Project
Set - 2

Statement: Design a 5-stage (as discussed in the lecture) pipeline RISC processor (with hazard detection and data forwarding unit as necessary) that can execute the following instructions:

0000 ADD reg1, reg2, reg3
0004 LW reg4, 7(reg1)
0008 SUBI reg5, reg4, 1234
0012 OR reg8, reg7, reg6
0016 NOR reg10, reg9, reg8

Initialize the register file with the following data
reg2 = 9 reg3 = 10
reg6 = 688CA reg7 = 964EA
reg9 = 1212E

The instruction format is as follows:

31 28	27 24	23 20	19 16	15 0
OPCODE	Dest Reg (WN)	Source Reg 1 (RN1)	Source Reg 2 (RN2)	Immediate value

Opcode for each instruction:
Instruction 1: 0000
Instruction 2: 0001
Instruction 3: 0011
Instruction 4: 0111
Instruction 5: 1111

The processor has the following control signals:
- ALUSrc - Select the second input of ALU
- ALUOp (2 bits) - Control ALU operation
- MR - Read data from memory
- MW - Write data into memory
- MReg - Move data from memory to register
- EnIM - Read instruction memory contents
- EnRW - Write data into the register file
- FA - Forward A mux control (used in data forwarding circuitry)
- FB - Forward B mux control (used in data forwarding circuitry)
- IFIDWrite - Disable IF/ID change (used in hazard detection circuit)

- PCWrite - Disable PC change (used in hazard detection circuit)
- ST - Control signal of mux which changes all control signals to zero (used in hazard detection circuit)

Initialize PC with all zeros. The instruction memory is of size 32 bytes, and the processor has 16 registers, numbered from reg0 to reg15, and the registers are 32 bits wide. A read operation from the instruction memory outputs 4 consecutive bytes of information (starting from the byte address provided to the memory) at the positive edge of the clock if the EnIM control signal is high. Assume that the register file has two 32-bit read ports: RD1 and RD2, for data reading and one 32-bit write port: WD, for data writing. At the rising edge of the clock, the read ports RD1 and RD2 output the data from the registers whose addresses are available at RN1 and RN2, respectively. At the falling edge of a clock, data is written via the write port WD to the register file whose address is present at WN if the EnRW signal is true. Design the data memory size as per the requirement. All the data are in hexadecimal format unless specified.

Draw the detailed architecture-level diagram of the processor, depicting all the blocks (e.g., register file, instruction memory, ALU, PC, combinational functional blocks, hazard detection unit, forwarding unit, etc.). Describe each block individually, and create behavioral verilog models for each architectural block separately. Build the top-level structural model of the processor by instantiating and interconnecting the individual architectural blocks. Specify the size and format of all pipeline registers including the fields holding the decoded control signals as well as data. Show all the input, output, and control signal waveforms in the report.

A Project Report
On

5 - STAGE PIPELINE RISC PROCESSOR

BY

ANVAY SAWAI 2022A3PS0701H
ANIRBAN NAYAK 2022A3PS0705H
MOHAMMAD ZAHID 2022A3PS0663H
SANIYA SHAHI 2022A8PS0810H

GROUP 2 - SET 2

Under the supervision of
DR. S GURUNARAYANAN

SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS OF COMPUTER ARCHITECTURE (CS F342)

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)
HYDERABAD CAMPUS
(APRIL 2025)

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to Dr. S Gurunarayanan for their invaluable guidance, support, and encouragement throughout this project. Their expertise and dedication have been instrumental in overcoming challenges we faced while working on this project. We deeply appreciate their willingness to share their knowledge and take the time out of their busy schedules to assist us.

We are grateful to Birla Institute of Technology and Science – Pilani, Hyderabad Campus for providing us with this wonderful opportunity to work on this project. We are grateful for the necessary resources and the conducive environment that we were provided with for this project.

Lastly, we would like to thank our friends, family and teachers for their constant support and motivation that helped us immensely while working on this project.

ABSTRACT

In this project, we designed and implemented a five-stage pipelined RISC processor capable of executing a fixed set of instructions including ADD, LW, SUBI, OR, and NOR. The architecture follows a classic pipeline model, consisting of Instruction Fetch, Instruction Decode, Execute, Memory, and Write-Back stages. To maintain efficient execution and handle data dependencies, we incorporated a hazard detection unit and a data forwarding unit. The processor was built with a 32-byte instruction memory and a 16-register file, each register being 32 bits wide. Our design supports concurrent instruction processing, ensuring that instructions can overlap across stages while preserving correct execution through careful control of pipeline hazards. Behavioral Verilog models were developed for all core modules, including the program counter, register file, ALU, memory units, and pipeline registers. The control logic was divided into a main control unit and an ALU control unit, simplifying instruction decoding and execution flow. Extensive simulation and verification were conducted using a custom testbench that provided clock signals, managed resets, and monitored key processor states. The processor successfully handled all specified instructions and demonstrated correct data hazard resolution through stalling and forwarding mechanisms. The waveform outputs and register state observations validated the functional correctness of the complete design. This project highlights the importance of structured pipeline management, efficient hazard handling, and modular design in building scalable and reliable processor architectures.

CONTENTS

INTRODUCTION

Pipelining is one of the most important techniques used in modern processor architectures to enhance instruction throughput and overall performance. By dividing instruction execution into multiple stages, each dedicated to a specific task, pipelining allows multiple instructions to be processed simultaneously at different stages of execution. In this project, we designed and implemented a 5-stage pipelined RISC processor that supports a set of basic arithmetic and logical instructions along with memory access operations. The five stages in our processor are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB), following the classic RISC pipeline model discussed in our course lectures.

