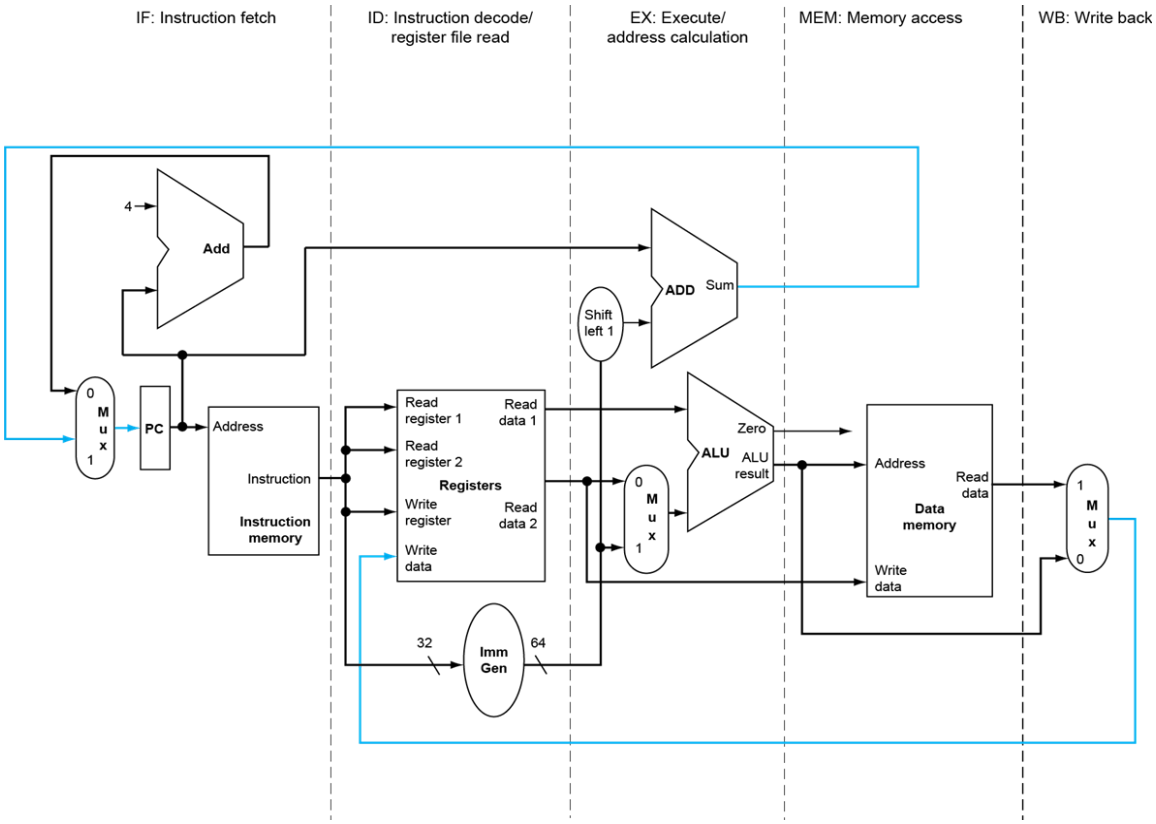
Sai Rithvik
2023102060

Overview:

- The aim of the project is to develop a processor architecture design based on the RISC-V architecture using Verilog.

- This report describes the design details of the various stages of the processor architecture.

Sequential Processor Design:

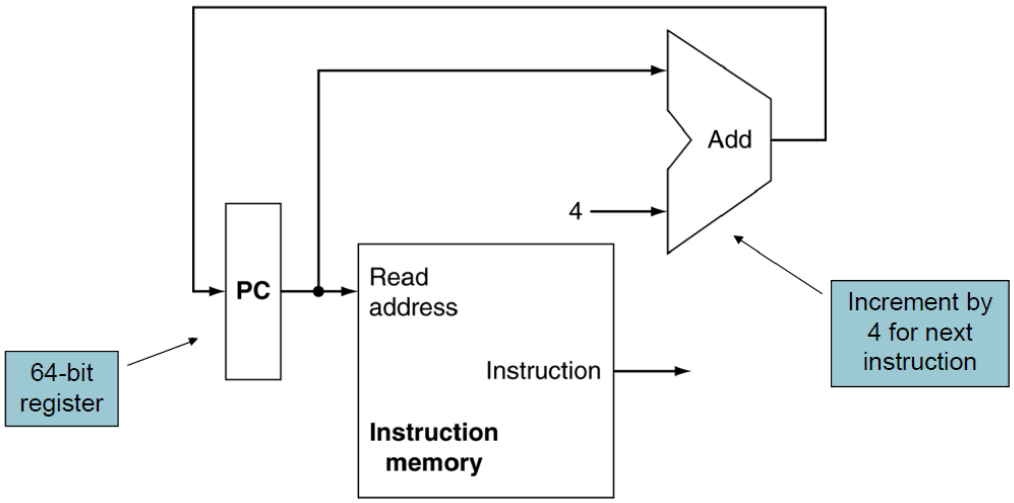


The sequential design consists of 5 stages

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

Module Description:

Instruction Fetch:



The **instruction fetch stage** in a RISC-V processor involves retrieving the next instruction from memory and updating the **Program Counter (PC)** to point to the subsequent instruction.

Working:

1. **Fetching the Instruction**:

   The **instruction memory** unit reads an instruction from memory using the **PC** as the address.

2. **Updating the Program Counter (PC)**:

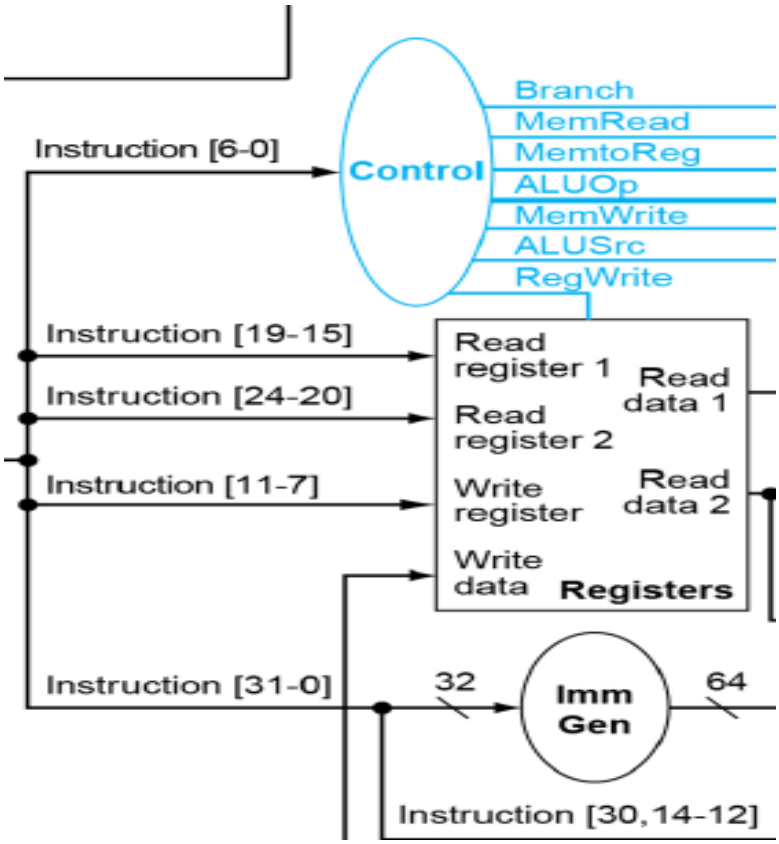
   The **PC** is stored in a **64-bit register**, which holds the address of the current instruction.

