Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

EC-220 Cptr System Architecture

Project Report

Course Instructor: Shahid Ismail
LE Usama Shaukat

Student Name:

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

Degree/ Syndicate: CE 45 A

Date: 2nd January, 2025

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

Table of Contents

Rida Zahra 465791 | Mawa Ch 474121 | Tazeen ur Rehman 466342

Project Objectives:

1. **Design a Pipelined Processor**: Create a pipelined processor in Verilog that demonstrates forwarding and stalling techniques to handle data hazards efficiently.
2. **Implement Forwarding and Stalling**: Implement forwarding to bypass data between pipeline stages and stalling when forwarding is not possible (e.g., load-use hazards).
3. **Handle Pipeline Hazards**: Ensure the processor can detect and resolve data and control hazards to maintain smooth instruction execution.
4. **Verilog Code and Testbench**: Develop the Verilog code for the processor and a testbench to validate its functionality.
5. **Report Documentation**: Provide a concise report detailing the design, Verilog implementation, and test results.

Project Description:

This project involves designing a pipelined processor in Verilog to demonstrate both **data forwarding** and **stalling**. The processor will use an ISA (RISC-V) and manage hazards like data dependencies and control hazards.

- **Forwarding**: Forward intermediate results between pipeline stages to reduce delays.
- **Stalling**: Introduce pipeline stalls when forwarding isn't sufficient, such as for load-use hazards.

The processor will be designed at the **Register Transfer Level (RTL)**, and a **testbench** will be used to verify the correct operation of both forwarding and stalling mechanisms.

Software Used: The Xilinx ISE (Integrated Synthesis Environment) is a comprehensive software suite that supports digital logic design, especially for FPGA-based lab tasks using Verilog. It enables students to write, simulate, synthesize, and implement Verilog code, streamlining the design workflow from initial code entry to real-time testing on FPGA hardware. ISE's tools for simulation and synthesis help verify and optimize designs before physical deployment, making it an essential tool for efficiently turning Verilog code into functioning digital circuits in the lab.

Background Information: The design and implementation of a *single-cycle 32-bit RISC-V processor* in a lab setting is essential for understanding computer architecture and digital design principles. This hands-on experience allows students to explore the RISC (Reduced Instruction Set Computer) architecture's efficiency, emphasizing a simplified instruction set that can execute in a single clock cycle. Through this practical work, students develop skills in hardware description languages (HDLs) and gain insights into data path and control unit design, timing analysis, and performance optimization. The flexibility and open nature of RISC-V also foster innovation and custom hardware development, making it highly relevant in fields such as embedded systems, IoT, and academic research.

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

Pipelined Processor:

A **pipelined processor** is a microprocessor architecture designed to improve instruction throughput by dividing the instruction execution process into multiple stages. Each stage performs a part of the instruction processing, and multiple instructions can be in different stages of execution at the same time. This parallel processing enhances the overall performance of the processor compared to a non-pipelined design, where instructions are processed sequentially.

In this project, we are using the **RISC-V** architecture for the pipelined processor. RISC-V is an open-source instruction set architecture (ISA) that follows a reduced instruction set computing (RISC) design philosophy, making it simple, efficient, and extensible. It provides a set of basic instructions that are easy to decode and execute, which is well-suited for pipelining.

A typical pipelined processor in RISC-V consists of several stages, including:

1. **Instruction Fetch (IF)**: The instruction is fetched from memory.
2. **Instruction Decode (ID)**: The instruction is decoded, and the operands (register values) are fetched.
3. **Execution (EX)**: The operation specified by the instruction is performed, or the address for memory access is calculated.
4. **Memory Access (MEM)**: Data is read from or written to memory if the instruction involves memory access.
5. **Write Back (WB)**: The result of the operation is written back to the register file.

In this pipelined design, while one instruction is being executed in one stage, other instructions can simultaneously be in different stages of the pipeline. This parallelism significantly improves the throughput of the processor.

Rida Zahra 465791 Mawa Ch 474121 Tazeen ur Rehman 466342

Complete Module:

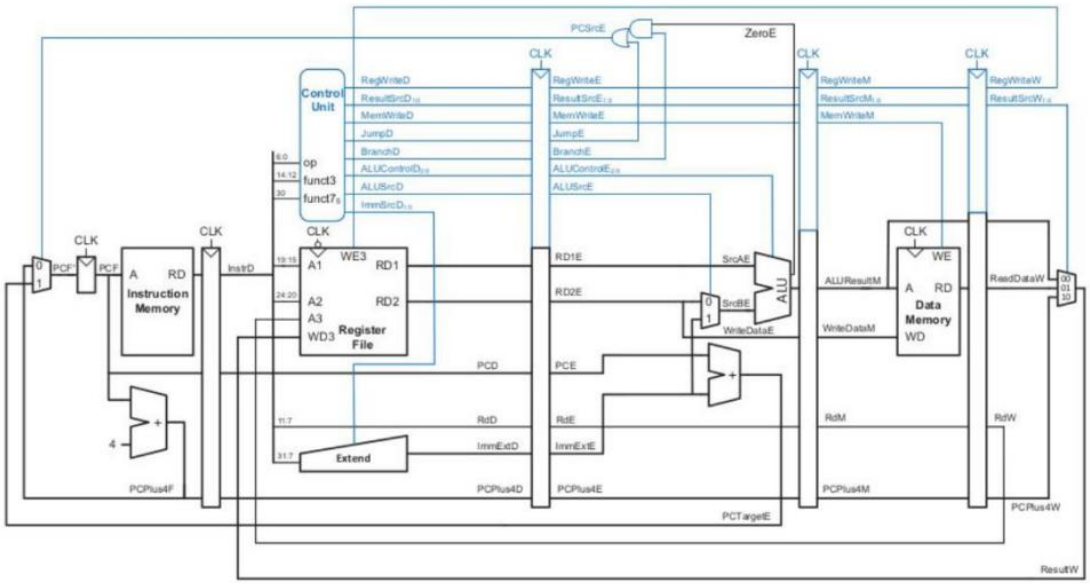


Fig: Complete Module

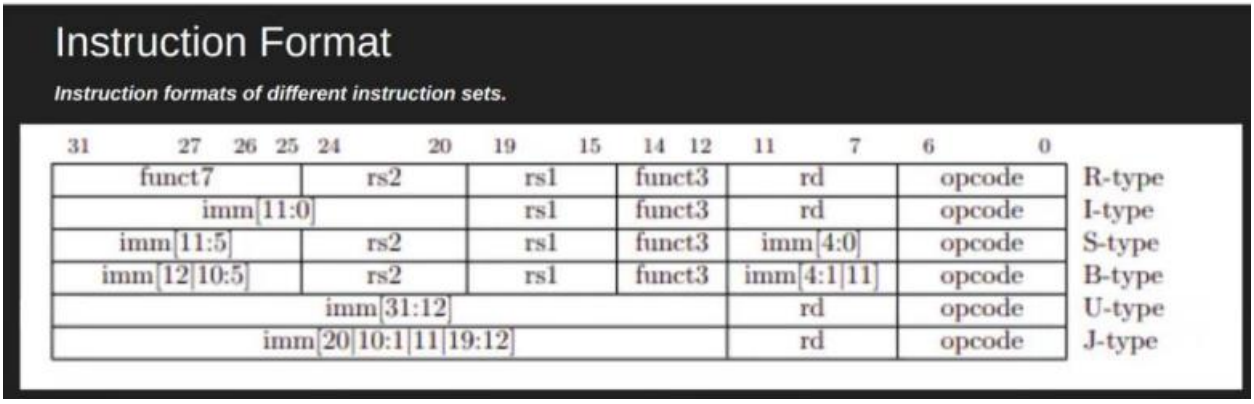
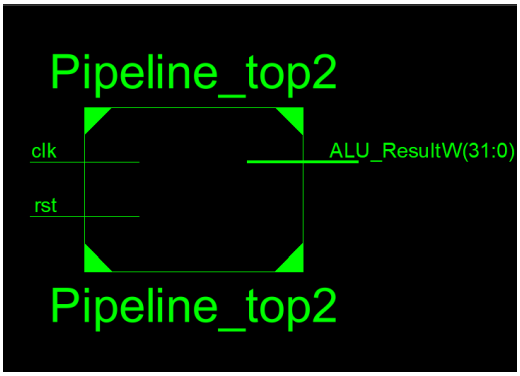


Fig: Instruction Format

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

RTL Diagram:



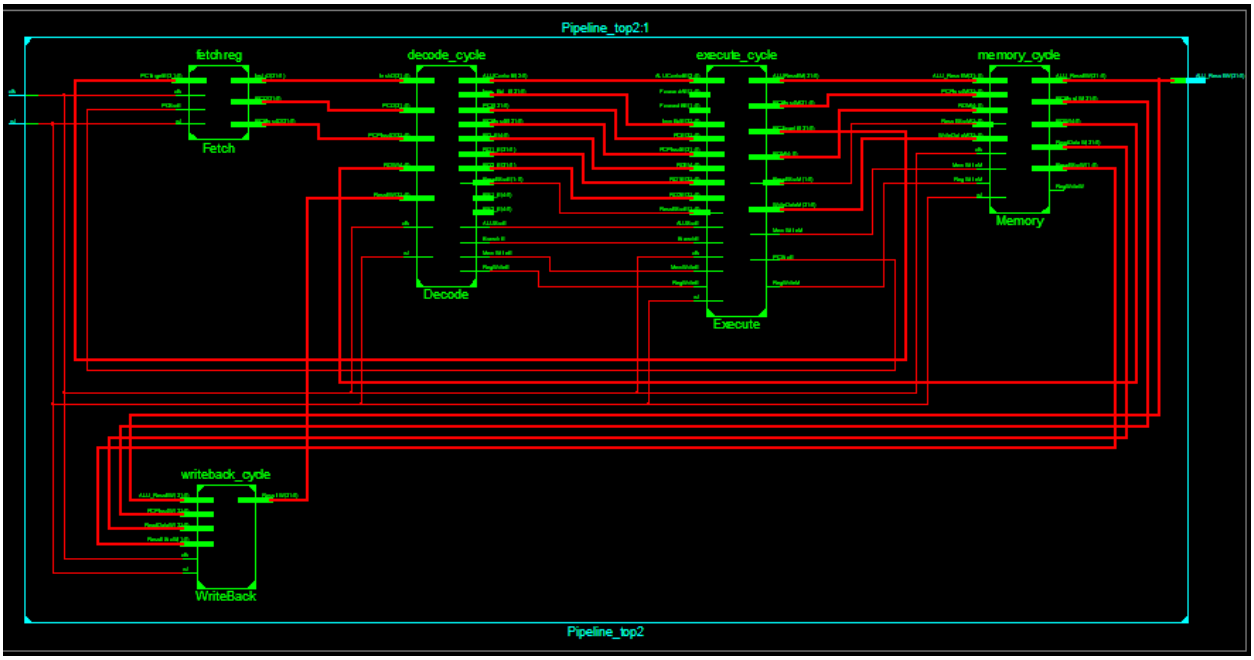


Fig: RTL Diagram

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

Simulation:

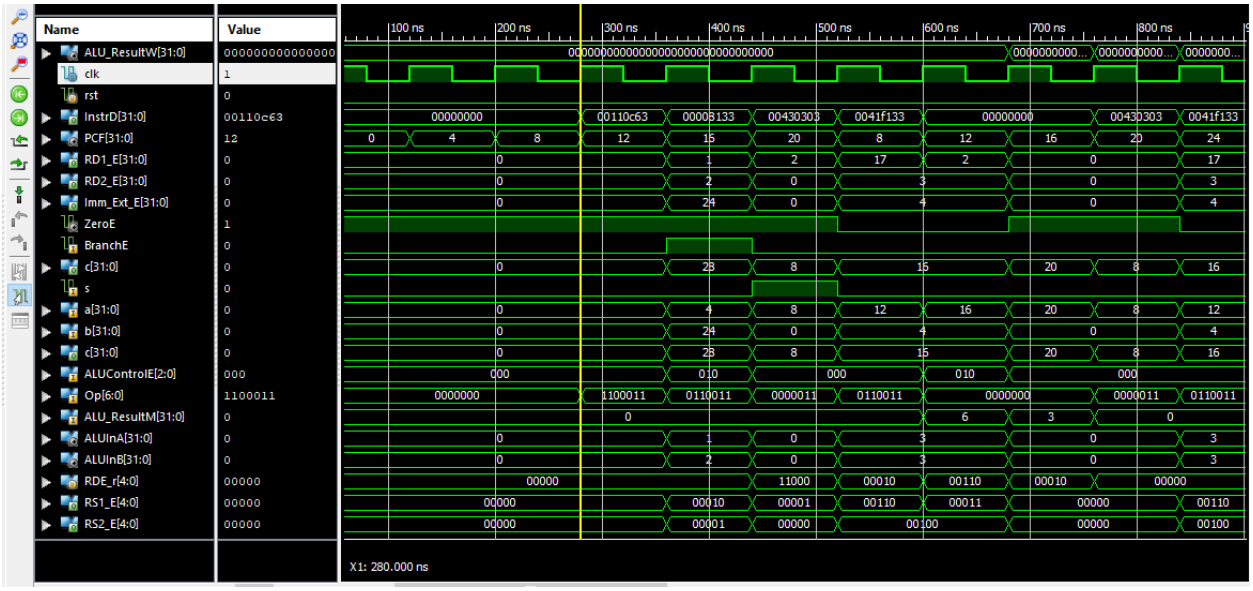


Fig: Simulation

```
  mem[0] = 32'h00110c63; // branch x1, x1, 4
  mem[1] = 32'h00008133; // add x2, x1, x0
  mem[2] = 32'h00430303; // lw x6, 4(x6)
  mem[3] = 32'h0041F133; // and x2, x3, x4
```

```
// Initialize register 0 to 0 at all times
    initial begin
    Register[0] = 32'h00000000;  // Register x0 is always 0
    Register[1] = 32'h00000002;  // Register x1 initialized to 0x00000002
    Register[2] = 32'h00000001;
    Register[3] = 32'h00000002;
    Register[4] = 32'h00000003;
    Register[5] = 32'h00000004;
    Register[6] = 32'h00000011;
    Register[7] = 32'h00000002;
    Register[8] = 32'h00000001;
    Register[9] = 32'h00000002;
    Register[10] = 32'h00000003;
    Register[11] = 32'h00000004;
    Register[12] = 32'h00000001;
    Register[13] = 32'h00000002;
```

Fig: Instruction Memory

Rida Zahra 465791 Mawa Ch 474121 Tazeen ur Rehman 466342

The implementation simulates the execution of a RISC-V pipelined processor performing a sequence of instructions (`add`, `lw`, `and`, and `branch`). The processor operates in a pipelined manner, where instruction fetch, decode, execute, memory access, and write-back stages overlap to improve performance. Hazard detection mechanisms are integrated to manage potential Read-After-Write (RAW) hazards and control hazards arising from branches. At this stage, no stalling is implemented, and hazards are being detected without applying stall signals. This approach provides a foundation for understanding pipeline behavior and hazard detection, with the next steps focusing on implementing mechanisms to resolve detected hazards for accurate execution.

Problem Faced with Simple Pipelining

The main problem addressed in this project involves managing data hazards in a pipelined processor architecture. Specifically, we are concerned with the load-use hazards that occur when an instruction depends on the result of a preceding load instruction. These hazards can cause pipeline stalls, reducing the overall performance of the processor. The challenge is to identify these hazards, such as those between the load (`lw`) instruction and subsequent instructions that use the loaded data, and implement solutions like stalling and data forwarding to mitigate the performance penalties and ensure the pipeline operates efficiently.

Overview of Changes in the Absence of Forwarding and Hazard Detection

Without **forwarding**, the processor cannot bypass intermediate results between pipeline stages. This means that whenever a data dependency exists (for example, when one instruction requires the result of a previous instruction), the processor must stall the pipeline until the required data is available. Without **hazard detection**, the processor lacks a mechanism to dynamically detect these hazards during runtime, forcing it to rely on **manual stall insertion** or a simpler control flow for handling hazards.

Key Differences in the Pipeline Stages

1. **Fetch Stage**:
    - The **Fetch Stage** in this design remains largely unchanged, as it is responsible for fetching instructions from memory. Without forwarding and hazard detection, the Fetch stage simply continues to fetch the next instruction unless a stall condition is met.
    - The **StallD** signal would be generated based on external logic to introduce stalls if the pipeline detects dependencies, although without dynamic hazard detection, the control over stalling is less efficient.
    - Since we don't have forwarding or hazard detection, we assume that each instruction must wait until the necessary data has been written back from the **Write Back Stage** before continuing to the next stage.
2. **Decode Stage**:
    - In the **Decode Stage**, without hazard detection, the processor does not proactively check for data hazards. Instead, if a load instruction is in the pipeline, the subsequent instruction that depends on that data will simply be stalled until the load instruction writes its result back.

- Stalling is achieved by introducing **nop (no-operation)** instructions or delaying the pipeline until the result from the previous instruction is available.
- The Decode stage does not have the ability to detect if an instruction is about to use data that is not yet available, so the control logic in this stage would be more static and less efficient.

3. **Execute Stage**:
   - The **Execute Stage** would also be less efficient without forwarding, as it would need to wait for the previous stage to complete before performing operations that require the data.
   - The **ALU operations** in this stage would be delayed if they depend on data that is still in the process of being written back to registers.
4. **Memory and Write Back Stages**:
   - The **Memory Stage** and **Write Back Stage** would operate similarly, but without forwarding, results that should be passed to the next stage must go through the normal register write-back process.
   - The data dependency between stages will be handled by introducing manual stalls, preventing the pipeline from executing too quickly or incorrectly.