The main objective of this project was to create a pipelined processor capable of executing a fixed instruction set, handling data hazards efficiently using hazard detection and data forwarding mechanisms. To achieve this, we designed each functional block—such as the Program Counter (PC), Instruction Memory, Register File, Arithmetic Logic Unit (ALU), Data Memory, and specialized control units—as individual Verilog modules. Each pipeline stage was separated by pipeline registers to hold both control and data signals, ensuring proper synchronization between stages during instruction flow.

In pipelined architectures, data hazards are inevitable when instructions depend on the results of previous instructions still in progress. To maintain correctness, we implemented a hazard detection unit that stalls the pipeline when a load-use hazard is detected. In addition, we designed a forwarding unit to resolve most data hazards by forwarding the needed data directly from later pipeline stages back to the EX stage, reducing unnecessary stalls and maintaining pipeline efficiency.

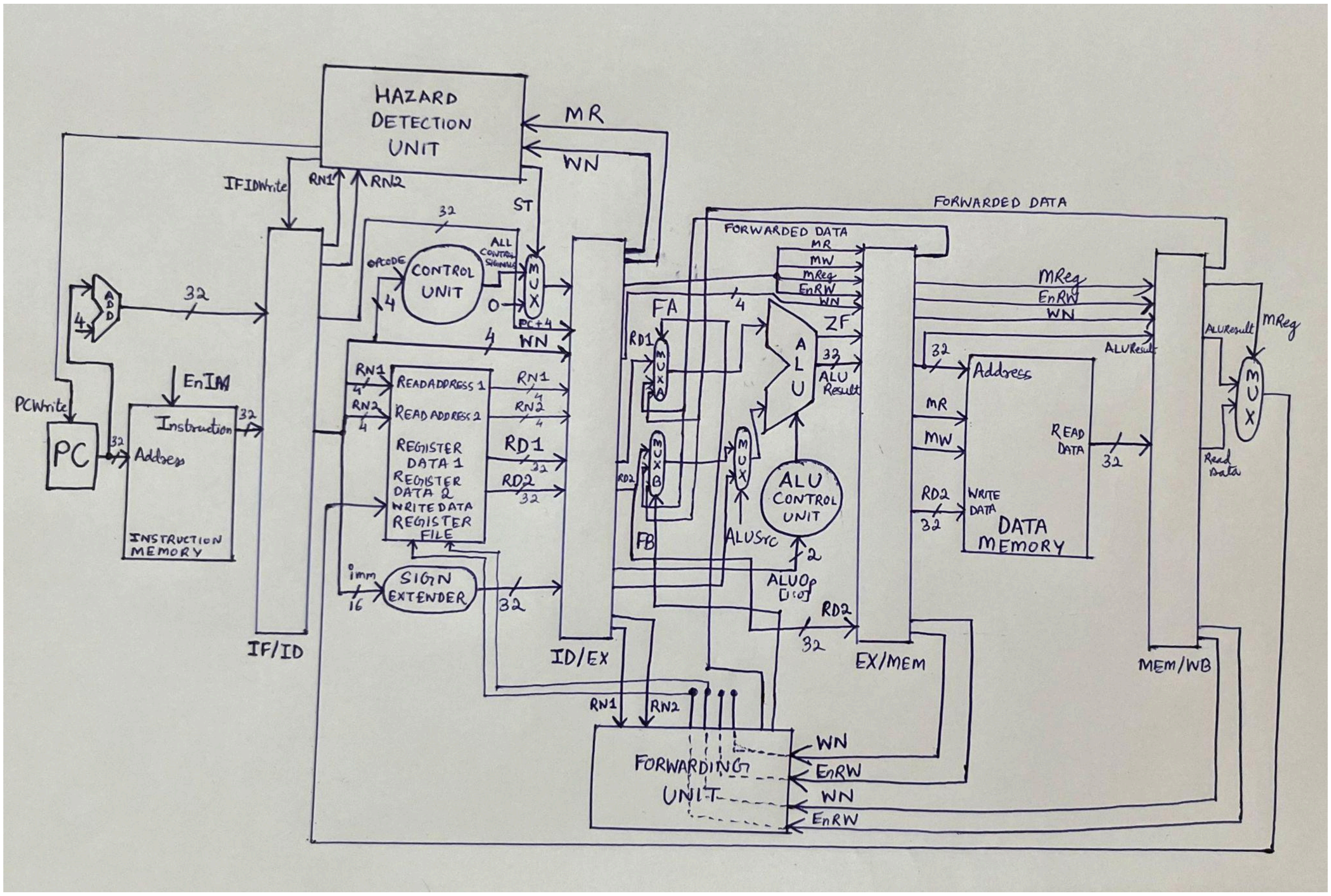
Our processor supports the execution of five specific instructions: ADD, LW (Load Word), SUBI (Subtract Immediate), OR, and NOR. Each instruction is encoded with a 4-bit opcode, and the instruction format was fixed as specified, with fields for opcode, destination register, source registers, and immediate values. To manage these operations, we developed a Main Control Unit that generates the necessary control signals based on the opcode and an ALU Control Unit that translates ALUOp and function codes into specific ALU operations.

The instruction memory was initialized with the required instructions, and the register file was preloaded with specific values to match the problem statement. We used two separate read ports and one write port in the register file, supporting simultaneous operand reading and result writing as per RISC design principles. Our ALU performs arithmetic and logical operations depending on the control inputs from the ALU control unit.

Simulation was carried out using a Verilog testbench that generated clock and reset signals, dumped waveform files for visualization, and monitored key internal signals such as the Program Counter, current instruction, and key register contents. This allowed us to verify instruction execution across pipeline stages, observe data hazards, and confirm correct behavior of stalling and forwarding mechanisms.

