DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF E&ME, NUST, RAWALPINDI

Computer System Architecture

Pipelined Processor

Semester Project Report

SUBMITTED TO:

Asst Prof Shahid Ismail / Engr. Usama Shoukat

SUBMITTED BY:

Aleeza Rizwan	**Ibrahim Abdullah**	**Aina Ajmal**	**Aameena Qadeer**
Reg# 456143	**Reg# 454188**	**Reg# 465903**	**Reg# 464167**

CE 45 A

Submission Date: 1st January, 2025.

Objective

The objective of this project is to design and implement a pipelined processor capable of efficiently managing instruction hazards through stalling and data forwarding techniques. The processor will be designed to demonstrate three distinct operational modes: stall-only design, forwarding-based design, and a combined stall + forwarding design. The implementation will use an instruction set architecture (ISA) such as MIPS, RISC-V, or ARM, selected for its suitability in demonstrating these concepts. The design will be verified through simulation to ensure correct functionality and performance under various scenarios. Additionally, the project aims to provide a detailed hardware structure (RTL), a fully functional Verilog implementation with a testbench, and an example instruction set to showcase the processor's capabilities. This work will contribute to a deeper understanding of pipelined processor design, hazard mitigation, and efficient instruction execution.

Introduction

Pipelining is a widely used technique in computer architecture aimed at improving instruction throughput by overlapping the execution of multiple instructions. However, instruction dependencies often introduce hazards that can disrupt the smooth flow of the pipeline. These hazards are broadly classified into data hazards**,** control hazards**,** and structural hazards. To address these challenges, techniques such as stalling and data forwarding (bypassing**)** are employed. Stalling pauses the pipeline until dependencies are resolved, while forwarding allows intermediate results to be directly passed to dependent instructions, reducing unnecessary delays. This project focuses on designing a pipelined processor capable of demonstrating three distinct operational modes: stall-only design**,** forwarding-based design**,** and stall + forwarding-based design.

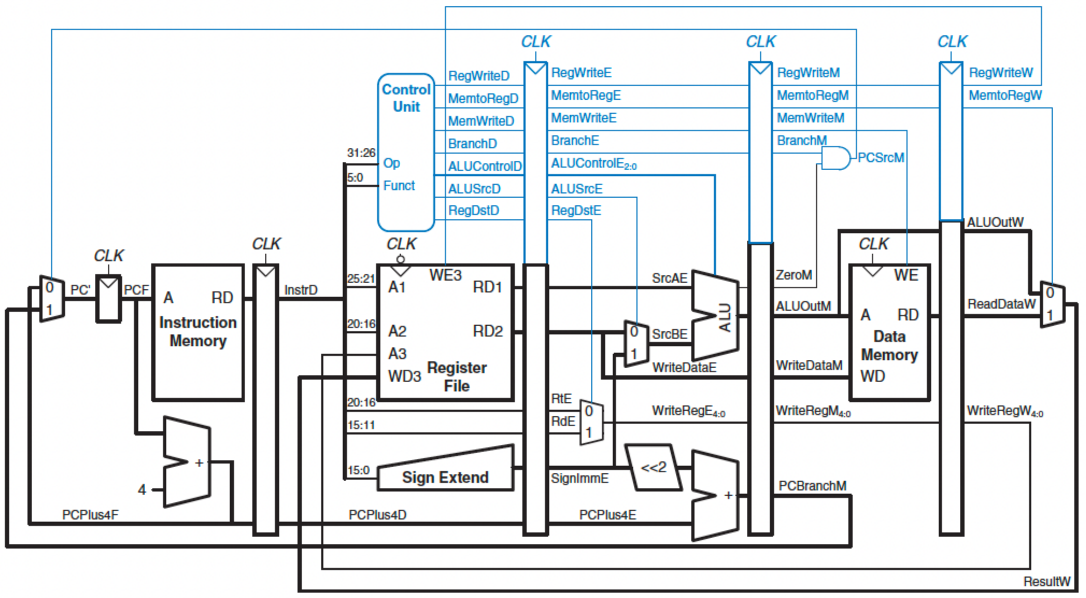
Features

- **Pipeline Stages:** The instruction execution process will be broken down into stages: instruction fetch, decode, execute, memory access (if needed), and write-back of results.
- **Instruction Set Support:** The processor will support a specific set of commonly used instructions, including basic operations like addition, data movement, and conditional decisions.
- **Pipeline Hazards Handling:** Techniques will be implemented to handle pipeline hazards, such as instruction conflicts or branches that alter the execution flow, to ensure smooth operation.
- **Branch Prediction:** Conditional branch outcomes will be predicted to keep the pipeline running smoothly, reducing delays by processing instructions ahead of time.
- **Data Forwarding:** Shortcuts will be established to pass data directly between pipeline stages, reducing delays associated with traditional data movement through the pipeline.
- **Memory Access:** The processor will efficiently read from and write to the memory subsystem as needed during execution.
- **Simulation and Testing:** Extensive testing will be conducted to ensure the processor works correctly under various scenarios and to identify potential issues before advancing.
- **Performance Evaluation:** The processor's performance will be evaluated by measuring instruction throughput, task completion speed, and resource efficiency.

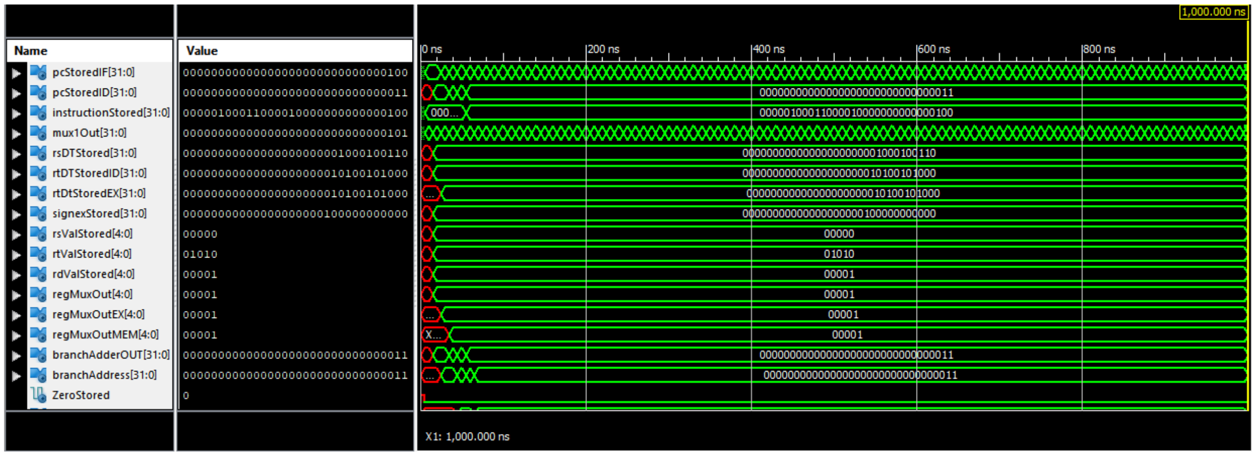
- **Synthesis and Optimization:** After confirming the design works, optimization will be performed to maximize efficiency and speed, while meeting all design requirements.
- **Documentation and Reporting:** Detailed records will be kept throughout the project to track design decisions, progress, and insights gained.