Impact on Performance and Efficiency

- **Stalling Mechanism**: Without forwarding, the processor must use **stall cycles** to avoid reading incorrect data. This introduces **pipeline hazards** that are not handled dynamically, which increases the overall latency and reduces the throughput of the processor.
- **Pipeline Inefficiency**: The absence of hazard detection means that the pipeline will not be optimized for avoiding hazards during execution. As a result, the processor will be less efficient, as it cannot skip unnecessary cycles where data is available in later pipeline stages but would have to wait for the data to be written back.
- **Pipeline Throughput**: Without forwarding, and with the reliance on manual stall insertion, the throughput of the processor decreases. Instructions that could have been executed in parallel (if forwarding were present) will instead have to wait for earlier instructions to complete, leading to fewer instructions being completed in a given period.

In summary, implementing a pipelined processor without forwarding and hazard detection significantly impacts performance. Without forwarding, instructions have to wait for results from previous stages, leading to increased delays. Without hazard detection, the processor cannot dynamically adapt to hazards, leading to manual stalling, which introduces inefficiencies in instruction execution. While this approach simplifies the design, it results in a processor that is less efficient and slower than one with proper forwarding and hazard detection mechanisms.

Rida Zahra 465791
Mawa Ch 474121
Tazeen ur Rehman 466342

Forwarding and Data Hazard Unit

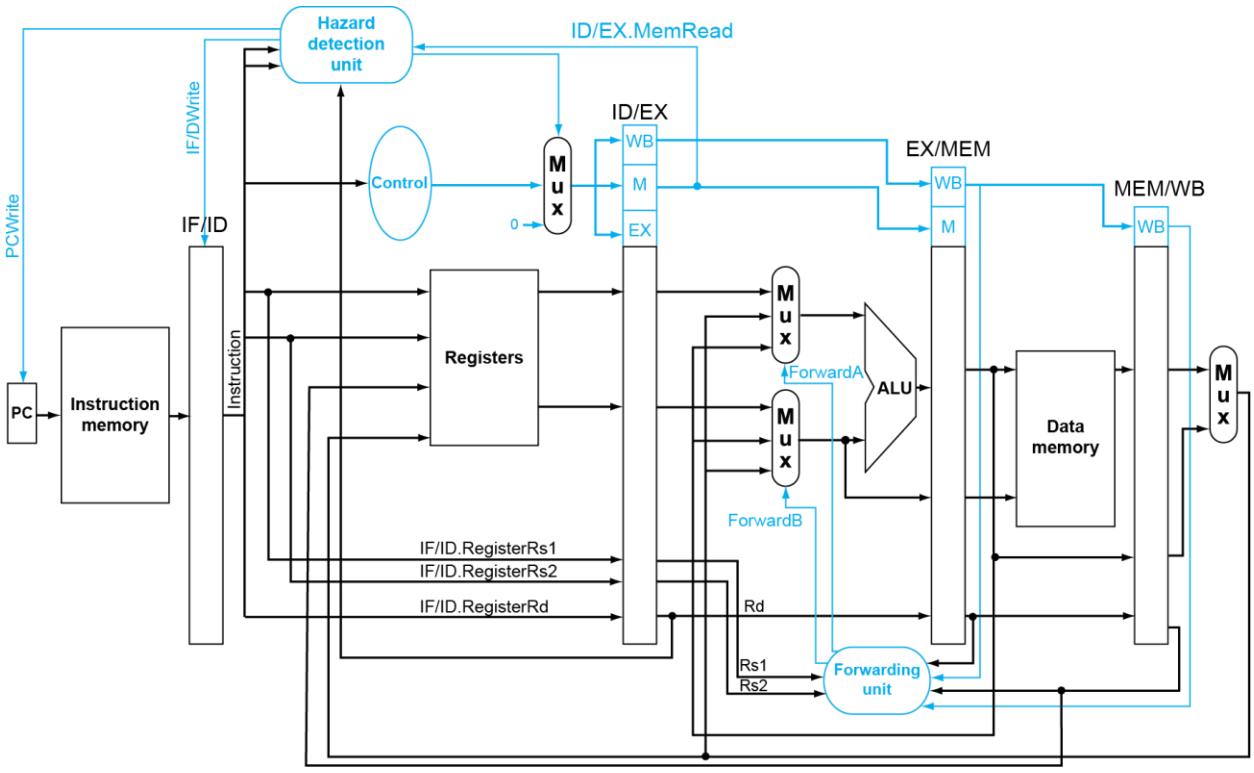
1. Data Hazard Unit:

The **Data Hazard Unit** is responsible for detecting data hazards, particularly **read-after-write** hazards (also called **RAW hazards**) where an instruction depends on the result of a previous instruction that is yet to complete. The Data Hazard Unit will monitor the pipeline and identify if there is a dependency between the instructions in the pipeline.

2. Forwarding Unit:

The **Forwarding Unit** is responsible for bypassing data from one pipeline stage to another without having to wait for the data to pass through all the stages (e.g., writing it back to the register file in the **WB** stage). This is especially important in pipelined processors to avoid stalls, particularly when data is available in the **MEM** or **EX** stage but is needed earlier.

By combining the **Data Hazard Unit** for stall detection and the **Forwarding Unit** for data forwarding, you can handle data hazards efficiently and improve the throughput of your pipelined processor.