Through this project, we developed a comprehensive understanding of pipelined processor design, hazard management, modular hardware development, and system-level debugging. This work lays a strong foundation for more advanced concepts such as branch prediction, multi-cycle operations, and deeper pipeline architectures in future processor designs.

DETAILED ARCHITECTURE LEVEL BLOCK DIAGRAM OF PROCESSOR



DESCRIPTION OF BLOCKS

1. PC ADDER

The PC adder is responsible for calculating the address of the next instruction to be fetched. Generally, in a RISC architecture, since each instruction is 4 bytes wide, we simply add 4 to the current PC value to move sequentially through memory. In our code, the `PC_adder` module takes a 5-bit input `pc_in` and outputs `npc` by adding 4. This ensures the processor fetches the correct next instruction every clock cycle. Although simple, the PC adder is essential for maintaining proper instruction flow and smooth operation of the pipeline stages in our design.

2. INSTRUCTION MEMORY

The instruction memory is responsible for storing and supplying the instructions that drive the pipeline stages. Instruction memory holds all the program code and allows the processor to fetch instructions sequentially or based on control logic. In our implementation, the `imem` module is a 32-byte memory array, preloaded with five instructions during initialization. Whenever the `EnIM` signal is active, the memory outputs a 32-bit instruction by reading four consecutive bytes starting from the given address. This simple setup ensures that our processor consistently fetches valid instructions every cycle and can smoothly execute the programmed instruction sequence.

3. SIGN EXTENDER

In our processor, the sign extension unit is crucial for properly handling immediate values during arithmetic and memory operations. This unit takes a smaller-width value and extends it to a larger width while preserving its sign, ensuring that negative and positive numbers are correctly interpreted by the ALU. In our implementation, the `sgn_extnd` module takes a 16-bit immediate value and extends it to 32 bits by replicating the most significant bit across the upper 16 bits. This way, we maintain the correct value representation whether the number is positive or negative, allowing our processor to perform accurate computations.

4. REGISTER FILE

The register file is essential for storing and providing quick access to operand data during instruction execution. Generally, a register file is a small, fast memory consisting of multiple registers that the processor uses for computation. We have implemented the `regfile` module with 16 registers, each 32 bits wide. We initialized specific registers with required values for our instruction set. The module reads two registers simultaneously through `rd1` and `rd2`, and writes back a value on the negative clock edge if the `EnRW` signal is enabled. This structure ensures efficient data access and smooth instruction flow.

5. CONTROL UNIT

The control unit is the brain of our processor, and is responsible for interpreting the instruction's opcode and generating the appropriate control signals needed to guide the operation through the pipeline stages. It determines the behavior of key components like the ALU, register file, and memory, depending on the type of instruction being executed. Without it, the hardware modules would not know how to behave, and the processor would not be able to correctly perform operations like addition, subtraction, or memory access. We've implemented the `ctrlunit` module that takes the 4-bit opcode as input and produces several important control outputs: ALUctrl, EnRW, ALUsrc, MReg, MR, and MW. Based on the opcode, we set ALUctrl to define the exact ALU operation, such as Add, Subtract, OR, or NOR. We configure EnRW to enable or disable register writes, while ALUsrc selects between register data and an immediate value as the ALU's second operand. MReg controls whether the data written back to the register comes from memory or directly from the ALU. MR and MW manage memory read and write operations, respectively. Our code uses a simple conditional structure, where each instruction type sets the necessary control signals to ensure correct data flow and operation. This setup allows us to handle different instruction types efficiently without overcomplicating the logic.

6. ALU SOURCE MUX

In our processor design, the ALU source multiplexer and the ALU itself are crucial components that work together to perform the core arithmetic and logic operations needed during instruction execution. The ALU source MUX is responsible for selecting the correct second input for the ALU. It chooses between using a value read from a register or an immediate value extended to 32 bits, based on the control signal `ALUsrc`. If `ALUsrc` is high, the MUX selects the extended immediate; otherwise, it selects the register value. This flexibility allows our processor to efficiently support both immediate and register-based instructions without needing separate ALU paths. Once the correct inputs are selected, the ALU performs the operation dictated by the `ALUctrl` control signal. Our ALU supports several operations such as AND, OR, Addition, Subtraction, NOR, and Set-on-Less-Than, depending on the 3-bit control input. The result is computed in a purely combinational manner and provided on the output each time the inputs or control signals change. Additionally, the ALU outputs a `zero` signal, which indicates if the result is zero, helping in conditional operations if needed later.

7. DATA FORWARDING UNIT AND FORWARDING MUX

The data forwarding unit is critical for maintaining smooth instruction flow by minimizing pipeline stalls caused by data hazards. This unit is meant to detect when an instruction in a later pipeline stage has already computed a result that an earlier instruction currently needs. Instead of stalling the pipeline and waiting for the register write-back to complete, we forward the needed data directly from the appropriate pipeline register to the ALU input.We've implemented this concept through the `data_forwarding` module that takes signals indicating whether the EX/MEM and MEM/WB stages are writing back to a register, as well as the destination and source register addresses. For each source operand, we first check if the result can be forwarded from the EX/MEM stage, ensuring it is not a load instruction since memory read operations do not have data ready yet. If not, we check the MEM/WB stage for a match and forward data from there if available. The outputs `FA` and `FB` control the selection of inputs to the ALU for the two operands.