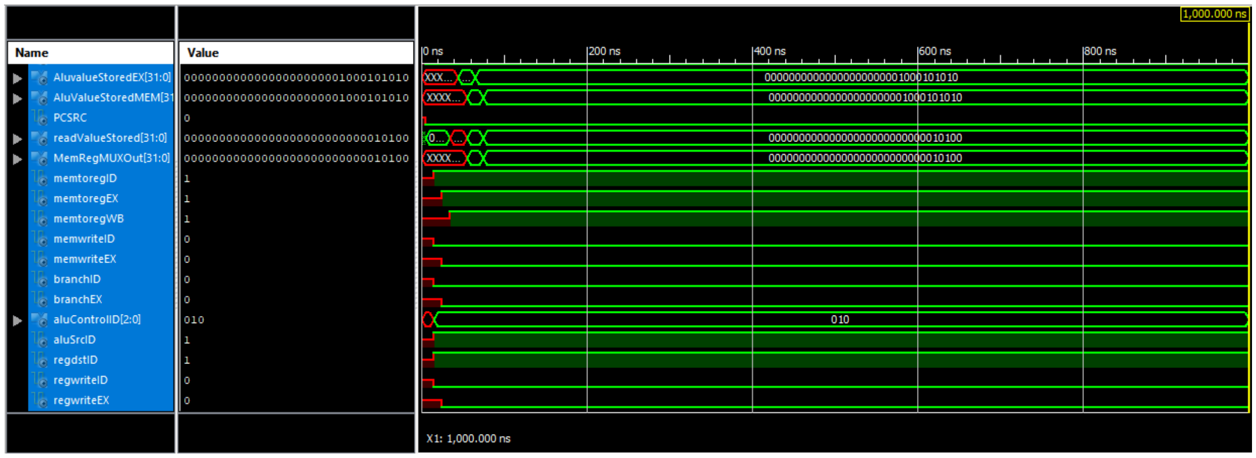
Diagram

A 32-bit pipelined processor is a microprocessor designed with 32-bit data path width and pipeline architecture to optimize performance. This design enables the processor to handle 32 bits of data simultaneously while dividing the instruction execution process into multiple overlapping stages. By allowing instructions to be executed in parallel across these pipeline stages, the processor achieves higher throughput and significantly improved operational efficiency.

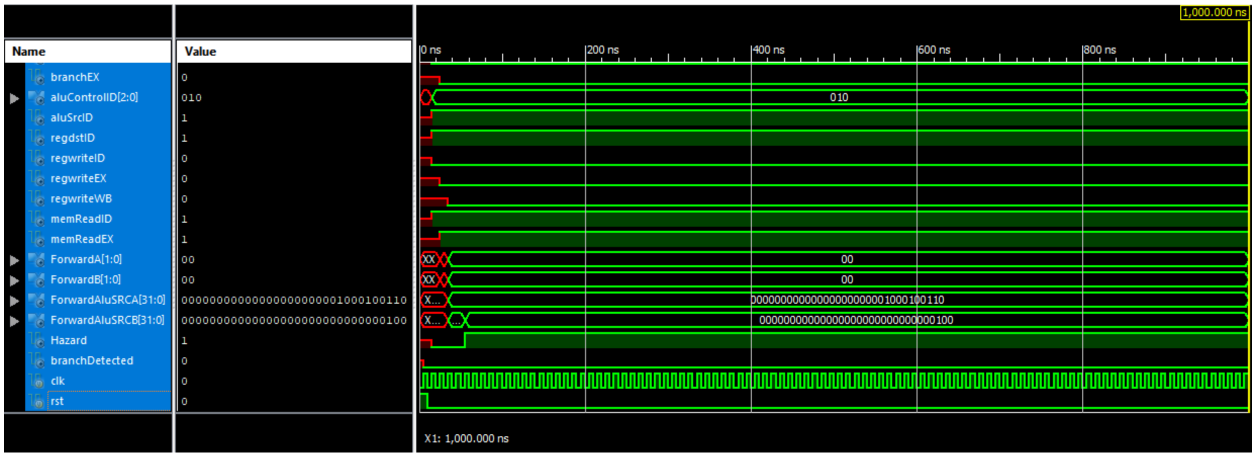


Simulation


```
Name                     Value
pcStoredIF[31:0]         00000000000000000000000000000100
pcStoredID[31:0]         00000000000000000000000000000011
instructionStored[31:0]  00000100011000010000000000000100
mux1Out[31:0]            00000000000000000000000000000101
rsDTStored[31:0]         00000000000000000000001000100110
rtDTStoredID[31:0]       00000000000000000000010100101000
rtDtStoredEX[31:0]       00000000000000000000010100101000
signexStored[31:0]       00000000000000000000100000000000
rsValStored[4:0]         00000
rtValStored[4:0]         01010
rdValStored[4:0]         00001
regMuxOut[4:0]           00001
regMuxOutEX[4:0]         00001
regMuxOutMEM[4:0]        00001
branchAdderOUT[31:0]     00000000000000000000000000000011
branchAddress[31:0]      00000000000000000000000000000011
ZeroStored               0
```



Name	Value
AluvalueStoredEX[31:0]	00000000000000000000001000101010
AluValueStoredMEM[31	00000000000000000000001000101010
PCSRC	0
readValueStored[31:0]	00000000000000000000000000010100
MemRegMUXOut[31:0]	00000000000000000000000000010100
memtoregID	1
memtoregEX	1
memtoregWB	1
memwriteID	0
memwriteEX	0
branchID	0
branchEX	0
aluControlID[2:0]	010
aluSrcID	1
regdstID	1
regwriteID	0
regwriteEX	0



Name	Value
branchEX	0
aluControlID[2:0]	010
aluSrcID	1
regdstID	1
regwriteID	0
regwriteEX	0
regwriteWB	0
memReadID	1
memReadEX	1
ForwardA[1:0]	00
ForwardB[1:0]	00
ForwardAluSRCA[31:0]	00000000000000000000001000100110
ForwardAluSRCB[31:0]	00000000000000000000000000000100
Hazard	1
branchDetected	0
clk	0
rst	0