Rida Zahra 465791 | Mawa Ch 474121 | Tazeen ur Rehman 466342

Code of Forwarding & Hazard Unit:

Forwarding Unit

```
module forwarding_unit (
   input [4:0] RS1E, RS2E, // Source registers in Execute stage
   input [4:0] RDM, RDW,  // Destination registers in Memory and Write-back stages
   input RegWriteM, RegWriteW, // Register write enable signals
   output reg [1:0] ForwardAE, ForwardBE // Forwarding controls
);
   always @(*) begin
      // ForwardAE logic
      if (RegWriteM && (RDM != 0) && (RDM == RS1E))
         ForwardAE = 2'b10; // Forward from Memory stage
      else if (RegWriteW && (RDW != 0) && (RDW == RS1E))
         ForwardAE = 2'b01; // Forward from Write-back stage
      else
         ForwardAE = 2'b00; // No forwarding

      // ForwardBE logic
      if (RegWriteM && (RDM != 0) && (RDM == RS2E))
         ForwardBE = 2'b10; // Forward from Memory stage
      else if (RegWriteW && (RDW != 0) && (RDW == RS2E))
         ForwardBE = 2'b01; // Forward from Write-back stage
      else
         ForwardBE = 2'b00; // No forwarding
   end
endmodule
```

Hazard Unit:

```
module hazard_detection_unit (
   input clk, rst,
   input [6:0] op,
   input [4:0] RS1_ID, RS2_ID, RD_EX, RD_MEM, RD_WB, // Source and destination registers
   input RegWrite_EX, RegWrite_MEM, RegWrite_WB,    // Write enable signals
   input MemRead_EX, // Flag indicating if EX stage is a memory read (prevents forwarding)
   output reg StallD // Stall signal for IF and ID stages
);

   reg stall_next; // Temporary signal to hold the stall value for the next cycle

   // Hazard detection logic
   always @(*) begin
      // Default values (no hazards detected)
```

```
    stall_next = 0;

                // EX stage hazard detection (RAW)
      if (RegWrite_EX && (RS1_ID != 5'b0) && (RS1_ID == RD_EX)) begin
        stall_next = 1; // Stall the decode stage if RS1 is the destination of the EX stage
      end
      if (RegWrite_EX && (RS2_ID != 5'b0) && (RS2_ID == RD_EX)) begin
        stall_next = 1; // Stall the decode stage if RS2 is the destination of the EX stage
      end

      // MEM stage hazard detection (RAW)
      if (RegWrite_MEM && (RS1_ID != 5'b0) && (RS1_ID == RD_MEM)) begin
        stall_next = 1; // Stall the decode stage if RS1 is the destination of the MEM stage
      end
      if (RegWrite_MEM && (RS2_ID != 5'b0) && (RS2_ID == RD_MEM)) begin
        stall_next = 1; // Stall the decode stage if RS2 is the destination of the MEM stage
      end

      // WB stage hazard detection (RAW)
      if (RegWrite_WB && (RS1_ID != 5'b0) && (RS1_ID == RD_WB)) begin
        stall_next = 1; // Stall the decode stage if RS1 is the destination of the WB stage
      end
      if (RegWrite_WB && (RS2_ID != 5'b0) && (RS2_ID == RD_WB)) begin
        stall_next = 1; // Stall the decode stage if RS2 is the destination of the WB stage
      end

      // Hazard detection for RAW (memory read and write conflicts)
      if (MemRead_EX && ((RS1_ID == RD_EX) || (RS2_ID == RD_EX))) begin
        stall_next = 1; // Stall the decode stage
      end

    if(MemRead_EX) begin
                stall_next = 1;
                end

    // Branch instruction stall
    if (op == 7'b1100011) begin
      stall_next = 1;
    end
  end

  // Update StallD with clock
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      StallD <= 0; // Reset stall signal
    end else begin
```

```
        if (StallD == 1) begin
            StallD <= 0; // Clear stall after one cycle
        end else begin
            StallD <= stall_next; // Update stall based on hazard detection
        end
    end
end

endmodule
```