   To move to the next instruction the PC is updated as follows**,**

   For Non-Branch Operation: **PC = PC + 4**

   For Branch: **PC = PC + 2 *Immediate**

Instruction Decode



The **instruction decode stage** in a RISC-V processor is responsible for interpreting the fetched instruction, extracting its fields, and preparing necessary control signals for execution.

Working:

- **Register File**:
  - The processor has **32-64 bit registers**, stored in a **register file**.
  - The register file has **two read ports** and **one write port** and requires a write control signal to update a register.
  - The register file provides us with the values of rs1 and rs2
  - After the ALU result is calculated it is written back to the register file.

- **Immediate Generator**:
    - Many instructions, like **load, store, and branch**, use an **immediate value** instead of a second register operand.
    - The **immediate generator** extracts the 32-bit instruction and properly sign extends the immediate value.
    - This is very important to calculate target address for Branch Instructions.
- **Control Unit**:

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Working:

- The **control unit** interprets the **opcode** and generates control signals to guide the processor's execution.

  It generates the following Control Signals:

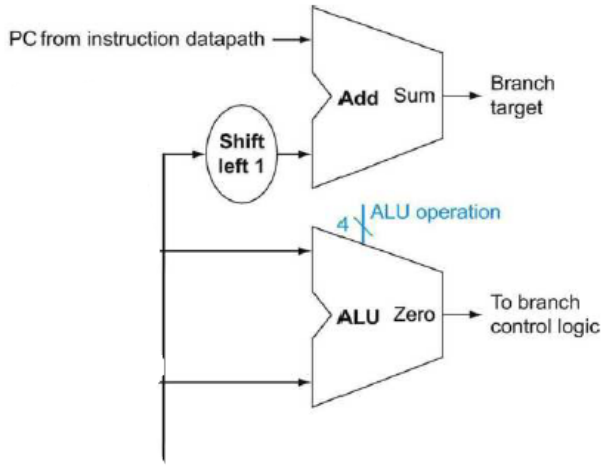
- **RegWrite** → Enables writing to the register file.
- **MemtoReg** → Selects whether the data written to the register file comes from memory (`ld`) or the ALU result (R-type and I-type).

- **ALUSrc** → Chooses between a register value or an immediate operand as the second input to the ALU.
- **MemRead** → Enables reading from memory (used in `ld` instruction).
- **MemWrite** → Enables writing to memory (used in `sd` instruction).
- **Branch** → Activates when a branch instruction (`beq`) is encountered.
- **ALUOp** → Specifies the operation for the ALU based on the instruction type (R-type, I-type, branch, etc.).

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

This decode stage ensures the proper data and control signals are prepared for the **execute stage**, where actual computation or memory access occurs.

Execute



The **execute stage** in a RISC-V processor is responsible for performing arithmetic, logic, memory address calculations, and branch evaluations.

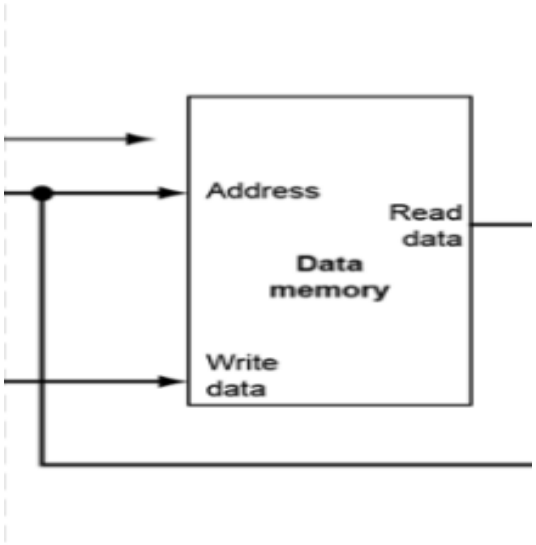
Working:

1. **ALU Operations (R-type and I-type Instructions)**
    - The **Arithmetic Logic Unit (ALU)** performs operations like addition, subtraction, AND, and OR.
    - The ALU is controlled by a **4-bit control signal ALUOp**, which determines the operation to be executed.
    - **R-type instructions** (e.g., `add x1, x2, x3`) take two operands from the **register file** and produce a result.

    - For **load (`ld`) and store (`sd`) instructions**, the ALU calculates the **effective memory address** by adding the **base register** and the **sign-extended immediate offset**.
    - The calculated address is sent to the **memory unit**, where data is either read (for loads) or written (for stores).
2. **Branch Execution (Control Flow Instructions)**
    - Branch instructions (`beq`) require:
        - **Comparing two registers** using the ALU (by performing subtraction).
        - Checking the **Zero flag** to determine if the values are equal.
        - If the branch condition is met, the **PC is updated** to the branch target address; otherwise, it continues sequentially.
    - The **branch target address** is computed using the **PC** and the **sign-extended immediate (shifted left by 1 bit)**.

The **execute stage** is crucial as it determines the core computation of an instruction, whether performing an arithmetic operation, accessing memory, or deciding program flow.

Memory Access



The memory stage in a RISC-V processor is responsible for accessing data memory for load and store instructions. This stage interacts with the data memory unit and determines whether data should be read from or written to memory.

Working:

Memory Access (Load and Store Instructions)

- **Load (`ld`) instructions**:
    - The ALU computes the memory address.
    - The address is sent to the **data memory unit**, which reads the value from memory.
    - The retrieved data is forwarded to the next stage for writing into a register.
- **Store (`sd`) instructions**:
    - The ALU computes the memory address.
    - The **data memory unit** writes the value from a register to the computed address.

Write Back

In a sequential processor, the write-back (WB) stage is the final step of instruction execution, where the result of an operation is written back to the register file.

Working:

1. Writing Results to Registers
    - If an instruction produces a result (e.g., arithmetic/logical operations, load instructions), the computed value is written to the destination register.
    - The register file updates at the end of the clock cycle, ensuring the value is available for the next instruction.
2. Multiplexer for Write Data Selection
    - A multiplexer selects the source of the data to be written:
        - ALU result (for arithmetic/logical instructions).
        - Data memory output (for load instructions).
    - The selected value is then sent to the destination register.
3. Control Signals for Write-Back
    - The register write enable signal ensures that only instructions requiring a register update perform write-back.
    - Store and branch instructions do not perform write-back, as they either modify memory or alter control flow.

Since a sequential processor executes each instruction one at a time, the write-back stage must fully complete before the next instruction begins execution.

The project involves implementing the following instructions:

- **ADD** (Addition)
- **SUB** (Subtraction)
- **AND** (Bitwise AND)
- **OR** (Bitwise OR)
- **LD** (Load)
- **SD** (Store)
- **BEQ** (Branch if Equal)

1. **ADD** (Addition)

Format (R-Type Instruction)

ADD rd rs1 rs2

Operation: Adds the values from registers `rs1` and `rs2`, storing the result in `rd`.

Working:

1. **Instruction Fetch:**
   - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
   - **Instruction Memory** fetches the `ADD` instruction.
   - **PC increments by 4** to point to the next instruction (`PC = PC + 4`).
2. **Instruction Decode:**
   - The **Control Unit** decodes the instruction and identifies it as an `ADD` operation.
   - Control signals are set:
     - `RegWrite = 1` (enable writing to `rd`).
     - `ALUOp = 10` (R-type operation).