Verilog Code

Top Module

```
module PipelineTop(
		input clk,
		input rst,

		//pipelining
		output [31:0] pcStoredIF,
		output [31:0] pcStoredID,
		output [31:0] instructionStored,
		output [31:0] mux1Out,
		output [31:0] rsDTStored,
		output [31:0] rtDTStoredID,
		output [31:0] rtDtStoredEX,
		output [31:0] signexStored,
		output [4:0] rsValStored,
		output [4:0] rtValStored,
		output [4:0] rdValStored,
		output [4:0] regMuxOut,
		output [4:0] regMuxOutEX,
		output [4:0] regMuxOutMEM,
		output [31:0] branchAdderOUT,
		output [31:0] branchAddress,
		output ZeroStored,
		output [31:0] AluvalueStoredEX,
		output [31:0] AluValueStoredMEM,
		output PCSRC,
		output [31:0] readValueStored,
		output [31:0] MemRegMUXOut,

		//Control Outputs
		output  memtoregID,
		output  memtoregEX,
		output  memtoregWB,
		output memwriteID,
		output memwriteEX,
		output branchID,
		output branchEX,
		output [2:0] aluControlID,
		output aluSrcID,
		output regdstID,
		output regwriteID,
		output regwriteEX,
		output regwriteWB,
		output memReadID,
		output memReadEX,

		//Forwarding
		output [1:0] ForwardA,
		output [1:0] ForwardB,
		output [31:0] ForwardAluSRCA,
		output [31:0] ForwardAluSRCB,

		//Hazard
		output Hazard,

		//Branch Hazard
		output branchDetected

	);
		//Wires
		//ALU
		wire [31:0] alu_result;
		wire alu_zero;
```

```
            //Reg file
            wire [31:0] reg_rsData;
            wire [31:0] reg_rtData;
            //Inst mem
            wire [31:0] inst_instruct;
            //Data mem
            wire [31:0] data_rd;
            //PC
            wire [31:0] pc_pcout;
            //Decoder
            wire [4:0] int_shamt;
            wire [5:0] int_func;
            wire [15:0] int_immed;
            wire [25:0] int_target;
            wire [4:0] int_rs;
            wire [4:0] int_rt;
            wire [4:0] int_rd;
            wire [5:0] int_opcode;
            wire [31:0] int_shiftl16;
            //Control Signal
            wire int_memtoreg;
            wire int_memwrite;
            wire int_branch;
            wire [2:0] int_aluControl;
            wire int_aluSrc;
            wire int_regdst;
            wire int_regwrite;
            wire [31:0]int_signExtend;
            wire pc_jump;
            wire int_memRead;
            //ALU Src
            wire [31:0] int_aluSrcSelect;
            //ResultSelect
            wire [31:0] int_resultSel;
            //RegSelect
            wire [4:0] int_rsOrd;
            wire Hazardrst_int;

    //Calling MUX
     mux1 mux1_int (
                                        .branch(PCSRC),
                                        .pcvalue(pc_pcout),
                                        .branchvalue(branchAddress),
                                        .muxOut(mux1Out)
                                        );
```

```
   //Calling Program Counter
  pc pc_int                    (
                                         .clk(clk),
                                         .rst(rst),
                                         .pcOut(pc_pcout),
                                         .hazard(Hazard),
                                         .pcIn(mux1Out),
                                         .pcprev(pcStoredIF)
                                         );

  //Instantiating Instruction memory
   inst_mem instmem_int (
                                   .address(pc_pcout),
                                   .instruction(inst_instruct)
                               );

   //Calling the Fetch/Decode pipelined module
   IFID IFID_int                     (
                                         .clk(clk),
                                         .pcvalue(pc_pcout),
                                         .pcStored(pcStoredIF),
                                         .instructionvalue(inst_instruct),
                                         .instructionStored(instructionStored)
                                    );

   //Calling Decoder
   Decoder decode_int    (
                                   .instruct(instructionStored),
                                   .opcode(int_opcode),
                                   .rs(int_rs),
                                   .rt(int_rt),
                                   .rd(int_rd),
                                   .shamt(int_shamt),
                                   .func(int_func),
                                   .immed(int_immed),
                                   .signExtend(int_signExtend),
                                   .target(int_target),
                                   .slby16(int_shiftl16)
                              );

   //Instantiating the registers from reg file
   reg_file regfile_int (
                                   .rs(int_rs),
                                   .rt(int_rt),
                                   .rd(regMuxOutMEM),
                                   .rst(rst),
                        .write(regwriteWB),
                        .rsData(reg_rsData),
                        .rtData(reg_rtData),
                        .mem_write(MemRegMUXOut)
                );


//Calling Decode/Execute pipeline module
IDEX IDEX_int   (
                        .clk(clk),
                        .pcvalue(pcStoredIF),
                        .pcvalueStored(pcStoredID),
                        .rsDt(reg_rsData),
                        .rsDtStored(rsDTStored),
                        .rtDt(reg_rtData),
                        .rtDtStored(rtDTStoredID),
                        .signEx(int_signExtend),
                        .signExStored(signexStored),
                        .rtval(int_rt),
                        .rtvalStored(rtValStored),
                        .rdval(int_rd),
                        .rdvalStored(rdValStored),
                        .memtoreg(int_memtoreg),
                        .memtoregStored(memtoregID),
                        .memwrite(int_memwrite),
                        .memwriteStored(memwriteID),
                        .branch(int_branch),
                        .branchStored(branchID),
                        .aluControl(int_aluControl),
                        .aluControlStored(aluControlID),
                        .aluSrc(int_aluSrc),
                        .aluSrcStored(aluSrcID),
                        .regdst(int_regdst),
                        .regdstStored(regdstID),
                        .regwrite(int_regwrite),
                        .regwriteStored(regwriteID),
                        .rsval(int_rs),
                        .rsvalStored(rsValStored),
                        .memRead(int_memRead),
                        .memReadStored(memReadID),
                        .hazard(Hazard),
                        .branchDet(branchDetected)
                );

//Instantiating the destination register
dstRegMux dstRegMux_int (
                                        .regdst(regdstID),
                                        .rt(rtValStored),
                                        .rd(rdValStored),
                                        .regmuxOut(regMuxOut)
                                        );

//Calling BranchAdderModule

branchAdder branchAdder_int (
                                              .pc(pcStoredID),
                                              .signEx(signexStored),