RTL Diagram

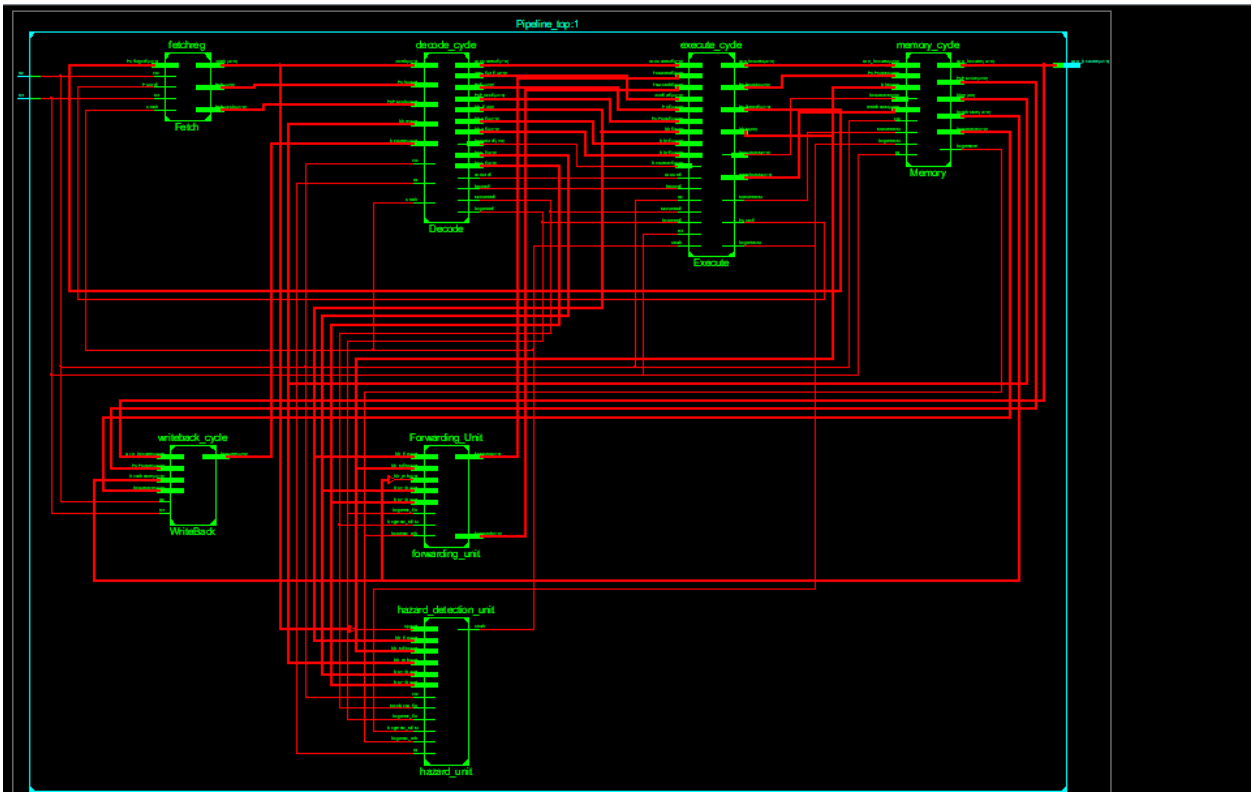


Fig: RTL Diagram

Top Module Combined Code:

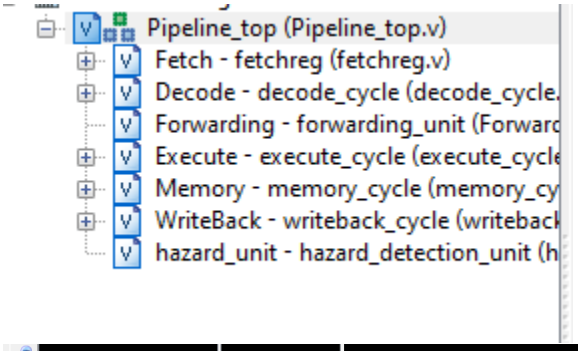
```
module Pipeline_top(
   input clk,
   input rst,
   output [31:0] ALU_ResultW
);

  // Declaration of Wires
  wire PCSrcE;
  wire RegWriteE, ALUSrcE, MemWriteE,  BranchE;
  wire RegWriteM, MemWriteM, RegWriteW;
  wire ResultSrcM, ResultSrcW, ResultSrcE;
  wire [2:0] ALUControlE;
  wire [4:0] RDE, RDM, RD_W, RS1_E, RS2_E;
  wire [31:0] PCTargetE, InstrD, PCD, PCPlus4D;
  wire [31:0] ResultW, RD1_E, RD2_E, Imm_Ext_E, PCE, PCPlus4E;
  wire [31:0] PCPlus4M, WriteDataM, ALU_ResultM;
  wire [31:0] PCPlus4W, ReadDataW;
  wire [1:0] ForwardAE, ForwardBE;

  // Hazard Detection Unit Signals
  wire StallD;

  // Fetch Stage
  fetchreg Fetch (
    .clk(clk),
    .rst(rst),
    .PCSrcE(PCSrcE),
    .PCTargetE(PCTargetE),
    .InstrD(InstrD),
    .PCD(PCD),
    .PCPlus4D(PCPlus4D),
          .StallD(StallD)
  );

  // Decode Stage
  decode_cycle Decode (
    .clk(clk),
    .rst(rst),
    .InstrD(InstrD),
    .PCD(PCD),
    .PCPlus4D(PCPlus4D),
    .RDW(RD_W),
    .ResultW(ResultW),
    .RegWriteE(RegWriteE),
```