By using this approach, we allow dependent instructions to proceed without waiting unnecessarily, improving the processor's throughput and reducing the performance penalties caused by data hazards. Our forwarding logic is simple but very effective, ensuring that the ALU always operates with the latest available data without risking incorrect execution.

8. DATA MEMORY

Data memory module is responsible for handling all memory read and write operations during the memory access stage. It stores the values needed for load and store instructions, allowing the processor to interact with external data beyond the register file. In our implementation, the `data_mem` module uses an array of bytes to represent memory, supporting both 32-bit read and write operations. When the `memread` signal is active, we read four consecutive bytes starting from the given address and combine them into a 32-bit word. If `memwrite` is active, we write the input data `wd` into four consecutive bytes at the specified address on the rising edge of the clock. We initialized some memory locations with specific values to allow proper simulation and testing. This setup ensures that our processor can handle load and store instructions efficiently and maintain proper data flow across the pipeline.

9. MEM TO REG MUX

The Memory-to-Register mux is responsible for selecting the correct data to be written back into the register file during the write-back stage. In general, depending on the instruction, we either need to write the result from the ALU or the data read from memory. Our `MUX_wb` module takes in both the ALU output and the memory output and uses the `MemtoReg` control signal to choose between them. If `MemtoReg` is high, we select the ALU result; otherwise, we select the memory data. This setup allows our processor to correctly complete both computational and load operations.

10. HAZARD DETECTION LOGIC

Finally, we have the Hazard Detection Unit that plays an essential role in maintaining the correct flow of instructions through the pipeline, especially when data hazards occur. In general, the hazard detection unit monitors the pipeline for situations where the next instruction depends on the result of a previous instruction that has not yet completed its memory access or execution stage. If such a dependency is detected, the unit takes corrective action to stall the pipeline and prevent incorrect instruction execution. Our `hazard_detection` module checks if the instruction currently in the ID/EX stage is performing a memory read, and whether its destination register (`ID_EX_RD`) matches either the source registers (`IF_ID_RS` or `IF_ID_RT`) of the instruction currently in the IF/ID stage. If a match is found and a memory read is active, we generate a stall by disabling updates to the PC and the IF/ID pipeline register using `PCWrite` and `IFIDWrite` signals. Additionally, we assert the `ST` signal to insert a bubble into the pipeline, effectively giving the earlier instruction time to complete before allowing the dependent instruction to proceed. By doing this, we ensure that the processor maintains correct data dependencies without executing wrong results. When no hazard is detected, all control signals are set to allow normal pipeline progression. This mechanism is simple yet crucial for ensuring the smooth operation and correctness of our pipelined processor, especially when handling load-use data hazards.

VERILOG CODE

DATAPATH

PC Adder

```
module PC_adder(
input [4:0] pc_in,
output [4:0] npc
);
    assign npc= pc_in + 4;
endmodule
```

Instruction Memory

```
module imem(
input EnIM,
input [4:0] addr,
output reg [31:0] instr,
input clk
);

    reg [7:0] inst_mem[31:0];

    initial begin
        {inst_mem[0],inst_mem[1],inst_mem[2],inst_mem[3]} = 32'h0123_0000;
        {inst_mem[4],inst_mem[5],inst_mem[6],inst_mem[7]} = 32'h1410_0007;
        {inst_mem[8],inst_mem[9],inst_mem[10],inst_mem[11]} = 32'h3540_1234;
        {inst_mem[12],inst_mem[13],inst_mem[14],inst_mem[15]} = 32'h7876_0000;
        {inst_mem[16],inst_mem[17],inst_mem[18],inst_mem[19]} = 32'hFA98_0000;
```

```
    end
    always @(*) begin
        if (EnIM)
            instr = {inst_mem[addr], inst_mem[addr+1], inst_mem[addr+2],
inst_mem[addr+3]};
        else
            instr = 32'h0; // Output NOP when disabled
    end
endmodule
```

Sign Extender

```
module sgn_extnd(
input [15:0] imm,
output [31:0] extnd_imm
);
    assign extnd_imm = { {16{imm[15]}}, imm };
endmodule
```

Register File

```
module regfile(
input [3:0] rn1, rn2, wn,
input [31:0] wd,
input  EnRW,
output [31:0] rd1, rd2,
input clk
);
    reg [31:0] register[15:0];
    initial begin
```

```
        register[2] = 32'h9;

        register[3] = 32'h10;

        register[6] = 32'h688CA;

        register[7] = 32'h964EA;

        register[9] = 32'h1212E;

    end


    assign

        rd1= register[rn1],

        rd2= register[rn2];


    always@(negedge clk) begin

        if (EnRW == 1'b1) begin

        register[wn]<= wd;

        end

    end

endmodule
```

ALU Source MUX

```
module MUX_alusrc(

input [31:0] b,extnd_imm,

input ALUsrc,

output [31:0] in2

);

    assign in2 = ALUsrc ? extnd_imm : b;

endmodule
```

ALU

```
module alu(
input [31:0] in1, in2,
input [2:0] ALUctrl,
output reg [31:0] result,
output zero );
    assign zero = (result == 0);
    always @(*) begin
        case(ALUctrl)
        3'b000: result = (in1 & in2);
        3'b001: result = (in1 | in2);
        3'b010: result = in1 + in2;
        3'b011: result = ~(in1 | in2);
        3'b110: result = in1 - in2;
        3'b111: result = (in1 < in2) ? 32'h1 : 32'h0;
        default: result = 0;
        endcase
    end
endmodule
```