                                              .branchAdderOut(branchAdderOUT)
                                         );
//Calling the Control Unit
ControlUnit control_int (
                                        .opcode(int_opcode),
                                        .func(int_func),
                                        .memtoreg(int_memtoreg),
                                        .memwrite(int_memwrite),
                                        .branch(int_branch),
                                        .aluControl(int_aluControl),
                                        .aluSrc(int_aluSrc),
                                        .regdst(int_regdst),
                                        .regwrite(int_regwrite),
                                        .jump(pc_jump),
                                        .memRead(int_memRead)
                                   );

//Calling Module for the Control Signal of ALU
AluSrc src_int  (
                                        .Rt(rtDTStoredID),
                                        .signExtend(int_signExtend),
                                        .alusrc(aluSrcID),
                                        .result(int_aluSrcSelect)
                             );
//Calling ALU
Alu Alu_int   (
                                        .srcA(ForwardAluSRCA),
                                        .srcB(ForwardAluSRCB),
                                        .opcode(aluControlID),
                                        .signExtend(int_signExtend),
                                        .zero(alu_zero),
                                        .result(alu_result)
                          );

//Forwarding unit
```

```
ForwardingUnit ForwardingUnit_int (
                                        .EXregwrite(regwriteEX),
                                        .WBregwrite(regwriteWB),
                                        .curRs(rsValStored),
                                        .curRt(rtValStored),
                                        .exRd(regMuxOutEX),
                                        .wbRd(regMuxOutMEM),
                                        .forwardA(ForwardA),
                                        .forwardB(ForwardB)
                                    );
ForwardMux1 ForwardMux1_int     (
                                        .forwardA(ForwardA),
                                        .curRSdt(rsDTStored),
                                        .EXRSdt(AluvalueStoredEX),
                                        .MEMRSdt(MemRegMUXOut),
                                        .aluSRCA(ForwardAluSRCA)
                                );
ForwardMux2 ForwardMux2_int     (
                                        .forwardB(ForwardB),
                                        .curRTdt(int_aluSrcSelect),
                                        .EXRTdt(AluvalueStoredEX),
                                        .MEMRTdt(MemRegMUXOut),
                                        .aluSRCB(ForwardAluSRCB)
                                );

//Calling Module for the Detection of Hazards
HazardDetection HazardDetection_int (
                                        .exMemRead(memReadEX),
                                        .curRs(int_rs),
                                        .curRt(int_rt),
                                        .exRd(rdValStored),
                                        .hazard(Hazard)
                                     );
//Branch Hazards
BranchHazard BranchHazard_int   (
                                        .branch(PCSRC),
                                        .branchDet(branchDetected)
                                );


//Calling the Execute/Memory Pipeline module
EXMEM EXMEM_int (
                                        .clk(clk),
                                        .pcvalue(branchAdderOUT),
                                        .pcvalueStored(branchAddress),
                                        .zero(alu_zero),
                                        .zeroStored(ZeroStored),
                                        .aluvalue(alu_result),
                                        .aluvalueStored(AluvalueStoredEX),
                                        .rtDt(rtDTStoredID),
                                        .rtDtStored(rtDtStoredEX),
                                        .rsrd(regMuxOut),
                                        .rsrdStored(regMuxOutEX),
                                        .memtoreg(memtoregID),
                                        .memtoregStored(memtoregEX),
                                        .memwrite(memwriteID),
                                        .memwriteStored(memwriteEX),
                                        .branch(branchID),
                                        .branchStored(branchEX),
                                        .regwrite(regwriteID),
                                        .regwriteStored(regwriteEX),
                                        .memRead(memReadID),
                                        .memReadStored(memReadEX),
                                        .branchDet(branchDetected)
                    );

//Module for Checking Branch Conditions
branchCondit branchCondit_int        (
                                        .branch(branchEX),
                                        .zero(ZeroStored),
                                        .pcSRC(PCSRC)
                                    );

//Instantiating Values from Memory by Calling DataMem
datamemory datamemory_int            (
                                        .writeEn(memwriteEX),
                                        .writedata(rtDtStoredEX),
                                        .a(AluvalueStoredEX),
                                        .rd(data_rd),
                                        .memRead(memReadEX)
                                    );

//Calling Memory/WriteBack Pipeline module
MEMWB MEMWB_int  (
                                    .clk(clk),
                                    .readvalue(data_rd),
                                    .readvalueStored(readValueStored),
                                    .aluvalue(AluvalueStoredEX),
                                    .aluvalueStored(AluValueStoredMEM),
                                    .rsrd(regMuxOutEX),
                                    .rsrdStored(regMuxOutMEM),
                                    .memtoreg(memtoregEX),
                                                          .memtoregStored(memtoregWB),
                                                          .regwrite(regwriteEX),
                                                          .regwriteStored(regwriteWB)
                                               );

    //Instantiating memTOregMux Module
    memTOregMux memTOregMux_int  (
                                                                    .memtoreg(memtoregWB),
                                                                    .readval(readValueStored),
                                                                    .aluval(AluValueStoredMEM),
                                                                    .MemRegOut(MemRegMUXOut)
                                                            );

endmodule
```