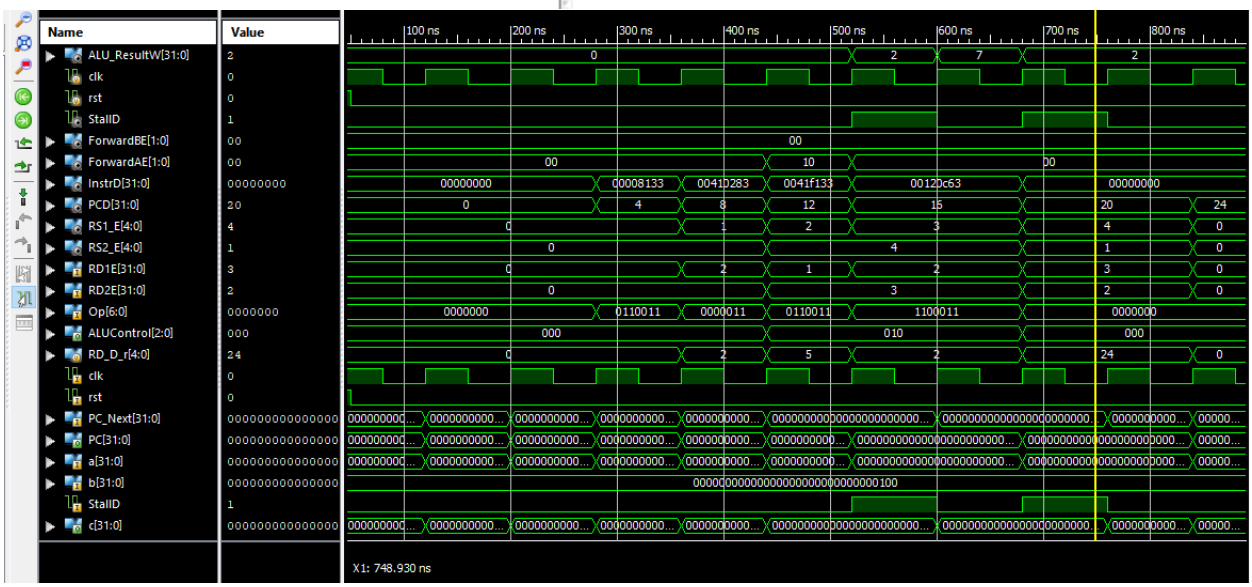
```
  .ALUSrcE(ALUSrcE),
  .MemWriteE(MemWriteE),
  .ResultSrcE(ResultSrcE),
  .BranchE(BranchE),
  .ALUControlE(ALUControlE),
  .RD1_E(RD1_E),
  .RD2_E(RD2_E),
  .Imm_Ext_E(Imm_Ext_E),
  .RD_E(RDE),
  .PCE(PCE),
  .PCPlus4E(PCPlus4E),
  .RS1_E(RS1_E),
  .RS2_E(RS2_E),
  .StallD(StallD)  // Stall signal from hazard detection unit
 );

  // Forwarding Unit
  forwarding_unit Forwarding (
     .RS1E(RS1_E),          // Connected RS1_E from decode stage
     .RS2E(RS2_E),          // Connected RS2_E from decode stage
     .RDM(RDM),
     .RDW(RD_W),
     .RegWriteM(RegWriteM),
     .RegWriteW(RegWriteW),
     .ForwardAE(ForwardAE),
     .ForwardBE(ForwardBE)
  );

          // Execute Stage
 execute_cycle Execute (
  .clk(clk),
  .rst(rst),
  .ALUSrcE(ALUSrcE),
  .RegWriteE(RegWriteE),
  .MemWriteE(MemWriteE),
  .ResultSrcE(ResultSrcE),
  .BranchE(BranchE),
  .ALUControlE(ALUControlE),
  .RD1E(RD1_E),
  .RD2E(RD2_E),
  .ImmExtE(Imm_Ext_E),
  .RDE(RDE),
  .PCE(PCE),
  .PCPlus4E(PCPlus4E),
  .PCSrcE(PCSrcE),
  .PCTargetE(PCTargetE),
```

```
  .RegWriteM(RegWriteM),
  .MemWriteM(MemWriteM),
  .PCPlus4M(PCPlus4M),
  .WriteDataM(WriteDataM),
  .ALUResultM(ALU_ResultM),
  .ResultSrcM(ResultSrcM),
  .RDM(RDM),
  .ForwardAE(ForwardAE),
  .ForwardBE(ForwardBE),
        .StallD(StallD)
);

// Memory Stage
memory_cycle Memory (
  .clk(clk),
  .rst(rst),
  .RegWriteM(RegWriteM),
  .MemWriteM(MemWriteM),
  .ResultSrcM(ResultSrcM),
  .RDM(RDM),
  .PCPlus4M(PCPlus4M),
  .WriteDataM(WriteDataM),
  .ALU_ResultM(ALU_ResultM),
  .RegWriteW(RegWriteW),
  .ResultSrcW(ResultSrcW),
  .RDW(RD_W),
  .PCPlus4W(PCPlus4W),
  .ALU_ResultW(ALU_ResultW),
  .ReadDataW(ReadDataW)
);

// Write Back Stage
writeback_cycle WriteBack (
  .clk(clk),
  .rst(rst),
  .ResultSrcW(ResultSrcW),
  .PCPlus4W(PCPlus4W),
  .ALU_ResultW(ALU_ResultW),
  .ReadDataW(ReadDataW),
  .ResultW(ResultW)
);

// Hazard Detection Unit
hazard_detection_unit hazard_unit (
  .clk(clk),
  .rst(rst),
```

```
        .op(InstrD[6:0]),
   .RS1_ID(RS1_E),
   .RS2_ID(RS2_E),
   .RD_EX(RDE),
   .RD_MEM(RDM),
   .RD_WB(RD_W),
   .RegWrite_EX(RegWriteE),
   .RegWrite_MEM(RegWriteM),
   .RegWrite_WB(RegWriteW),
   .MemRead_EX(ResultSrcE),
   .StallD(StallD)
  );

endmodule
```

Simulation:





Rida Zahra 465791 Mawa Ch 474121 Tazeen ur Rehman 466342

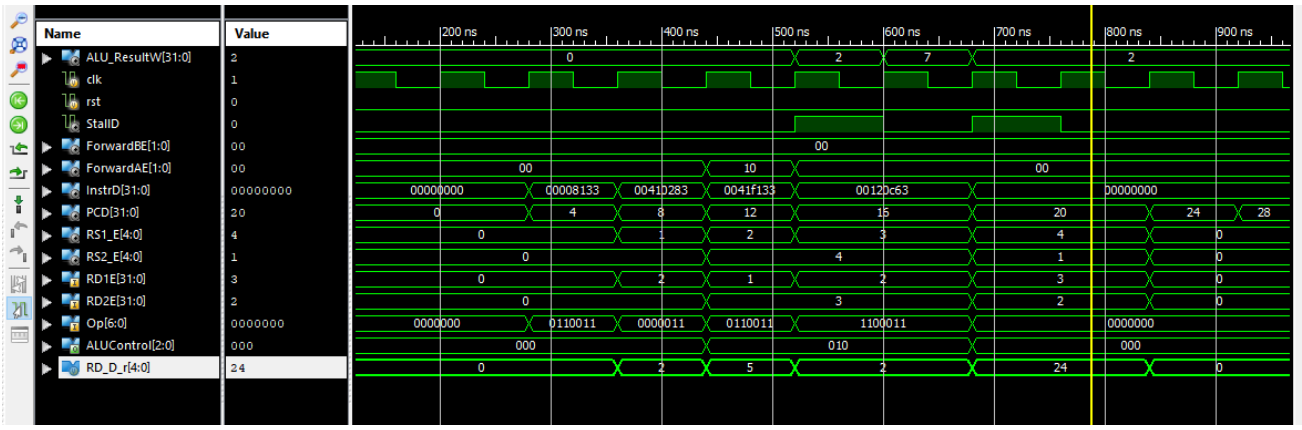


Fig: Simulations

4 instruction are in the instruction memory and are tested for their complete cycles

```
// Load instructions into specific memory locations
mem[0] = 32'h00008133; // add x2, x1, x0
mem[1] = 32'h00410283; // lw x6, 4(x2)
mem[2] = 32'h0041F133; // and x2, x3, x4
mem[3] = 32'h00120c63; // branch x1, x2, 4
```