3. **Register Read:**
   - The **Register File** reads values from **rs1** and **rs2**.

4. **ALU Execution:**
    - The **ALU** performs the addition:

Result = rs1+rs2

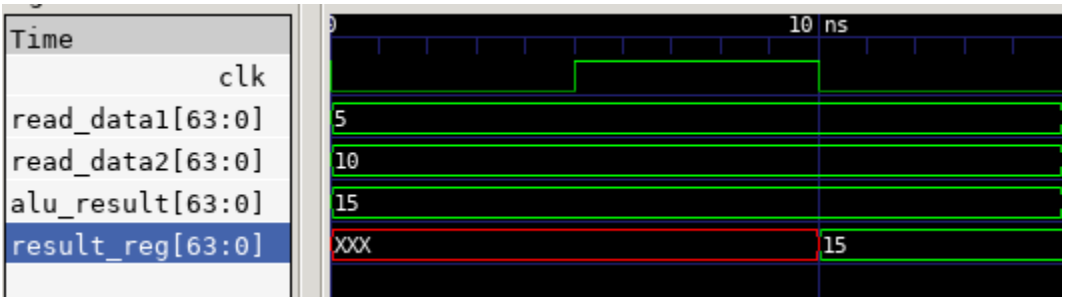
5. **Write Back:**
    - The computed **Result** is written back to **rd** in the **Register File**.

For input instruction : **add x3 x2 x1**

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000000100010000000110110011;// add x3, x2, x1
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```

```
a1_input =                    5, a2_input =                   10, alu_result =                   15,
```

We get GTK Wave output as on 2nd positive edge of clock



2. **SUB** (Subtraction)
Format (R-Type Instruction)

SUB rd,rs1,rs2

Working:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `SUB` instruction.
    - **PC increments by 4** to point to the next instruction (`PC = PC + 4`).
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as a `SUB` operation.
    - Control signals are set:
        - `RegWrite = 1` (enable writing to `rd`).
        - `ALUOp = 10` (R-type operation).
3. **Register Read:**
    - The **Register File** reads values from **rs1** and **rs2**.
4. **ALU Execution:**
    - The **ALU** performs subtraction:

$$\textbf{Result} = \textbf{rs1} - \textbf{rs2}$$

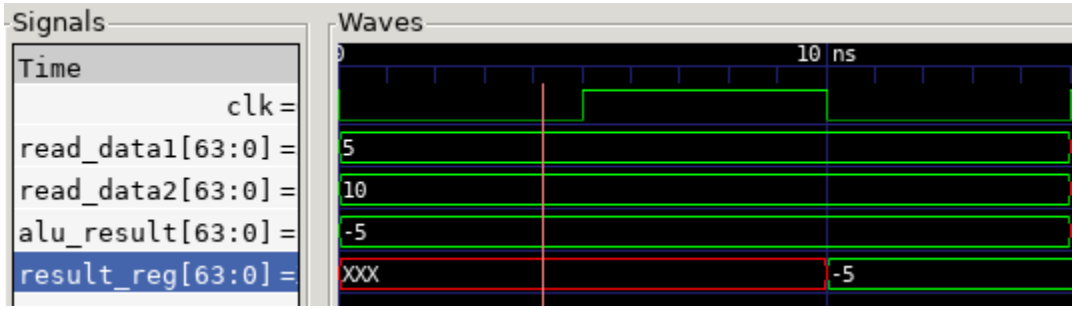
5. **Write Back:**
    - The computed **Result** is written back to **rd** in the **Register File**.

For input instruction : **sub x3 x2 x1**

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b01000000000100010000000110110011;// sub x3, x2, x1
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```

We get output as:

```
a1_input =                    5, a2_input =                   10, alu_result =                   -5,
```



3. **AND** (Bitwise AND)
Format (R-Type Instruction)

AND rd rs1 rs2

●**Operation:** Performs a bitwise **AND** operation between the values in `rs1` and `rs2`, then stores the result in `rd`.

Working:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `AND` instruction.
    - **PC increments by 4** (`PC = PC + 4`).
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as an `AND` operation.
    - Control signals are set:
        - `RegWrite = 1` (enable writing to `rd`).
        - `ALUOp = 10` (R-type operation).
3. **Register Read:**
    - The **Register File** reads values from **rs1** and **rs2**.
4. **ALU Execution:**
    - The **ALU** performs the bitwise AND operation:

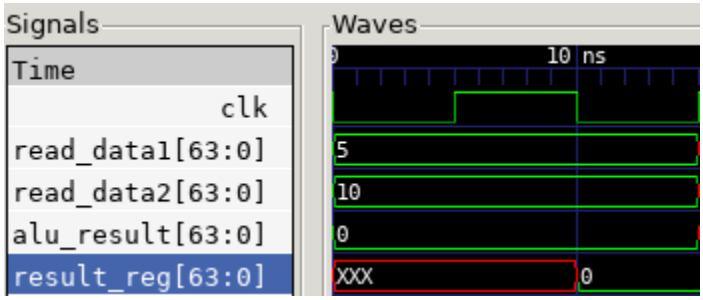
Result = rs1&rs2

5. **Write Back:**
    - The computed **Result** is written back to **rd** in the **Register File**.

For input instruction : **and x3 x2 x1**

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000000100010111000110110011;// and x3, x2, x1
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```

```
Time =                0, Reset = 1, ReadData1 =                5, ReadData2 =               10, ALU Result =                0
```



4. **OR** (Bitwise OR)
Format (R-Type Instruction)

OR rd,rs1,rs2

Operation: Performs a bitwise **AND** operation between the values in `rs1` and `rs2`, then stores the result in `rd`.

Working:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `AND` instruction.
    - **PC increments by 4** (`PC = PC + 4`).
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as an `AND` operation.
    - Control signals are set:
        - `RegWrite = 1` (enable writing to `rd`).
        - `ALUOp = 10` (R-type operation).
3. **Register Read:**
    - The **Register File** reads values from **rs1** and **rs2**.
4. **ALU Execution:**
    - The **ALU** performs the bitwise AND operation:

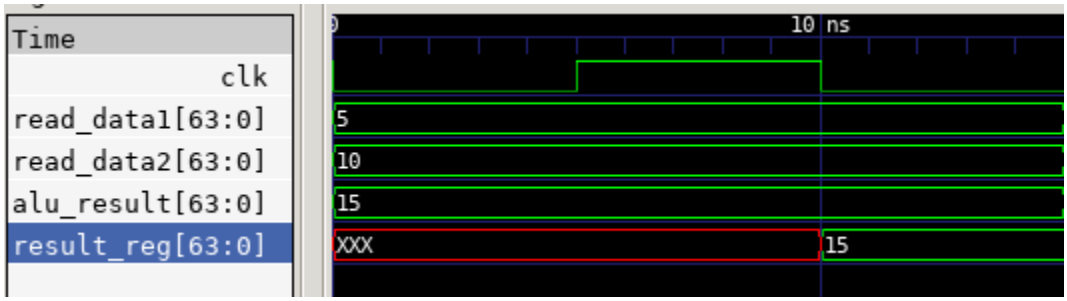
      Result = rs1 | rs2

5. **Write Back:**
    - The computed **Result** is written back to **rd** in the **Register File**.