Other Modules

```
module mux1(
    input wire branch,            // Branch control signal
    input wire [31:0] pcvalue,    // Value from the PC
    input wire [31:0] branchvalue,// Branch target address
    output wire [31:0] muxOut     // Output of the multiplexer
);

    // Multiplexer logic: if branch is true, output branch value, otherwise output pcvalue
    assign muxOut = (branch) ? branchvalue : pcvalue;

endmodule

module pc #(parameter N=32)
(
    input clk,
    input rst,
    output reg [N-1:0] pcOut,
    input hazard,
    input [N-1:0] pcIn,
    input [31:0] pcprev
);

    always @(posedge clk or posedge rst)
    begin
        if (rst)
        begin
            pcOut <= {N{1'b0}};  // Reset pcOut to 0
        end
        else begin
```

```
            if (hazard) begin
                pcOut <= pcprev;  // Retain previous PC value during hazard
            end
            else begin
                if (pcIn == 31) begin
                    pcOut <= 0;  // Reset to 0 when pcIn reaches 31
                end
                else begin
                    pcOut <= pcIn + 1;  // Increment PC
                end
            end
        end
    end

endmodule

module inst_mem #(parameter N=32, D=32)
(
    input [N-1:0] address,
    output reg [N-1:0] instruction
);

    reg [N-1:0] memory [D-1:0];

    initial begin
        // Initialize memory with instructions
     memory[0]  = 32'b000000_00000_01010_00001_00000_000000; // lw   $t2, 0($t0)
     memory[1]  = 32'b000001_00011_00001_00000_00000_000100; // sw   $t4, 0($t0)
     memory[2]  = 32'b000100_00011_00001_00000_00000_100101; // or   $t3,$t4, $t1
     memory[3]  = 32'b000100_01011_00001_00000_00000_100010; // sub $t3,$t2, $t1
     memory[4]  = 32'b000100_00010_00011_00000_00000_100101; // or  $t4, $t2, $t3
     memory[5]  = 32'b000000_00011_00001_00000_00000_000100; // lw  $t4, 0($t0)
     memory[6]  = 32'b000010_00110_00101_00000_00000_000001; // beq
     memory[7]  = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1, $t0
     memory[8]  = 32'b000100_01010_01011_01001_00000_100000; // add $t2,$t2, $t3
     memory[9]  = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4, $t1
     memory[10] = 32'b000100_00001_01011_01010_00000_100000; // add $t2,$t1,$t3
     memory[11] = 32'b000011_00000_00000_00000_00000_000010; // jump
     memory[12] = 32'b000001_00000_01010_00001_00000_000000; // sw $t2, 0($t0)
     memory[13] = 32'b000000_00000_01011_00001_00000_000000; // lw $t3, 0($t0)
     memory[14] = 32'b000000_00000_01010_00001_00000_000000; // lw $t2, 0($t0)
     memory[15] = 32'b000100_01011_00001_00000_00000_100010; // sub $t3,$t2,$t1
     memory[16] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[17] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[18] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[19] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[20] = 32'b000100_00010_00001_00000_00000_100100; // and $t2,$t1,$t0
     memory[21] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[22] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[23] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[24] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[25] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[26] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[27] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[28] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[29] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
     memory[30] = 32'b000100_00010_00001_00000_00000_000001; // add $t2,$t1,$t0
     memory[31] = 32'b000100_00011_00001_00000_00000_000001; // add $t3,$t4,$t1
end

    always @* begin
        // Ensure the address is divided by 4 to index the instruction memory properly
        instruction = memory[address >> 2];
    end
endmodule

module IFID
(
    input clk,
    input rst,              // Reset signal to initialize registers
    input [31:0] pcvalue,   // Input PC value
    output reg [31:0] pcStored, // Stored PC value
    input [31:0] instructionvalue, // Input instruction value
    output reg [31:0] instructionStored // Stored instruction value
);

    // Always block triggered on positive edge of the clock or reset
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            // Initialize the registers to 0 on reset
            pcStored <= 32'b0;
            instructionStored <= 32'b0;
        end else begin
            // Store the current PC and instruction values on the rising edge of clock
            pcStored <= pcvalue;
            instructionStored <= instructionvalue;
        end
    end
```

```
endmodule

module Decoder(
    input [31:0] instruct,
    output reg [5:0] opcode,
    output reg [4:0] rs,
    output reg [4:0] rt,
    output reg [4:0] rd,
    output reg [4:0] shamt,
    output reg [5:0] func,
    output reg [15:0] immed,
      output reg [31:0] signExtend,
    output reg [25:0] target,
      output reg [31:0] slby16
    );
      always@(*)
      begin
      opcode<=instruct[31:26];
      rs<=instruct[25:21];
      rt<=instruct[20:16];
      rd<=instruct[15:11];
      shamt<=instruct[10:6];
      func<=instruct[5:0];
      immed<=instruct[15:0];
      slby16<=immed << 16;
      signExtend<={ {16{immed[15]}}, immed };
      target<=instruct[25:0];
      end

endmodule

module reg_file #(parameter N=32,D=32)(
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd,
    input rst,
    input write,
    output reg [N-1:0] rsData,
    output reg [N-1:0] rtData,
    input [N-1:0] mem_write
    );
      reg [N-1:0]memory [D-1:0];
      initial
      begin
      memory[0]=32'd550;
      memory[1]=32'd600;
      memory[2]=32'd550;
      memory[3]=32'd600;
      memory[4]=32'd432;
      memory[5]=32'd543;
      memory[6]=32'd800;
      memory[7]=32'd210;
      memory[8]=32'd140;
      memory[9]=32'd940;
      memory[10]=32'd1320;
      memory[11]=32'd160;
      memory[12]=32'd830;
      memory[13]=32'd400;
      memory[14]=32'd340;
      memory[15]=32'd270;
      memory[16]=32'd550;
      memory[17]=32'd1000;
      memory[18]=32'd220;
      end
      always@(*)
      begin
      rsData=memory[rs];
      rtData=memory[rt];
      end
      //assign rsData=memory[rs];
      //assign rtData=memory[rt];
      always@(*)
      begin
      if(write)
      memory[rd]<=mem_write;
      end

endmodule

module IDEX(
    input clk,
    input rst,              // Reset signal to initialize registers
    input [31:0] pcvalue,    // Input PC value
    output reg [31:0] pcvalueStored, // Stored PC value
    input [31:0] rsDt,       // Input RS data value
    output reg [31:0] rsDtStored, // Stored RS data
    input [31:0] rtDt,       // Input RT data value
    output reg [31:0] rtDtStored, // Stored RT data
    input [31:0] signEx,     // Input sign-extended value
    output reg [31:0] signExStored, // Stored sign-extended value
    input [4:0] rtval,       // Input RT register value
    output reg [4:0] rtvalStored, // Stored RT register value
    input [4:0] rdval,       // Input RD register value
    output reg [4:0] rdvalStored, // Stored RD register value
    input memtoreg,          // Control signal for mem-to-reg
    output reg memtoregStored, // Stored mem-to-reg control signal
    input memwrite,          // Control signal for memory write
    output reg memwriteStored, // Stored memory write control signal
    input branch,            // Control signal for branching
    output reg branchStored, // Stored branch control signal
    input [2:0] aluControl,  // ALU control signal
    output reg [2:0] aluControlStored, // Stored ALU control signal
    input aluSrc,            // ALU source control signal
    output reg aluSrcStored, // Stored ALU source control signal
    input regdst,            // Register destination control signal
    output reg regdstStored, // Stored regdst control signal
    input regwrite,          // Register write control signal
    output reg regwriteStored, // Stored register write control signal
    input [4:0] rsval,       // RS value (for forwarding)
    output reg [4:0] rsvalStored, // Stored RS value
    input memRead,           // Memory read control signal
    output reg memReadStored, // Stored memory read control signal
    input hazard,            // Hazard detection signal
    input branchDet          // Branch detection signal
);

    // Always block triggered on positive edge of the clock or reset
    always @(posedge clk or posedge rst)