Data Memory

```
module data_mem(
input [31:0] addr,
input [31:0] wd,
input memread,memwrite,
output reg [31:0] rd,
input clk
);
```

```
    reg [7:0] data_mem [128:0];

    initial begin

        {data_mem[32], data_mem[33], data_mem[34], data_mem[35]} = 32'h12345678;

        {data_mem[26], data_mem[27], data_mem[28], data_mem[29]} = 32'h12345678;

        {data_mem[20], data_mem[21], data_mem[22], data_mem[23]} = 32'h12345678;

    end


    always @(*) begin

        if (memread) begin

            rd = {data_mem[addr], data_mem[addr + 1], data_mem[addr + 2], data_mem[addr
+ 3]};

        end

        else begin

            rd = 32'h0;

        end

    end

    always @(posedge clk) begin

        if (memwrite) begin

            data_mem[addr] <= wd[31:24];

            data_mem[addr+1] <= wd[23:16];

            data_mem[addr+2] <= wd[15:8];

            data_mem[addr+3] <= wd[7:0];

        end

    end

endmodule
```

Memory to Register MUX

```
module MUX_wb(
```

```
input MemtoReg,
input [31:0] data_out, ALUout,
output [31:0] reg_wd
);
    assign reg_wd = MemtoReg ? ALUout:data_out;
endmodule
```

CONTROL UNIT

```
module ctrlunit(
input [3:0] opcode,
output reg [2:0] ALUctrl,
output reg  EnRW, ALUsrc, MReg, MR, MW,
input clk
);

    always@(*) begin
        if (opcode== 4'b0001) begin //lw
            ALUctrl= 3'b010; //add
            EnRW=1'b1;
            ALUsrc= 1'b1;
            MReg= 1'b0;
            MR= 1'b1;
            MW= 1'b0;
            end

        else if (opcode == 4'b0000) begin // ADD
            ALUctrl = 3'b010;   //add
```

```
    EnRW = 1'b1;

    ALUsrc = 1'b0;

    MReg = 1'b1;

    MR = 1'b0;

    MW = 1'b0;

    end


else if (opcode== 4'b0010) begin //sw

    ALUctrl=3'b010;  //add

    EnRW=1'b0;

    ALUsrc= 1'b1;

    MReg= 1'b1;

    MR= 1'b0;

    MW= 1'b1;


    end

else if (opcode== 4'b0011) begin //subi

    MR= 1'b0;

    MW= 1'b0;

    ALUsrc= 1'b1;

    MReg= 1'b1;

    ALUctrl= 3'b110; //sub

    EnRW=1'b1;

    end

else if (opcode== 4'b0111) begin //or

    MR= 1'b0;

    MW= 1'b0;
```

```
            ALUsrc= 1'b0;
            MReg= 1'b1;
            ALUctrl= 3'b001; //or
             EnRW=1'b1;
            end
        else if (opcode== 4'b1111) begin //nor
            MReg= 1'b1;
            MR= 1'b0;
            MW= 1'b0;
            ALUsrc= 1'b0;
            ALUctrl= 3'b011; //nor
            EnRW=1'b1;
            end
    end

endmodule
```

HAZARD DETECTION UNIT AND DATA FORWARDING

```
module hazard_detection(
    input ID_EX_MemRead,
    input [3:0] ID_EX_RD,
    input [3:0] IF_ID_RS,
    input [3:0] IF_ID_RT,
    output reg PCWrite,
    output reg IFIDWrite,
    output reg ST
);
```

```
    initial begin
        PCWrite = 1'b1;
        IFIDWrite = 1'b1;
        ST = 1'b0;
    end

    always @(*) begin
        PCWrite = 1'b1;
        IFIDWrite = 1'b1;
        ST = 1'b0;
        if (ID_EX_MemRead && ((ID_EX_RD == IF_ID_RS) || (ID_EX_RD == IF_ID_RT))) begin
            PCWrite = 1'b0;     //for stalling PC
            IFIDWrite = 1'b0;   //for stalling IF/ID
            ST = 1'b1;          //bubble
        end
    end
endmodule

module data_forwarding(
    input EX_MEM_RegWrite, MEM_WB_RegWrite,
    input EX_MEM_MemRead,
    input [3:0] EX_MEM_RD,
    input [3:0] MEM_WB_RD,
    input [3:0] ID_EX_RS, ID_EX_RT,
    output reg [1:0] FA,FB
);
    always @(*) begin
```

```
        //forwarding for FA
        if (EX_MEM_RegWrite && !EX_MEM_MemRead && (EX_MEM_RD == ID_EX_RS))
            FA = 2'b10; //forward EX/MEM ALU result
        else if (MEM_WB_RegWrite && (MEM_WB_RD == ID_EX_RS))
            FA = 2'b01; //forward MEM/WB data
        else
            FA = 2'b00;

        //forwarding for FB
        if (EX_MEM_RegWrite && !EX_MEM_MemRead && (EX_MEM_RD == ID_EX_RT))
            FB = 2'b10;
        else if (MEM_WB_RegWrite && (MEM_WB_RD == ID_EX_RT))
            FB = 2'b01;
        else
            FB = 2'b00;
    end
endmodule
```