For input instruction : **or x3 x2 x1**

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000000100010110000110110011;// or x3, x2, x1
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```

```
a1_input =                    5, a2_input =                   10, alu_result =                   -5,
```



5. **LD** (Load Word - lw)
Format (I-Type Instruction)

LD rd, offset(rs1)

Operation: Loads a word from memory at the address `rs1 + offset` and stores it in `rd`.

Execution Steps:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `LD` instruction.
    - **PC increments by 4** (`PC = PC + 4`).
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as a **LOAD** operation.
    - Control signals are set:
        - `RegWrite = 1` (enable writing to `rd`).
        - `MemRead = 1` (enable memory read).
        - `ALUSrc = 1` (use immediate as ALU input).
        - `MemtoReg = 1` (data comes from memory).
3. **Register Read:**
    - The **Register File** reads the value of **rs1**.
4. **Immediate Generation:**
    - The **Immediate Generator** extracts the **offset** from the instruction.
5. **Address Calculation:**

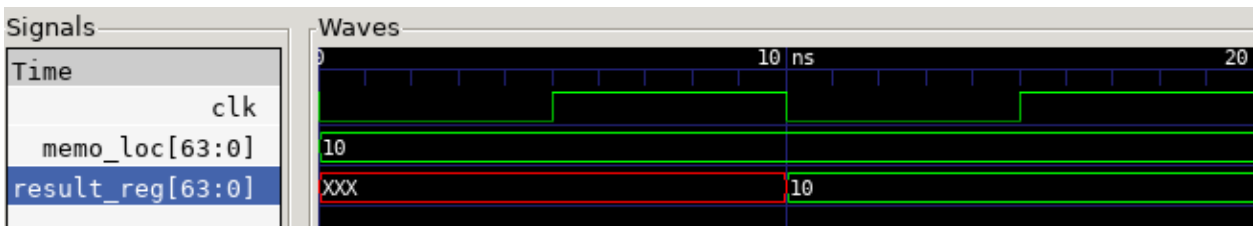
- The **ALU** computes the effective memory address:

Address = rs1+Offset

6. **Memory Access:**
   - The **Data Memory** reads data from the computed **Address**.
7. **Write Back:**
   - The loaded **data** is stored in **rd** in the **Register File**.

For input instruction : **ld x3 0(x2)**

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000000000001001100011000011;// ld x3 0(x2)
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```



6. **STORE**
Format (S-Type Instruction)

SD rs2, offset(rs1)

Operation: Stores the value from `rs2` into memory at the address **`rs1 + offset`**.

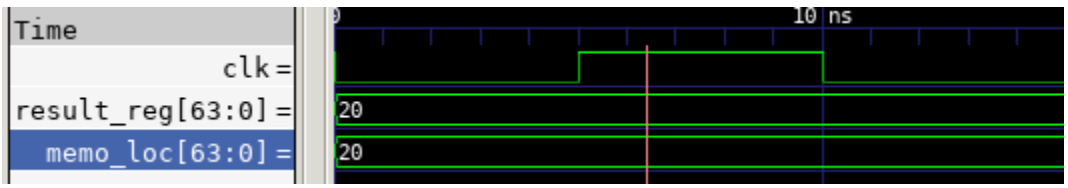
Execution Steps:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `SW` instruction.
    - **PC increments by 4** (`PC = PC + 4`).
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as a **STORE** operation.
    - Control signals are set:
        - `MemWrite = 1` (enable memory write).
        - `ALUSrc = 1` (use immediate as ALU input).
3. **Register Read:**
    - The **Register File** reads the values of **rs1** (base address) and **rs2** (data to store).
4. **Immediate Generation:**
    - The **Immediate Generator** extracts the **offset** from the instruction.

5. **Address Calculation:**
    - The **ALU** computes the effective memory address:

      Address = rs1+Offset
6. **Memory Write:**
    - The **Data Memory** writes the value of **rs2** to the computed **Address**.

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000001100010011000000100011;// sd x3 0(x2)
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```



7. **BEQ** (Branch if Equal)
Format (B-Type Instruction)

BEQ rs1, rs2, offset

Operation: If `rs1 == rs2`, the **Program Counter (PC)** jumps to **`PC + offset`**; otherwise, it proceeds to the next instruction.

Execution Steps:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** provides the instruction address to **Instruction Memory**.
    - **Instruction Memory** fetches the `BEQ` instruction.
    - **PC increments by 4** (`PC = PC + 4`) as a placeholder for the next instruction.
2. **Instruction Decode:**
    - The **Control Unit** decodes the instruction and identifies it as a **Branch Equal (BEQ)** operation.
    - Control signals are set:
        - `Branch = 1` (enable branch control).
        - `ALUOp = 01` (ALU performs a subtraction for comparison).
3. **Register Read:**
    - The **Register File** reads the values of **rs1** and **rs2**.
4. **ALU Execution:**
    - The **ALU** performs subtraction to check equality:

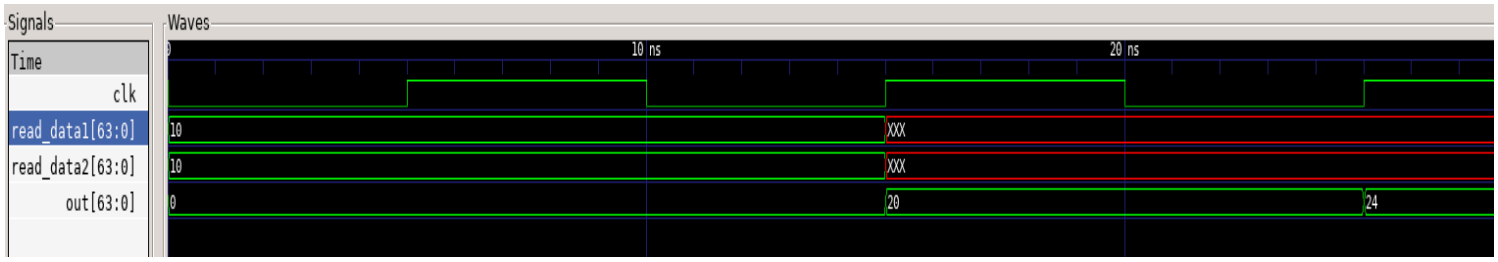
$$\textbf{Result} = \textbf{rs1} - \textbf{rs2}$$

5. **Branch Decision:**

   - If **Result == 0** (i.e., `rs1 == rs2`), the branch is **taken**: PC=PC+Offset
   - Otherwise, the branch is **not taken**, and execution continues with the next instruction: PC=PC+4

```
module instruction_memory (
    input [63:0] addr,
    output reg [31:0] inst
);

  reg [31:0] mem[63:0];

  initial begin
    mem[0] = 32'b00000000001000001000010101100011;// beq x1, x2, 10
    mem[4] = 32'b00000000001000001000001001100011;// beq x1, x2, 4
    mem[8] = 32'b00000000001000001000000110110011; // add x3 x1 x2
  end

  always @(*) begin
    inst <= mem[addr];
  end

endmodule
```

Output:



Sum of Natural Numbers Algorithm for Testing

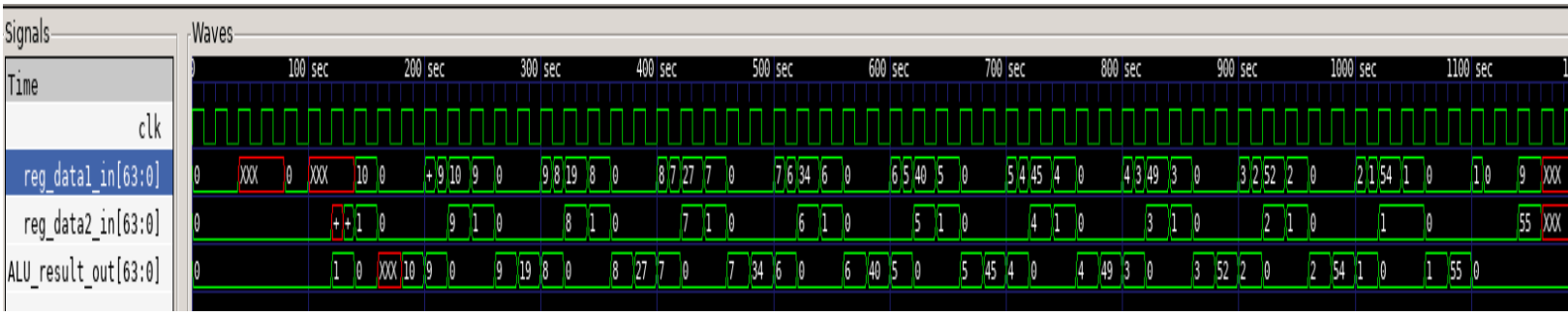
Instructions:

```
00000000000000000011000010000011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000001011100110000011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000010100110011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000010011000010001100011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000001001101010000010100110011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
01000000000110011000100110110011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
11111111001110011000110111100011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000101000010011000000100011
```

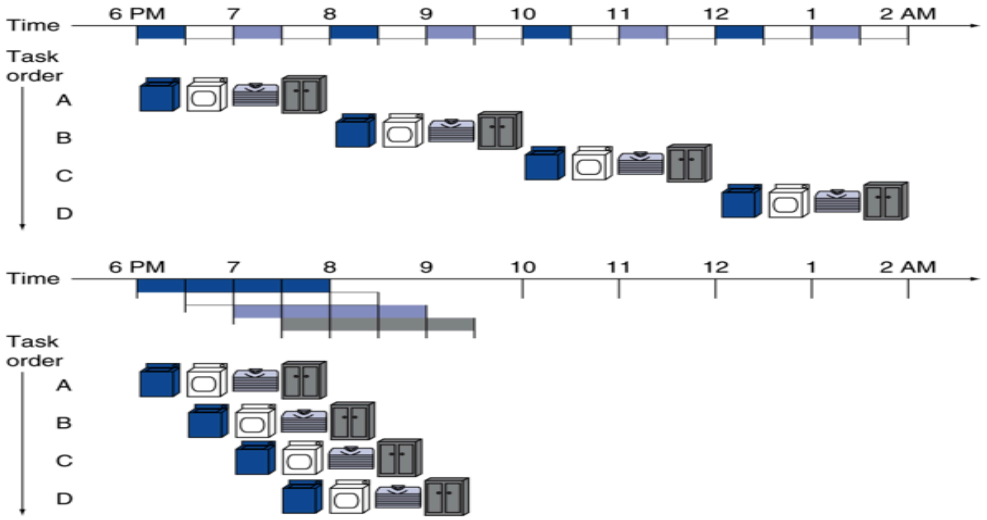
Result:

```
we are trying to find the sum of natural numbers
result =                    x, memory_location =                    x time =                    0
result =                    0, memory_location =                    x time =                   50
result =                   10, memory_location =                    x time =                   90
result =                   19, memory_location =                    x time =                  170
result =                   27, memory_location =                    x time =                  250
result =                   34, memory_location =                    x time =                  330
result =                   40, memory_location =                    x time =                  410
result =                   45, memory_location =                    x time =                  490
result =                   49, memory_location =                    x time =                  570
result =                   52, memory_location =                    x time =                  650
result =                   54, memory_location =                    x time =                  730
result =                   55, memory_location =                    x time =                  810
result =                   55, memory_location =                   55 time =                  880
```

GTK:

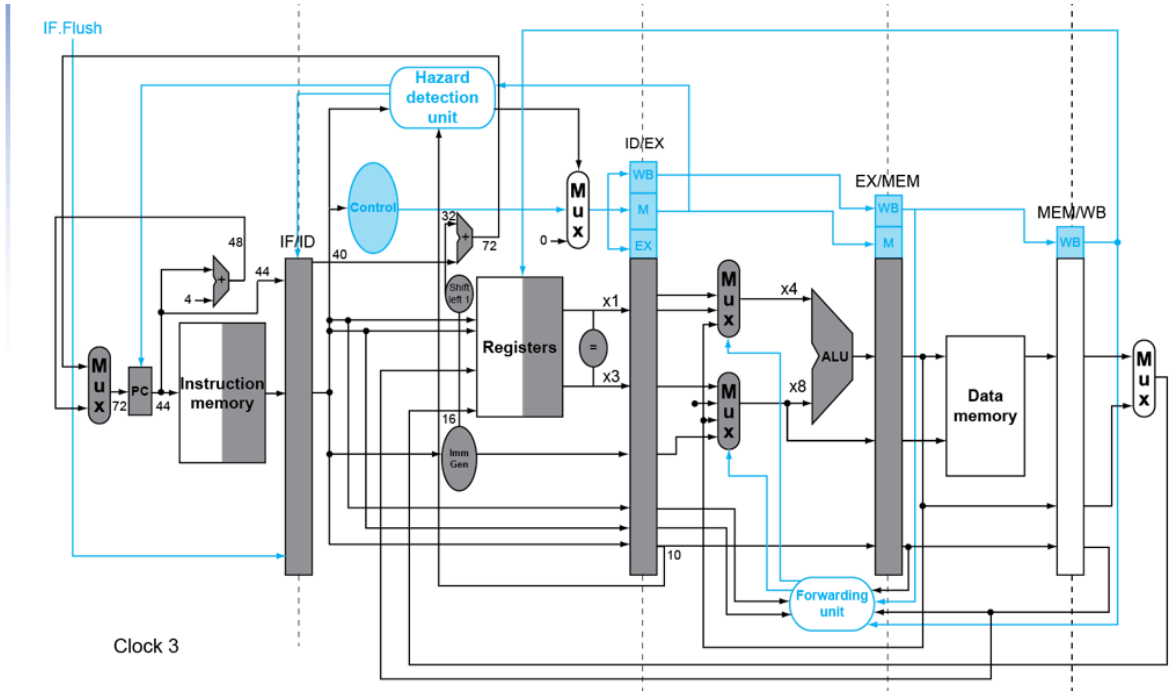


Pipelined Processor



Pipelining is a technique in modern processors that enhances execution speed by allowing multiple instructions to overlap. Instead of executing one instruction at a time, a pipelined processor begins the next instruction before the previous one is completed, improving overall throughput.

Hardware structure of pipeline:



A 5-stage pipeline consists of the following stages:

1.IF: Instruction fetch

Function:-

● Fetches the next instruction from memory using the Program Counter (PC).

● Increments the PC to point to the next instruction.

Steps:

1. PC sends address to instruction memory.

2. Instruction memory provides instruction at that address.

3. PC is updated to PC + 4 (for 32-bit instruction width).

Hardware Components Used:

● Program Counter (PC)

● Instruction Memory

● Adder (for PC + 4 calculation)

After Instruction fetch we have placed a IF/ID register which stores the results of Instruction fetch.

2.ID: Instruction decode

Function:

● Decodes the instruction to identify the operation type.

● Reads register operands (rs1, rs2) from the IF/ID register.

● Extracts immediate values (if required).