    begin
        if (rst) begin
            // Initialize all registers to 0 on reset
            pcvalueStored <= 32'b0;
            rsDtStored <= 32'b0;
            rtDtStored <= 32'b0;
            signExStored <= 32'b0;
            rsvalStored <= 5'b0;
            rtvalStored <= 5'b0;
            rdvalStored <= 5'b0;
            memtoregStored <= 0;
            memwriteStored <= 0;
            branchStored <= 0;
            aluControlStored <= 3'b0;
            aluSrcStored <= 0;
            regdstStored <= 0;
            regwriteStored <= 0;
            memReadStored <= 0;
        end
        else if (hazard || branchDet) begin
            // Disable memory write and register write when hazard or branch detected
            memwriteStored <= 0;
            regwriteStored <= 0;
        end

        else begin
            // Store input values when no hazard or branch is detected
            pcvalueStored <= pcvalue;
            rsDtStored <= rsDt;
            rtDtStored <= rtDt;
            signExStored <= signEx;
            rsvalStored <= rsval;
            rtvalStored <= rtval;
            rdvalStored <= rdval;
            memtoregStored <= memtoreg;
            memwriteStored <= memwrite;
            branchStored <= branch;
            aluControlStored <= aluControl;
            aluSrcStored <= aluSrc;
            regdstStored <= regdst;
            regwriteStored <= regwrite;
            memReadStored <= memRead;
        end
    end

endmodule

module dstRegMux
(
    input wire regdst,
    input wire [4:0] rt,
    input wire [4:0] rd,
    output wire [4:0] regmuxOut
);
    assign regmuxOut = regdst ? rd : rt;

endmodule

module branchAdder
(
                input [31:0] pc,
                input [31:0] signEx,
                output reg [31:0] branchAdderOut
    );
```

```
      reg [31:0] branch;
      reg [4:0] brancH;
      always @(*)
      begin
      branch<=signEx <<2;
      brancH<=branch[4:0];
      branchAdderOut=pc + brancH;
      end

endmodule

module ControlUnit
(
      input [5:0] opcode,
      input [5:0] func,
      output reg memtoreg,
      output reg memwrite,
      output reg branch,
      output reg [2:0] aluControl,
      output reg aluSrc,
      output reg regdst,
      output reg regwrite,
      output reg jump,
      output reg memRead
    );

      always@(opcode or func)
      begin
      case(opcode)
            ///xori
            6'b001001:
                  begin
                        aluControl<=3'b111;
                        memtoreg<=0;
                        branch<=1'b0;
                        aluSrc<=1'b1;
                        regdst<=1'b0;
                        regwrite<=1'b1;
                        jump<=1'b0;
                        memRead<=1'b0;
                  end
            //slti
            6'b001000:
                  begin
                        aluControl<=3'b101;
                        memtoreg<=0;
				branch<=1'b0;
				aluSrc<=1'b1;
				regdst<=1'b1;
				regwrite<=1'b1;
				jump<=1'b0;
				memRead<=1'b0;
			end
		//slt
		6'b000111:
			begin
				aluControl<=3'b111;
				memtoreg<=0;
				branch<=1'b0;
				aluSrc<=1'b0;
				regdst<=1'b1;
				regwrite<=1'b1;
				jump<=1'b0;
				memRead<=1'b0;
			end
		//and
		6'b000110:
			begin
				aluControl<=3'b011;
				memtoreg<=0;
				branch<=1'b0;
				aluSrc<=1'b1;
				regdst<=1'b0;
				regwrite<=1'b1;
				jump<=1'b0;
				memRead<=1'b0;
			end
		//lui
		6'b000101:
			begin
				aluControl<=3'b010;
				memtoreg<=2'b10;
				branch<=1'b0;
				aluSrc<=1'b0;
				regdst<=1'b0;
				regwrite<=1'b1;
				jump<=1'b0;
				memRead<=1'b0;
			end
		//lw
		6'b000000:
			begin
```

```
                aluControl<=3'b010;
                memtoreg<=1'b1;
                branch<=1'b0;
                aluSrc<=1'b1;
                memwrite<=1'b0;
                regdst<=1'b1;
                regwrite<=1'b0;
                jump<=1'b0;
                memRead<=1'b1;
        end
//sw
6'b000001:
        begin
                aluControl<=3'b010;
                memtoreg<=1'b0;
                memwrite<=1'b1;
                branch<=1'b0;
                aluSrc<=1'b1;
                regdst<=1'b0;
                regwrite<=1'b0;
                jump<=1'b0;
                memRead<=1'b0;
        end
//beq
6'b000010:
        begin
                aluControl<=3'b110;
                memtoreg<=1'b0;
                memwrite<=1'b0;
                branch<=1'b1;
                aluSrc<=1'b0;
                regdst<=1'b0;
                regwrite<=1'b0;
                jump<=1'b0;
                memRead<=1'b0;
        end
//jump
6'b000011:
        begin
                aluControl<=3'b010;
                memtoreg<=2'b00;
                memwrite<=1'b1;
                branch<=1'b0;
                aluSrc<=1'b0;
                regdst<=1'b1;
                regwrite<=1'b1;
```

```
                        jump<=1'b1;
                        memRead<=1'b0;
                end
        6'b000100:
                case(func)
                        //add
                        6'b100000:
                                begin
                                        aluControl<=3'b010;
                                        memtoreg<=2'b00;
                                        memwrite<=1'b1;
                                        branch<=1'b0;
                                        aluSrc<=1'b0;
                                        regdst<=1'b1;
                                        regwrite<=1'b1;
                                        jump<=1'b0;
                                        memRead<=1'b0;
                                end
                        //sub
                        6'b100010:
                                begin
                                        aluControl<=3'b110;
                                        memtoreg<=2'b00;
                                        memwrite<=1'b1;
                                        branch<=1'b0;
                                        aluSrc<=1'b0;
                                        regdst<=1'b1;
                                        regwrite<=1'b1;
                                        jump<=1'b0;
                                        memRead<=1'b0;
                                end
                        //and
                        6'b100100:
                                begin
                                        aluControl<=3'b011;
                                        memtoreg<=2'b00;
                                        memwrite<=1'b1;
                                        branch<=1'b0;
                                        aluSrc<=1'b0;
                                        regdst<=1'b1;
                                        regwrite<=1'b1;
                                        jump<=1'b0;
                                        memRead<=1'b0;
                                end
                        //or
                        6'b100101:
```

```
                    begin
                        aluControl<=3'b001;
                        memtoreg<=2'b00;
                        memwrite<=1'b1;
                        branch<=1'b0;
                        aluSrc<=1'b0;
                        regdst<=1'b1;
                        regwrite<=1'b1;
                        jump<=1'b0;
                        memRead<=1'b0;
                    end
                //set on less than
                6'b101010:
                    begin
                        aluControl<=3'b111;
                        memtoreg<=2'b00;
                        memwrite<=1'b1;
                        branch<=1'b0;
                        aluSrc<=1'b0;
                        regdst<=1'b1;
                        regwrite<=1'b1;
                        jump<=1'b0;
                        memRead<=1'b0;
                    end
                default:
                    begin
                        aluControl<=3'b000;
                        memtoreg<=2'b00;
                        memwrite<=1'b0;
                        branch<=1'b0;
                        aluSrc<=1'b0;
                        regdst<=1'b0;
                        regwrite<=1'b0;
                        jump<=1'b0;
                        memRead<=1'b0;
                    end
            endcase
        default:
            begin
                aluControl<=3'b000;
                memtoreg<=2'b00;
                memwrite<=1'b0;
                branch<=1'b0;