TOP MODULE

```
module top(
);
    reg [4:0]PC;
    reg clk;
    initial begin
        clk = 0;
        PC=0;
    end
```

```
    always begin
        #5 clk = ~clk;
    end

//IF_ID reg
reg [31:0] IF_ID_IR;
reg [4:0] IF_ID_NPC;

//ID_EX reg
reg [4:0] ID_EX_NPC;
reg [31:0] ID_EX_A,ID_EX_B, ID_EX_IR;
reg [31:0] ID_EX_IMM;
reg [3:0] ID_EX_RD, ID_EX_RS, ID_EX_RT;
reg [2:0] ID_EX_ALUctrl;
reg ID_EX_RegWrite;
reg ID_EX_MemtoReg;
reg ID_EX_MemRead, ID_EX_MemWrite;
reg ID_EX_ALUsrc;

//EX_MEM reg
reg [31:0] EX_MEM_IR, EX_MEM_ALUout, EX_MEM_B;
reg [3:0] EX_MEM_RD;
reg EX_MEM_MemtoReg;
reg EX_MEM_RegWrite;
reg EX_MEM_MemRead, EX_MEM_MemWrite;

//MEM_WB reg
```

```
reg [3:0] MEM_WB_RD;

reg [31:0] MEM_WB_ALUout;

reg [31:0] MEM_WB_IR;

reg [31:0] MEM_WB_DATA;

reg MEM_WB_RegWrite;

reg MEM_WB_MemtoReg;


//wires

wire [31:0] instr, nxt_instr, rd1,rd2, extnd_imm;

wire [4:0] NPC;

wire [31:0] reg_wd, data_rd;

wire [15:0] imm;

wire [3:0] opcode;

wire [3:0] rs,rt,rd;

wire [31:0] mux1_out;

wire [31:0] ALUout;

wire zero, EnRW, ALUsrc, MReg, MR;

wire [2:0]ALUctrl;

wire [1:0] FA,FB;

wire [31:0] forwardA_mux_out, forwardB_mux_out;

wire PCWrite, IFIDWrite, ST;


//IF blocks______________________________________________________________

PC_adder nextpc(.pc_in(PC), .npc(NPC));

imem instr_mem( .addr(PC), .instr(instr), .EnIM(PCWrite));

imem next_instr( .addr(NPC), .instr(nxt_instr), .EnIM(PCWrite));
```

```
//ID blocks________________________________________________________________

sgn_extnd sgn_extnd(.imm(imm), .extnd_imm(extnd_imm));


regfile reg_file(.clk(clk),.rn1(rs), .rn2(rt), .wn(MEM_WB_RD),
    .wd(reg_wd), .EnRW(MEM_WB_RegWrite), .rd1(rd1), .rd2(rd2));


ctrlunit ctrl_call(.opcode(opcode), .ALUctrl(ALUctrl),
     .EnRW(EnRW), .ALUsrc(ALUsrc),
     .MReg(MReg), .MR(MR), .MW(MW));


//EX blocks________________________________________________________________

MUX_alusrc m1(.ALUsrc(ID_EX_ALUsrc), .b(forwardB_mux_out), .extnd_imm(ID_EX_IMM),
.in2(mux1_out));


assign forwardA_mux_out =
    (FA == 2'b10) ? EX_MEM_ALUout :
    (FA == 2'b01) ? reg_wd :
    ID_EX_A;
assign forwardB_mux_out =
    (FB == 2'b10) ? EX_MEM_ALUout :
    (FB == 2'b01) ? reg_wd :
    ID_EX_B;


alu alu_call(.in1(forwardA_mux_out), .in2(mux1_out),
    .ALUctrl(ID_EX_ALUctrl), .result(ALUout), .zero(zero) );


//MEM blocks_______________________________________________________________
```

```
data_mem data_mem( .memread(EX_MEM_MemRead), .memwrite(EX_MEM_MemWrite),

        .addr(EX_MEM_ALUout), .wd(EX_MEM_B), .rd(data_rd), .clk(clk));


//WB blocks_______________________________________________________________

MUX_wb m3(.MemtoReg(MEM_WB_MemtoReg), .data_out(MEM_WB_DATA),

            .ALUout(MEM_WB_ALUout), .reg_wd(reg_wd));


//hazard detection_______________________________________________________________

hazard_detection hdu( .ID_EX_MemRead(ID_EX_MemRead), .ID_EX_RD(ID_EX_RD),
.IF_ID_RS(rs),

    .IF_ID_RT(rt), .PCWrite(PCWrite), .IFIDWrite(IFIDWrite), .ST(ST) );


data_forwarding forwarding(.EX_MEM_RegWrite(EX_MEM_RegWrite),
.MEM_WB_RegWrite(MEM_WB_RegWrite),

    .EX_MEM_RD(EX_MEM_RD), .EX_MEM_MemRead(EX_MEM_MemRead), .MEM_WB_RD(MEM_WB_RD),

    .ID_EX_RS(ID_EX_RS), .ID_EX_RT(ID_EX_RT),

    .FA(FA), .FB(FB) );


//stages

always@(posedge clk) begin //IF

    if(PCWrite) begin

        IF_ID_IR<= instr;

        IF_ID_NPC<= NPC;

        PC<= NPC;

        end

    end


//Instruction Decoding Logic
```

```
assign opcode= IF_ID_IR[31:28];
assign rd= IF_ID_IR[27:24];
assign rs= IF_ID_IR[23:20];
assign rt= IF_ID_IR[19:16];
assign imm = IF_ID_IR[15:0];

always@(posedge clk) begin //ID
    ID_EX_RS<= rs;
    ID_EX_RT<= rt;
    ID_EX_A<=rd1; //read data1
    ID_EX_B<=rd2; //read data2
    ID_EX_IMM<= extnd_imm;
    ID_EX_IR<= IF_ID_IR;
    ID_EX_NPC<= IF_ID_NPC;
    ID_EX_RD<= rd; //destination addr.

    if (ST) begin
        ID_EX_RegWrite<= 1'b0;
        ID_EX_MemWrite<= 1'b0;
        ID_EX_MemRead<= 1'b0;
        ID_EX_ALUsrc<= 1'b0;
        ID_EX_ALUctrl <= 3'b000;
        ID_EX_MemtoReg <= 1'b0;
        end
    else begin
        ID_EX_RegWrite<= EnRW;
        ID_EX_MemWrite<= MW;
```

```
        ID_EX_MemRead<= MR;
        ID_EX_ALUsrc<= ALUsrc;
        ID_EX_MemtoReg<= MReg;
        ID_EX_ALUctrl<= ALUctrl;
        end
    end

always@(posedge clk) begin //EX
    EX_MEM_ALUout<=ALUout;
    EX_MEM_B<= ID_EX_B;
    EX_MEM_IR<= ID_EX_IR;
    EX_MEM_RD<= ID_EX_RD;

    EX_MEM_RegWrite <= ID_EX_RegWrite;
    EX_MEM_MemtoReg<= ID_EX_MemtoReg;
    EX_MEM_MemRead<= ID_EX_MemRead;
    EX_MEM_MemWrite<= ID_EX_MemWrite;
    end

always@(posedge clk) begin //MEM
    MEM_WB_RD<= EX_MEM_RD;
    MEM_WB_ALUout<= EX_MEM_ALUout;
    MEM_WB_DATA<= data_rd;
    MEM_WB_RegWrite<= EX_MEM_RegWrite;
    MEM_WB_MemtoReg<= EX_MEM_MemtoReg;
    end
endmodule
```