● Generates control signals for execution.

● Verifies if a load hazard has occurred or not.

Steps:

1. The opcode is used to determine the instruction type.
2. The Control Unit generates control signals and pushes into 2 x1 mux.
3. The Register File fetches values from registers and pushes it into ID/EX register .
4. The Immediate Generator extracts immediate values (for LD, SW, BEQ) and pushes it into ID/EX register
5. Hazard detection unit acts as a select line and decides whether a stall is inserted or operations from control unit are pushed to ID/EX register.

Hardware Components Used:

- Control Unit
- Register File
- Immediate Generator
- Hazard detection unit
- 2x1 MUX

After Instruction Decode we have placed a ID/EX register which stores the results of Instruction Decode.

3.EX: Execute operation

Function:

- Performs arithmetic/logic operations using the ALU.
- Computes memory addresses for LD and SW.
- Evaluates branch conditions (BEQ).
- The forwarding unit retrieves values from previous ALU results as needed to prevent hazards.

Steps:

1.The two muxes decide the inputs to alu,whether it comes from register file or previous alu results or data memory.

2. ALU Control Unit decides the ALU operation based on funct3 and funct7 from ID/EX register.

3. ALU executes the operation:

- ○ ADD/SUB/AND/OR
- ○ Address computation for LD/SW
- ○ Comparison for BEQ

and transfers the results into the EX/MEM register.

4. If BEQ, branch condition is checked.

Hardware Components Used:

- ALU
- Forwarding unit
- 2 3x2 Muxes

After Execute operation we have placed a EX/MEM register which stores the results of Execute operation.

4.MEM: Access memory operand

Function:

- Reads from or writes to Data Memory (for LD and SW).
- If a load (LD), the value from memory is read.
- If a store (SW), the value from rs2 is stored into memory.

Steps:

1. For Load (LD):

○ ALU computed address in previous stage which can be fetched from EX/MEM register.

○ Memory reads data from that address.

2. For Store (SW):

○ ALU computed addressin previous stage which can be fetched from EX/MEM register.

○ Memory writes rs2 data to that address.

Hardware Components Used:

● Data Memory

● Memory Control Logic

After access memory operation we have placed a MEM/WB register which stores the results of access memory operation.

5. WB: Write result back to register

Function:

● Writes the final ALU result or memory data back into the Register File.

Steps:

1. If the instruction is arithmetic (ADD, SUB, etc.),
○ The ALU result is written back into the register file (rd).
2. If the instruction is load (LD),
○ The value read from memory is stored in rd.

Hardware Components Used:

● Register File
● Multiplexer (to select ALU result or memory data)

Pipeline hazards (Execution Delays Due to Dependencies)

Hazards prevent the smooth execution of instructions in the pipeline. There are three types:

1. **Structural Hazards** (Hardware Conflict)
    - Occurs when two instructions compete for the same resource (e.g., ALU, Memory) simultaneously.
    - Solution: Implement separate instruction and data memory (Harvard Architecture) or use multi-port memory.
2. **Data Hazards** (Instruction Dependency Issues)
    - Occurs when an instruction relies on the outcome of a preceding instruction that has not yet finished execution.
    - Solution: Implement data forwarding or insert pipeline stalls (bubbles).

3.**Control Hazards** (Branching Issues)

● Occurs with branch instructions (e.g .BEQ, JAL), as the processor does not know the next instruction to fetch.

● Solution: Use branch prediction or delayed branching to minimize stalls.

Changes for Pipeline:

•Inserting hazard detection unit:

We have added a hazard detection unit which is used to detect load hazard and insert stall **or flush** if a load hazard has occurred.

•Inserting Forwarding Unit:

We have added a forwarding unit to resolve data hazards by forwarding the most recent values to the ALU instead of waiting for them to be written back to the register file. It determines the correct data source for ALU operands using control signals (`ForwardA` and `ForwardB`). If an operand is available in the pipeline's EX/MEM or MEM/WB stages, it is forwarded to avoid stalls. This ensures efficient instruction execution by reducing pipeline delays caused by data dependencies.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Inserting Pipeline Registers:

• The next step to pipelining is inserting the pipeline registers.

• We know that in a pipelined implementation we rearrange some of the hardware and signals in the SEQ implementation and insert pipeline register between each stage.

• These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.

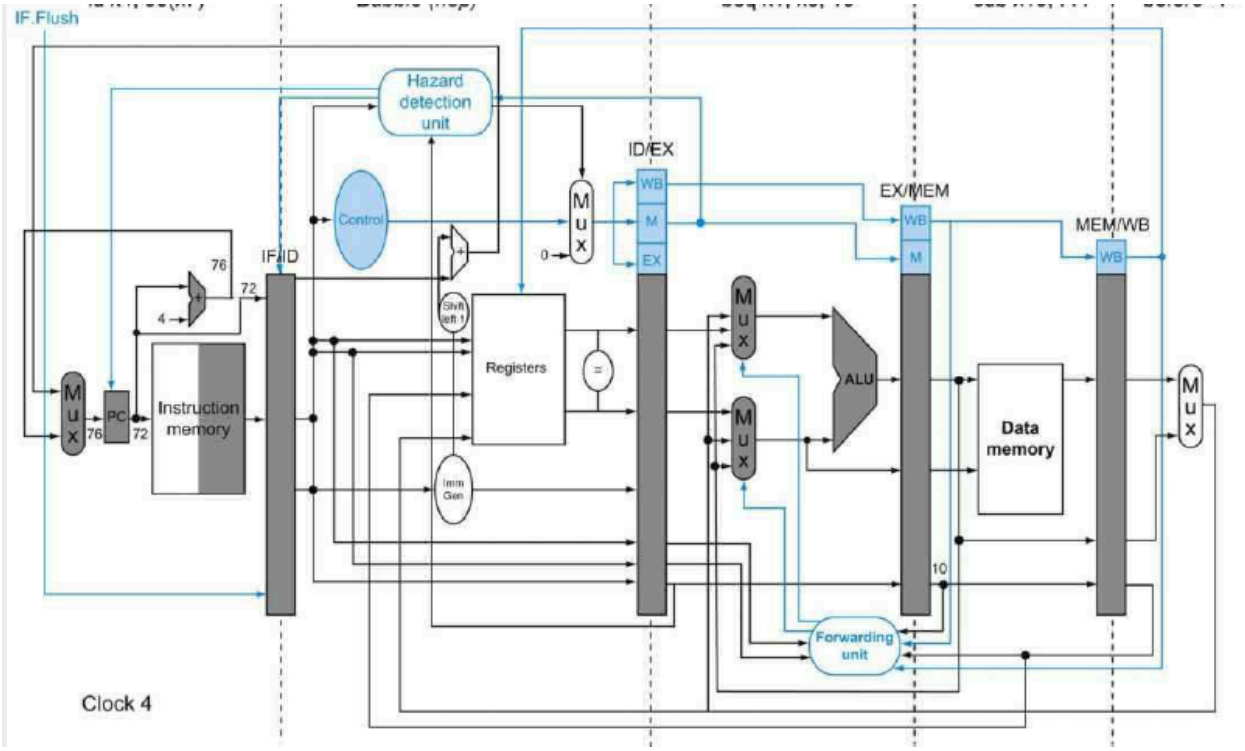
• IF/ID sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage

• ID/EX sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage .

•EX/MEM sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.

•MEM/WB sits between the memory stage and the write-back paths that supply the computed results to the register file for writing.