                aluSrc<=1'b0;
                regdst<=1'b0;
                regwrite<=1'b0;
```

```
                                        jump<=1'b0;
                                        memRead<=1'b0;
                              end
                    endcase
      end

endmodule

module AluSrc
(
                input [31:0] Rt,
                input [31:0] signExtend,
                input alusrc,
                output reg [31:0] result
);
      always@(alusrc or Rt or signExtend)
      begin
      if(alusrc)
      result<=signExtend;
      else
      result<=Rt;
      end

endmodule

module Alu
(
  input [31:0] srcA,
  input [31:0] srcB,
  input [2:0] opcode,
  input [31:0] signExtend,
  output reg zero,
  output reg [31:0] result,
  output reg [31:0] HI_reg,
  output reg [31:0] LO_reg
);

    reg [31:0] evenCheck;
    wire [31:0] divi;
    assign divi=32'd2;
  always @(*) begin
    case (opcode)
                3'b010: result = srcA + srcB;
                3'b110: result = srcA - srcB;
      3'b010: result = srcA + srcB;
      3'b110: result = srcA - srcB;
      //3'b000: result = srcA & srcB;
                3'b000: begin
                result=srcA * srcB;
                {HI_reg,LO_reg}=result;
                end
      3'b001: result = srcA | srcB;
      //3'b100: result = srcA ^ srcB;
                //3'b111: result = srcA < srcB? 1:0;
                3'b111: begin    //// xori instruction
                evenCheck = srcA ^ signExtend;
                result= evenCheck%divi==0? evenCheck: srcB;
                end
                3'b101: result = srcA < signExtend? 1:0;
                3'b011: result = srcA & srcB;
      default: result = srcA;
    endcase

    if (result == 32'd0)
      zero <= 1'b1;
    else
      zero <= 1'b0;
  end

endmodule

module ForwardingUnit
(
                input EXregwrite,
                input WBregwrite,
                input [4:0] curRs,
                input [4:0] curRt,
                input [4:0] exRd,
                input [4:0] wbRd,
                output [1:0] forwardA,
                output [1:0] forwardB
    );
      assign forwardA = (EXregwrite && curRs==exRd) ? 2'b01:
                        (WBregwrite && curRs==wbRd) ? 2'b10:
                        2'b00;

      assign forward = (EXregwrite && curRt==exRd) ? 2'b01:
                       (WBregwrite && curRt==wbRd) ? 2'b10:
                       2'b00;
endmodule

module ForwardMux1
```

```
(
            input [1:0] forwardA,
            input [31:0] curRSdt,
            input [31:0] EXRSdt,
            input [31:0] MEMRSdt,
            output [31:0] aluSRCA
   );
     assign aluSRCA = (forwardA==2'b01) ? EXRSdt:
                          (forwardA==2'b10) ? MEMRSdt : curRSdt;

endmodule

module ForwardMux2
(
   input [1:0] forwardB,       // Forwarding control signal
   input [31:0] curRTdt,       // RT value from the current instruction
   input [31:0] EXRTdt,        // RT value from EX stage
   input [31:0] MEMRTdt,       // RT value from MEM stage
   output [31:0] aluSRCB       // Second operand for ALU
);

   // Forwarding logic
   assign aluSRCB = (forwardB == 2'b01) ? EXRTdt :  // Forward from EX stage
               (forwardB == 2'b10) ? MEMRTdt : // Forward from MEM stage
               curRTdt;                    // Use current RT value

endmodule

module HazardDetection
(
   input exMemRead,
   input [4:0] curRs,
   input [4:0] curRt,
   input [4:0] exRd,
   output reg hazard
);
    always @(*)
    begin
    hazard<=1'b0;
    hazard <=(exMemRead && ((curRs == exRd) || (curRt == exRd))) ? 1'b1: 1'b0;
    end
endmodule

module BranchHazard
(
            input branch,
            output reg branchDet
);
     always@(*)
     begin
     if(branch)
     branchDet<=1'b1;
     else
     branchDet<=1'b0;
     end

endmodule

module EXMEM
(
   input clk,
   input [31:0] pcvalue,
   output reg [31:0] pcvalueStored,
   input zero,
   output reg zeroStored,
   input [31:0] aluvalue,
   output reg [31:0] aluvalueStored,
   input [31:0] rtDt,
   output reg [31:0] rtDtStored,
   input [4:0] rsrd,
   output reg [4:0] rsrdStored,
   input memtoreg,
   output reg memtoregStored,
   input memwrite,
   output reg memwriteStored,
   input branch,
   output reg branchStored,
   input regwrite,
   output reg regwriteStored,
     input memRead,
     output reg memReadStored,
     input branchDet
);
   always @(posedge clk)
   begin
     if(branchDet)
     begin
     memwriteStored<=1'b0;
     regwriteStored<=1'b0;
     zeroStored <= 1'b0;
     end
     else
     begin
       pcvalueStored <= pcvalue;
       zeroStored <= zero;
       aluvalueStored <= aluvalue;
       rtDtStored <= rtDt;
       rsrdStored <= rsrd;
       memtoregStored <= memtoreg;
       memwriteStored <= memwrite;
       branchStored <= branch;
       regwriteStored <= regwrite;
                   memReadStored<=memRead;
                   end
   end
endmodule

module branchCondit
(
   input wire branch,
   input wire zero,
   output reg pcSRC
);
     always @(*)
     begin
   pcSRC <= (branch && zero) ? 1 : 0;
     end
endmodule

module datamemory
(
   input wire writeEn,
   input wire [31:0] writedata,
   input wire [31:0] a,
   output reg [31:0] rd,
   input wire memRead
   );

   reg [31:0] memory [0:31]; // Define a memory array of 32 elements

   // Initialize memory with some values
   initial begin
       memory[0] = 32'd5;
       memory[1] = 32'd50;
       memory[2] = 32'd55;
       memory[3] = 32'd6;
       memory[4] = 32'd4;
       memory[5] = 32'd43;
       memory[6] = 32'd80;
       memory[7] = 32'd21;
       memory[8] = 32'd40;
       memory[9] = 32'd90;
       memory[10] = 32'd20;
       memory[11] = 32'd10;
       memory[12] = 32'd30;
       memory[13] = 32'd40;
       memory[14] = 32'd39;
       memory[15] = 32'd27;
       memory[16] = 32'd52;
       memory[17] = 32'd1000;
       memory[18] = 32'd22;
       memory[19] = 32'd560;
       memory[20] = 32'd420;
       memory[21] = 32'd690;
       memory[22] = 32'd2;
       memory[23] = 32'd1;
       memory[24] = 32'd3;
       memory[25] = 32'd7;
       memory[26] = 32'd4;
       memory[27] = 32'd16;
       memory[28] = 32'd72;
       memory[29] = 32'd13;
       memory[30] = 32'd60;
       memory[31] = 32'd8;
   end

   // Memory write operation (synchronous)
   always @(*) begin
       if (writeEn) begin
           memory[a[4:0]] <= writedata; // Use only lower 5 bits of 'a' for addressing
       end
   end

   // Memory read operation (combinational)
   always @(*) begin
       if (memRead) begin
           rd = memory[a[4:0]]; // Use only lower 5 bits of 'a' for addressing
       end else begin
           rd = 32'b0; // Default value when memRead is not asserted
       end
   end

endmodule
```

```
module MEMWB
(
   input clk,                         // Clock signal
   input [31:0] readvalue,            // Data read from memory (Memory read value)
   output reg [31:0] readvalueStored,// Stored memory value for write-back stage
   input [31:0] aluvalue,            // ALU result value (ALU value after execution)
   output reg [31:0] aluvalueStored, // Stored ALU result for write-back stage
   input [4:0] rsrd,                 // Register destination (destination register address)
   output reg [4:0] rsrdStored,     // Stored destination register address
   input memtoreg,                   // Control signal to decide whether to write memory
data or ALU result
   output reg memtoregStored,        // Stored memtoreg control signal for write-back
   input regwrite,                   // Control signal to enable/disable register write
   output reg regwriteStored         // Stored regwrite control signal for write-back
);

   // Sequential logic to store values at the positive edge of the clock
   always @(posedge clk)
   begin
      // Store the values for the write-back stage (MEM -> WB pipeline stage)
      readvalueStored <= readvalue;    // Store memory read value
      aluvalueStored <= aluvalue;      // Store ALU result
      rsrdStored <= rsrd;              // Store destination register address
      memtoregStored <= memtoreg;      // Store memtoreg control signal
      regwriteStored <= regwrite;      // Store regwrite control signal
   end
endmodule

module memTOregMux
(
   input wire memtoreg,
   input wire [31:0] readval,
   input wire [31:0] aluval,
   output wire [31:0] MemRegOut
);
   assign MemRegOut = memtoreg ? readval : aluval;
endmodule
```