TESTBENCH

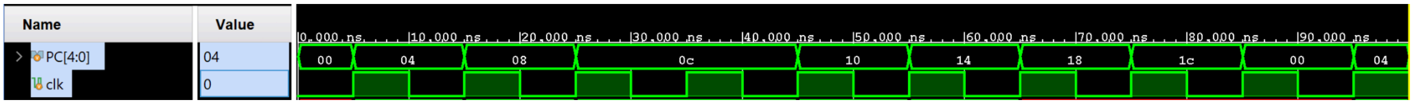
SIMULATION - Behavioral Simulation - Functional - sim_1 - pipeline_processor_tb

Tcl Console | Messages | Log

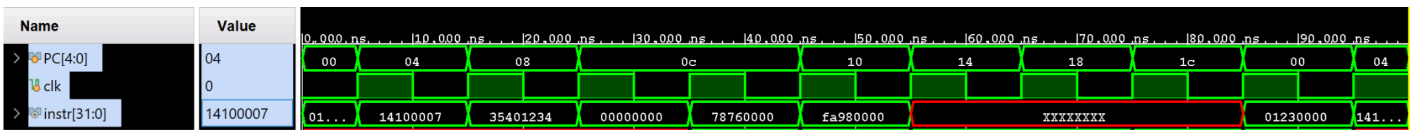
```
# run 1000ns
=== Starting Pipelined Processor Simulation ===
Initial Register Values:
reg2 = 00000009
reg3 = 00000010
reg6 = 000688ca
reg7 = 000964ea
reg9 = 0001212e
Time=0, PC=0, ST=0, FA=xx, FB=xx
Time=5000, PC=4, ST=0, FA=xx, FB=xx
Time=15000, PC=8, ST=0, FA=0, FB=0
Time=25000, PC=12, ST=1, FA=10, FB=0
Time=35000, PC=12, ST=0, FA=0, FB=0
Time=45000, PC=16, ST=0, FA=1, FB=0
Time=55000, PC=20, ST=0, FA=0, FB=0
Time=65000, PC=24, ST=0, FA=0, FB=10
Time=75000, PC=28, ST=0, FA=0, FB=0
Time=85000, PC=0, ST=0, FA=0, FB=0
Time=95000, PC=4, ST=0, FA=0, FB=0
Time=105000, PC=8, ST=0, FA=0, FB=0

=== Final Register Values ===
reg1 = 00000019 (ADD result)
reg4 = 12345678 (LW result)
reg5 = 12344444 (SUBI result)
reg8 = 000fecea (OR result)
reg10 = fff01211 (NOR result)

=== Verification ===
âœ" reg1: PASS - ADD operation correct
âœ" reg4: PASS - LW operation correct
âœ" reg5: PASS - SUBI operation correct
âœ" reg8: PASS - OR operation correct
âœ" reg10: PASS - NOR operation correct

=== Simulation Complete ===
$finish called at time : 110 ns : File "C:/Users/zahid/OneDrive/Desktop/PROJECT work/PROJECT work.srcs/sim_1/new/tb.v" Line 58
```