Inserting Extra Hardware for Dealing with Control Hazards

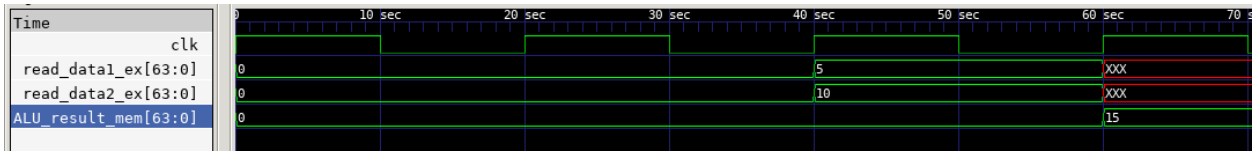


To efficiently handle control hazards in the pipeline, we introduce a **BEQ Comparator** that determines whether the branch is taken or not. If the branch is taken (`rs1 == rs2`), we **flush only the IF stage** and update the PC to the branch target (`PC = PC + 2 * imm`). If the branch is not taken, execution continues normally.

We also shift the PC + target adder to ID Stage to deal with the Hazard efficiently

ADD:

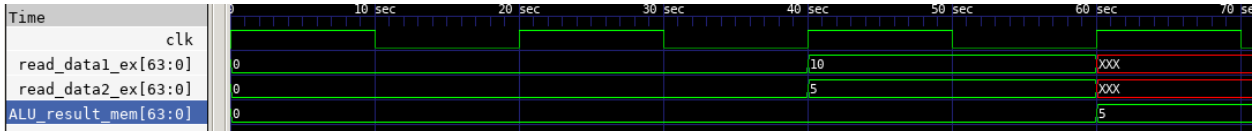
```
initial begin
  // register values
  p1.id1.r1.registers[10] = 64'd5;
  p1.id1.r1.registers[19] = 64'd10;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[5] = 64'd10;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000001001101010000010100110011; // add x10 x10 x19
```



Value of x10=x10+x19=10+5=15

SUB:

```
initial begin
  // register values
  p1.id1.r1.registers[10] = 64'd10;
  p1.id1.r1.registers[19] = 64'd5;
  // p1.id1.r1.registers[3] = 64'd5;
  // p1.id1.r1.registers[4] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[5] = 64'd10;



  //instructions
  p1.if1.im1.mem[0] = 32'b01000001001101010000010100110011; // sub x10 x10 x19
```



Value of x10=x10-x19=10-5=5

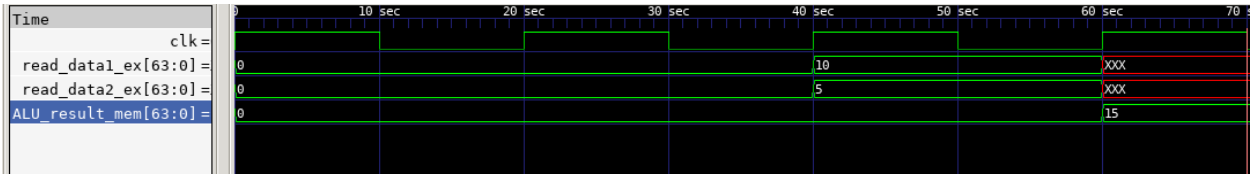
AND:

```
initial begin
  // register values
  p1.id1.r1.registers[10] = 64'd10;
  p1.id1.r1.registers[19] = 64'd5;
  // p1.id1.r1.registers[3] = 64'd5;
  // p1.id1.r1.registers[4] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[5] = 64'd10;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000001001101010111010100110011; // and x10 x10 x19
```

Value of x19=x10&x19=0

OR:

```
initial begin
  // register values
  p1.id1.r1.registers[10] = 64'd10;
  p1.id1.r1.registers[19] = 64'd5;
  // p1.id1.r1.registers[3] = 64'd5;
  // p1.id1.r1.registers[4] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[5] = 64'd10;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000001001101010110010100110011; // or x10 x10 x19
```



Value of x10=x10|x19=15

LD:

```
initial begin
  // register values
  p1.id1.r1.registers[1] = 64'd5;

  // memory values
  p1.dm1.memory[20] = 64'd16;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000000111100001011000110000011; // ld x3, 15(x1)
```

Output:

Value of x3 =value at(15 +x1)=x20=16

```
result =                    x, time =                    5
result =                   16, time =                   90
```

SD:

```
initial begin
  // register values
  p1.id1.r1.registers[1] = 64'd1;
  p1.id1.r1.registers[3] = 64'd10;

  // memory values
  p1.dm1.memory[20] = 64'd16;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000000001100001011011110100011; // sd x3, 15(x1)
```

Output:

x16= value(x3)=10

```
result =                    x, time =                    5
result =                   10, time =                   60
```

BEQ:

```
initial begin
  // register values
  p1.id1.r1.registers[1] = 64'd1;
  p1.id1.r1.registers[4] = 64'd13;

  // memory values
  p1.dm1.memory[20] = 64'd16;



  //instructions
  p1.if1.im1.mem[0] = 32'b00000000000000000000001001100011; // beq x0, x0, 4
  p1.if1.im1.mem[8] = 32'b00000000010000001000000010110011; // add x1 x1 x4
```

Output:

x1=x1+x4=13+1

```
result =                   14, time =                  130
```

Data Hazard Testing

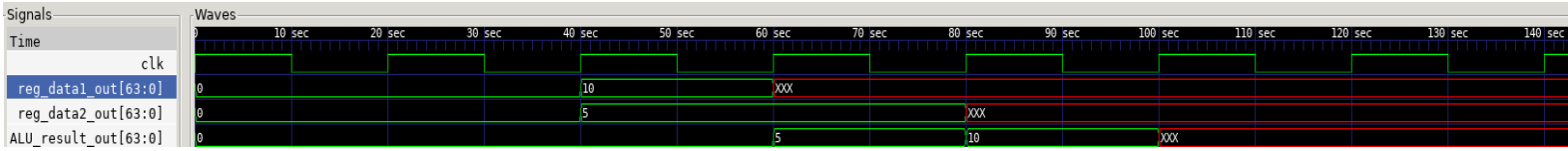
```
initial begin
  // register values
  p1.id1.r1.registers[1] = 64'd10;
  p1.id1.r1.registers[3] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;

  //instructions
  p1.if1.im1.mem[0] = 32'b01000000001100001000000100110011; // sub x2, x1, x3
  p1.if1.im1.mem[4] = 32'b00000000001100001000011100110011; // add x14, x2, x3
```

We can see Data Hazard occurs as x2 value is needed in next Instruction

Output:

```
result =                    0, time =                    5
result =                    5, time =                   40
result =                   10, time =                   60
```

GTK Output



We can see it stalls till ALU result is Calculated then only it Forwards the x2 value to the next stage