Test Bench

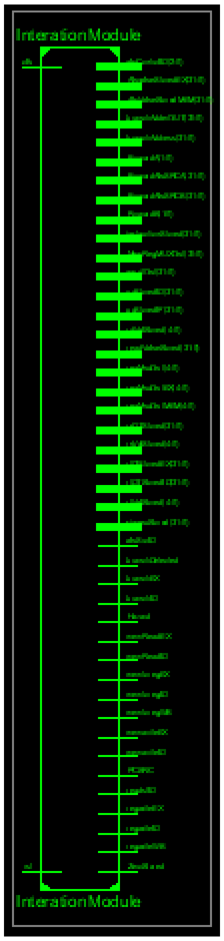
```
module MyTestBench;

    // Inputs
    reg clk;
    reg rst;
```

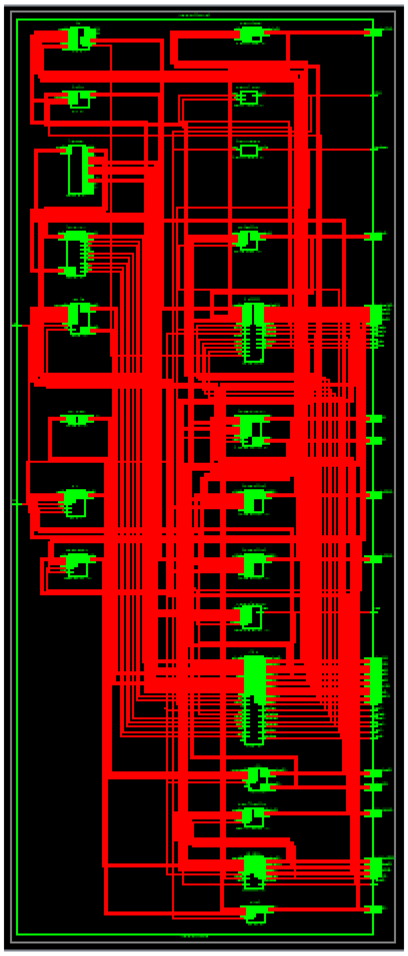
```
// Outputs
wire [31:0] pcStoredIF;
wire [31:0] pcStoredID;
wire [31:0] instructionStored;
wire [31:0] mux1Out;
wire [31:0] rsDTStored;
wire [31:0] rtDTStoredID;
wire [31:0] rtDtStoredEX;
wire [31:0] signexStored;
wire [4:0] rsValStored;
wire [4:0] rtValStored;
wire [4:0] rdValStored;
wire [4:0] regMuxOut;
wire [4:0] regMuxOutEX;
wire [4:0] regMuxOutMEM;
wire [31:0] branchAdderOUT;
wire [31:0] branchAddress;
wire ZeroStored;
wire [31:0] AluvalueStoredEX;
wire [31:0] AluValueStoredMEM;
wire PCSRC;
wire [31:0] readValueStored;
wire [31:0] MemRegMUXOut;
wire memtoregID;
wire memtoregEX;
wire memtoregWB;
wire memwriteID;
wire memwriteEX;
wire branchID;
wire branchEX;
wire [2:0] aluControlID;
wire aluSrcID;
wire regdstID;
wire regwriteID;
wire regwriteEX;
wire regwriteWB;
wire memReadID;
wire memReadEX;
wire [1:0] ForwardA;
wire [1:0] ForwardB;
wire [31:0] ForwardAluSRCA;
wire [31:0] ForwardAluSRCB;
wire Hazard;
wire branchDetected;

// Instantiate the Unit Under Test (UUT)
```

```
PipelineTop uut (
        .clk(clk),
        .rst(rst),
        .pcStoredIF(pcStoredIF),
        .pcStoredID(pcStoredID),
        .instructionStored(instructionStored),
        .mux1Out(mux1Out),
        .rsDTStored(rsDTStored),
        .rtDTStoredID(rtDTStoredID),
        .rtDtStoredEX(rtDtStoredEX),
        .signexStored(signexStored),
        .rsValStored(rsValStored),
        .rtValStored(rtValStored),
        .rdValStored(rdValStored),
        .regMuxOut(regMuxOut),
        .regMuxOutEX(regMuxOutEX),
        .regMuxOutMEM(regMuxOutMEM),
        .branchAdderOUT(branchAdderOUT),
        .branchAddress(branchAddress),
        .ZeroStored(ZeroStored),
        .AluvalueStoredEX(AluvalueStoredEX),
        .AluValueStoredMEM(AluValueStoredMEM),
        .PCSRC(PCSRC),
        .readValueStored(readValueStored),
        .MemRegMUXOut(MemRegMUXOut),
        .memtoregID(memtoregID),
        .memtoregEX(memtoregEX),
        .memtoregWB(memtoregWB),
        .memwriteID(memwriteID),
        .memwriteEX(memwriteEX),
        .branchID(branchID),
        .branchEX(branchEX),
        .aluControlID(aluControlID),
        .aluSrcID(aluSrcID),
        .regdstID(regdstID),
        .regwriteID(regwriteID),
        .regwriteEX(regwriteEX),
        .regwriteWB(regwriteWB),
        .memReadID(memReadID),
        .memReadEX(memReadEX),
        .ForwardA(ForwardA),
        .ForwardB(ForwardB),
        .ForwardAluSRCA(ForwardAluSRCA),
        .ForwardAluSRCB(ForwardAluSRCB),
        .Hazard(Hazard),
        .branchDetected(branchDetected)
```

```
    );

    initial begin
            // Initialize Inputs
            clk = 0;
            rst = 1;

            // Wait 100 ns for global reset to finish
            #10;

            // Deassert reset
            rst = 0;
    end

    // Clock generation process
    always begin
            #5 clk = ~clk; // Toggle clock every 5 ns (10 ns period)
    end

endmodule
```

RTL Schematic Diagram





Conclusion

In conclusion, the design and implementation of our pipelined processor successfully demonstrated effective hazard management through stalling, data forwarding, and a combined stall + forwarding approach. By carefully structuring the processor into key pipeline stages—Fetch, Decode, Execute, Memory Access, and Write-back—we ensured efficient instruction flow while minimizing delays caused by data and control hazards. The integration of hazard detection units and forwarding mechanisms allowed the processor to dynamically resolve dependencies, improving overall performance and reliability.
Through simulation, we validated the functionality and efficiency of our design under real-world conditions. Additionally, the synthesis report provided insights into resource utilization, highlighting areas for potential optimization. This project not only achieved its objectives but also enhanced our understanding of pipelined processor design, hazard management techniques, and digital hardware implementation.