Instructions

```
initial begin
Register[0] = 32'h00000000;  // Register x0 is always 0
Register[1] = 32'h00000002;  // Register x1 initialized to 0x00000002
Register[2] = 32'h00000001;
Register[3] = 32'h00000002;
Register[4] = 32'h00000003;
Register[5] = 32'h00000004;
Register[6] = 32'h00000011;
Register[7] = 32'h00000002;
Register[8] = 32'h00000001;
Register[9] = 32'h00000002;
Register[10] = 32'h00000003;
Register[11] = 32'h00000004;
Register[12] = 32'h00000001;
Register[13] = 32'h00000002;
```

Register File

Rida Zahra 465791 Mawa Ch 474121 Tazeen ur Rehman 466342

Stalling & Forwarding Unit:

Fig: Simulation of Stalling and Forwarding Unit

Complete Stalling & Forwarding signals implemented and checked in the output

How we Implemented it?

Forwarding Unit Implementation

The forwarding unit is responsible for resolving data hazards where an instruction depends on the result of a previous instruction that has not yet completed the entire pipeline. Instead of stalling the pipeline, forwarding allows the processor to "forward" the intermediate results from one pipeline stage to another to ensure the dependent instruction can proceed without delay.

Here's how forwarding is implemented in the code:

1. **Inputs to the Forwarding Unit:**
   - **RS1_E, RS2_E:** These are the source registers for the current instruction in the Execute stage. They represent the registers that are used by the instruction currently in the Execute stage.
   - **RDM, RD_W:** These are the destination registers from the Memory and Writeback stages respectively. They represent the registers that hold the results of the memory and writeback operations from previous instructions.
   - **RegWriteM, RegWriteW:** These signals indicate whether the destination register in the Memory or Writeback stage is being written to.
2. **Forwarding Decision:**
   - The forwarding unit checks if there is a hazard for the source registers **RS1_E** and **RS2_E**. It compares these with the destination registers **RDM** (from the Memory stage) and **RD_W** (from the Writeback stage).
   - If a match is found, and the corresponding **RegWriteM** or **RegWriteW** signal is active, forwarding occurs.
   - **ForwardAE** and **ForwardBE** are the signals that determine where the values should be forwarded from:
     - **ForwardAE**: Determines if data for **RS1_E** should be forwarded from the Memory or Writeback stage.
     - **ForwardBE**: Determines if data for **RS2_E** should be forwarded from the Memory or Writeback stage.

Rida Zahra 465791 Mawa Ch 474121 Tazeen ur Rehman 466342

Hazard Detection Unit Implementation

The hazard detection unit is designed to identify and handle hazards that can arise in the pipeline, particularly **data hazards** and **control hazards**. A hazard occurs when an instruction depends on data that is not yet available because a previous instruction is still in progress.

The primary function of the hazard detection unit in this code is to detect situations where the data needed for the current instruction is not yet available, requiring the pipeline to stall.

1. **Inputs to the Hazard Detection Unit:**
    - **InstrD[6:0]:** The opcode of the instruction in the Decode stage. This helps determine if a control hazard (like a branch) is present.
    - **RS1_ID, RS2_ID:** The source registers for the current instruction in the Decode stage.
    - **RD_EX, RD_MEM, RD_WB:** The destination registers from the Execute, Memory, and Writeback stages.
    - **RegWrite_EX, RegWrite_MEM, RegWrite_WB:** These signals indicate whether the destination register in the respective pipeline stages is being written to.
    - **MemRead_EX (ResultSrcE):** This signal indicates whether the instruction in the Execute stage is performing a memory read operation.
2. **Hazard Detection Logic:**
    - **Data Hazards:** The hazard unit detects **load-use hazards** when a load instruction in the Execute stage tries to use data that is not yet written back from the Memory stage. In such cases, the pipeline needs to stall.
        - If the current instruction in the Decode stage is dependent on a register that will be written by an instruction in the Execute stage (and that instruction has not written its result yet), the pipeline will be stalled. The **StallD** signal is asserted to halt the pipeline until the data is available.
    - **Control Hazards:** The hazard detection unit also checks for control hazards such as branches (using the opcode) and ensures the pipeline handles them appropriately. The **PCSrcE** and **PCTargetE** signals are used to handle control hazards related to branches and jumps.
3. **Stalling Mechanism:**
    - When the hazard unit detects that data is not available or a control hazard is present, it will assert the **StallD** signal, which is sent to the **Fetch** stage. This halts the instruction fetch and decode stages until the data hazard or control hazard is resolved.

Flow of Forwarding and Hazard Detection

- During the Decode stage, the hazard detection unit checks if any data hazards are present and may assert **StallD** to prevent the pipeline from progressing until the hazard is resolved.
- The forwarding unit then determines if any data should be forwarded from the Memory or Writeback stages to the Execute stage, bypassing any potential delays caused by waiting for data to be written back to registers.

- If forwarding is not possible, the processor stalls by holding the Fetch and Decode stages, ensuring the instruction flow continues correctly once the hazard is cleared.

Conclusion

In this project, we successfully designed and implemented a pipelined processor in Verilog that addresses critical performance issues such as data forwarding and pipeline stalling. The processor was able to efficiently handle data hazards through forwarding, reducing unnecessary delays by bypassing intermediate results between pipeline stages. In cases where forwarding was not possible, especially for load-use hazards, pipeline stalling mechanisms were introduced to ensure correct execution of instructions.

The Verilog code was developed at the Register Transfer Level (RTL), allowing for precise control over the processor's operations. A comprehensive testbench was also created to validate the functionality of both the forwarding and stalling techniques. Through rigorous testing, the processor demonstrated its ability to resolve data and control hazards, ensuring smooth and efficient execution of instructions.

Overall, this project not only enhanced our understanding of pipelined processor design but also provided practical insights into optimizing processor performance through effective hazard handling techniques. The design and implementation serve as a foundation for further exploration into more advanced pipelining techniques and processor optimizations.