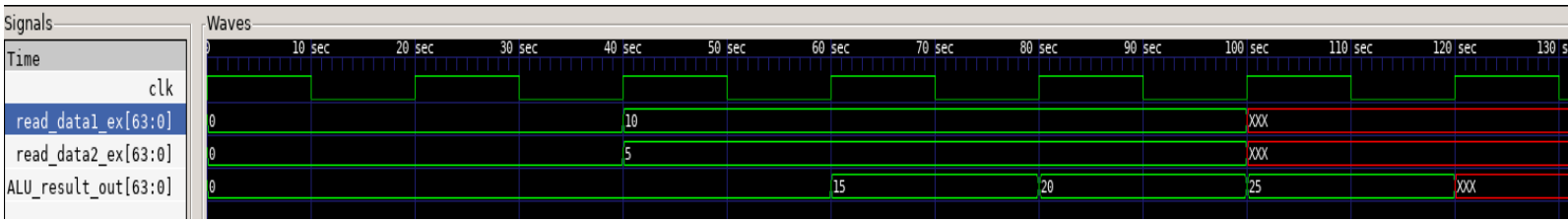
Double Data Hazard Testing

```
initial begin
  // register values
  p1.id1.r1.registers[1] = 64'd10;
  p1.id1.r1.registers[2] = 64'd5;
  p1.id1.r1.registers[3] = 64'd5;
  p1.id1.r1.registers[4] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;

  //instructions
  p1.if1.im1.mem[0] = 32'b00000000001000001000000010110011; // add x1, x1, x2
  p1.if1.im1.mem[4] = 32'b00000000001100001000000010110011; // add x1, x1, x3
  p1.if1.im1.mem[8] = 32'b00000000001000001000000010110011; // add x1, x1, x4
```

Here Double Data occurs as x1 value is used multiple times in these instructions.

Output:

```
result =                    0, time =                    5
result =                   15, time =                   40
result =                   20, time =                   60
result =                   25, time =                   80
```



Load Use Hazard Testing

```
initial begin
  // register values
  p1.id1.r1.registers[2] = 64'd5;
  p1.id1.r1.registers[5] = 64'd5;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[5] = 64'd10;


  //instructions
  p1.if1.im1.mem[0] = 32'b00000000000000001001100001000011; // ld x1,0(x2)
  p1.if1.im1.mem[4] = 32'b01000000010100001000001000110011; // sub x4, x1, x5
```

We print x4 value which is correct i.e x1-x5=10-5=5.
Code Stalls till we get x1 then Appropriate Forwarding is done.

```
result =                    x, time =                    5
result =                    5, time =                  130
```

We see we get the Result only after everything is loaded properly after stalling and forwarding.

Sum of Natural Numbers Algorithm for Testing,includes Control Hazard

```
initial begin
  // register values
  p1.id1.r1.registers[2] = 64'd9;

  // memory values
  p1.dm1.memory[0] = 64'd1;
  p1.dm1.memory[1] = 64'd10;

  //instructions
  p1.if1.im1.mem[0] = 32'b00000000000000000011000010000011; // ld x1 0(x0)
  p1.if1.im1.mem[4] = 32'b00000000000000001011100110000011; // ld x19 0(x1)
  p1.if1.im1.mem[8] = 32'b00000000000000000000010100110011; // add x10 x0 x0
  p1.if1.im1.mem[12] = 32'b00000000000010011000010001100011; // beq x19 x0 8
  p1.if1.im1.mem[16] = 32'b00000001001101010000010100110011; // add x10 x10 x19
  p1.if1.im1.mem[20] = 32'b01000000000110011000100110110011; // sub x19 x19 x1
  p1.if1.im1.mem[24] = 32'b11111111000000000000110111100011; // beq x0 x0 -6
  p1.if1.im1.mem[28] = 32'b00000000101000010011000000100011; // sd x10 0(x2)

end
```

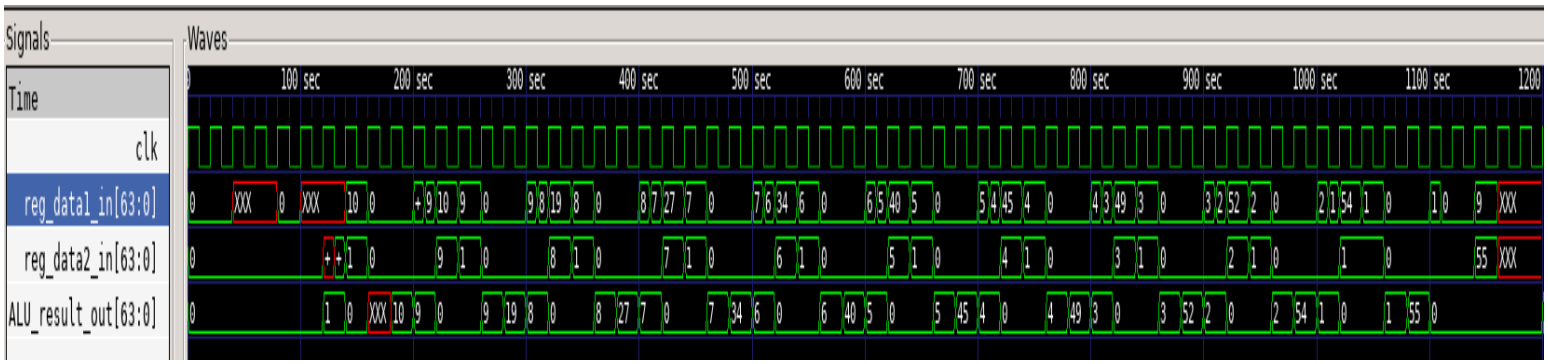
Terminal Output

Time	Result	PC_if	PC_id	Flush
5	x	0	0	0
20	x	4	0	0
40	x	8	4	0
80	x	12	8	0
100	x	16	12	0
120	x	20	16	0
140	x	24	20	0
150	0	24	20	0
160	0	28	24	1
180	0	12	0	0
190	10	12	0	0
200	10	16	12	0
220	10	20	16	0
240	10	24	20	0

260	10	28	24	1
280	10	12	0	0
290	19	12	0	0
300	19	16	12	0
320	19	20	16	0
340	19	24	20	0
360	19	28	24	1
380	19	12	0	0
390	27	12	0	0
400	27	16	12	0
420	27	20	16	0
440	27	24	20	0
460	27	28	24	1
480	27	12	0	0
490	34	12	0	0
500	34	16	12	0
520	34	20	16	0
540	34	24	20	0
560	34	28	24	1
580	34	12	0	0
590	40	12	0	0
600	40	16	12	0
620	40	20	16	0
640	40	24	20	0
660	40	28	24	1
680	40	12	0	0
690	45	12	0	0
700	45	16	12	0
720	45	20	16	0
740	45	24	20	0
760	45	28	24	1
780	45	12	0	0
790	49	12	0	0
800	49	16	12	0
820	49	20	16	0

840	49	24	20	0
860	49	28	24	1
880	49	12	0	0
890	52	12	0	0
900	52	16	12	0
920	52	20	16	0
940	52	24	20	0
960	52	28	24	1
980	52	12	0	0
990	54	12	0	0
1000	54	16	12	0
1020	54	20	16	0
1040	54	24	20	0
1060	54	28	24	1
1080	54	12	0	0
1090	55	12	0	0
1100	55	16	12	0
1110	55	16	12	1
1120	55	28	0	0
1140	55	32	28	0

GTK Output:



In this Code we verify Control Hazard is also working properly,when a beq instruction comes, PC proceeds to next stage but when we have beq is true Flushing is Happening Successfully and PC moves to target address accordingly.

Individual Contributions

Soham→
Control Unit,ALU Control,Data Memory,EX/MEM and MEM/WB Pipeline Registers,Data Hazard,Control Hazard,Project Report

Aryan→
Program Counter,Instruction Fetch,Register Files,Muxs,IF/ID and ID/EX Pipeline Registers,Load use Hazards,Project Report

Rithvik→
Connecting and Integration of all the Individual Components of Sequential and Pipelined Processor,Testing and Debugging,Control Hazard