OUTPUT WAVEFORMS

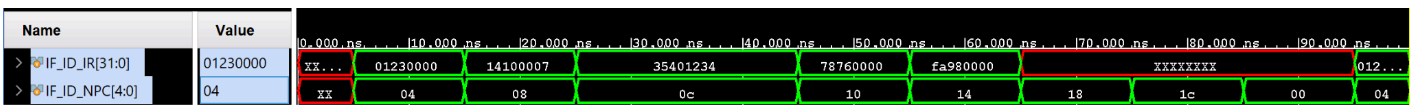
PC WAVEFORMS



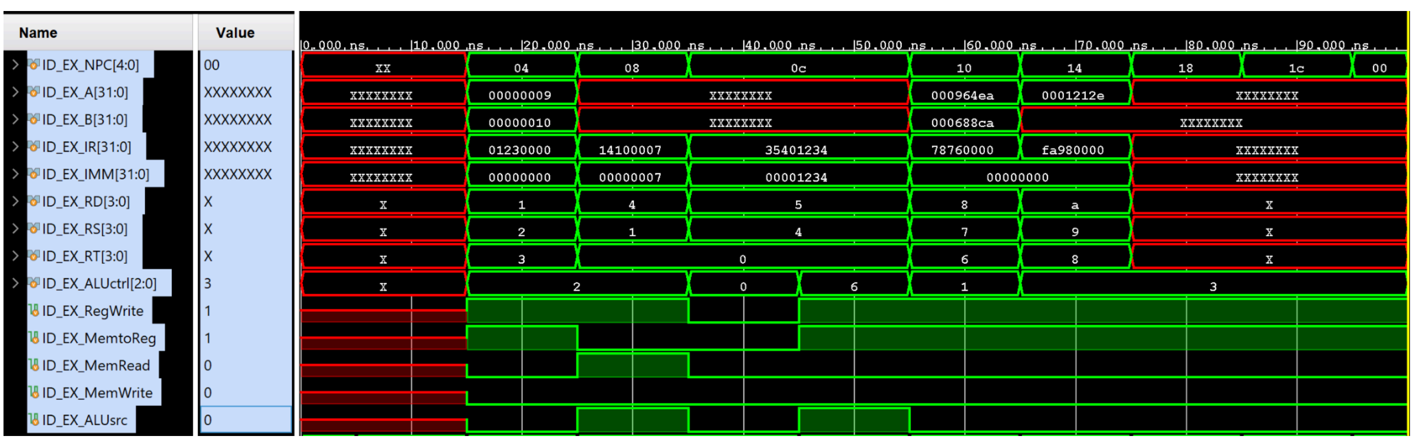
INSTRUCTION MEMORY WAVEFORM



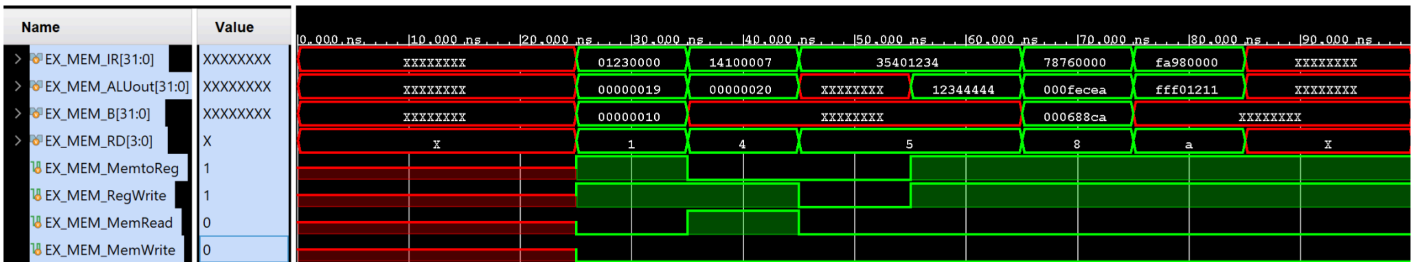
IF STAGE WAVEFORM



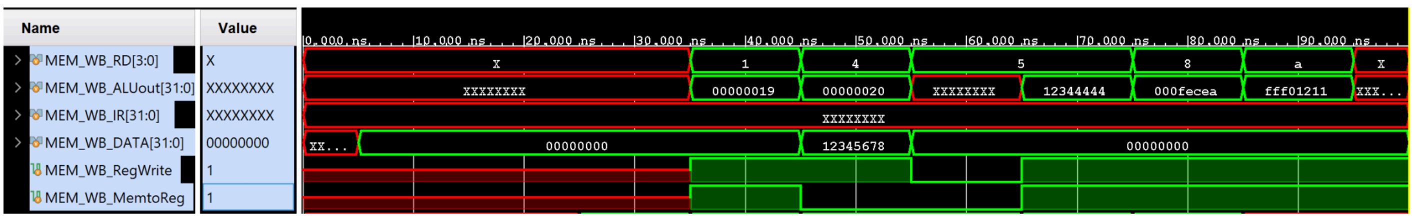
ID STAGE WAVEFORM



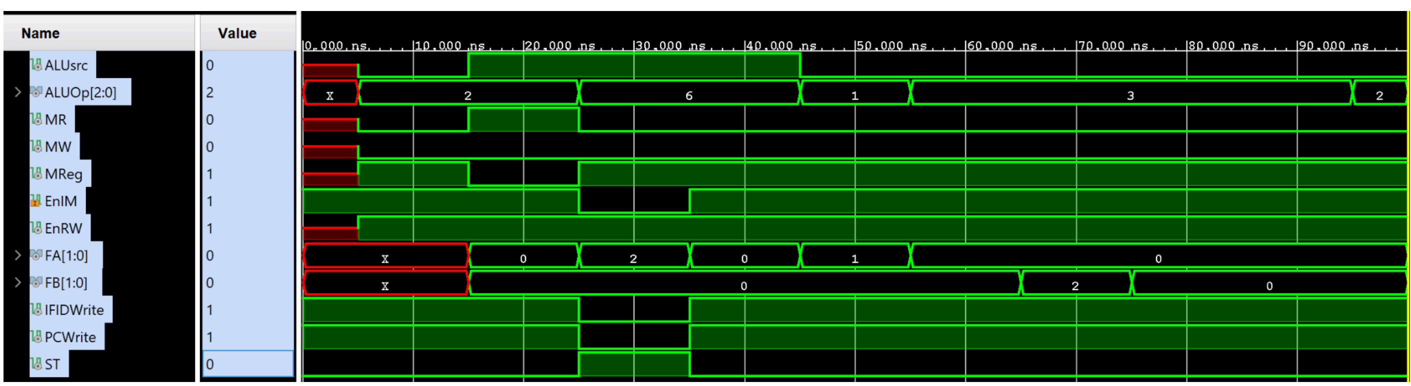
EX STAGE WAVEFORM



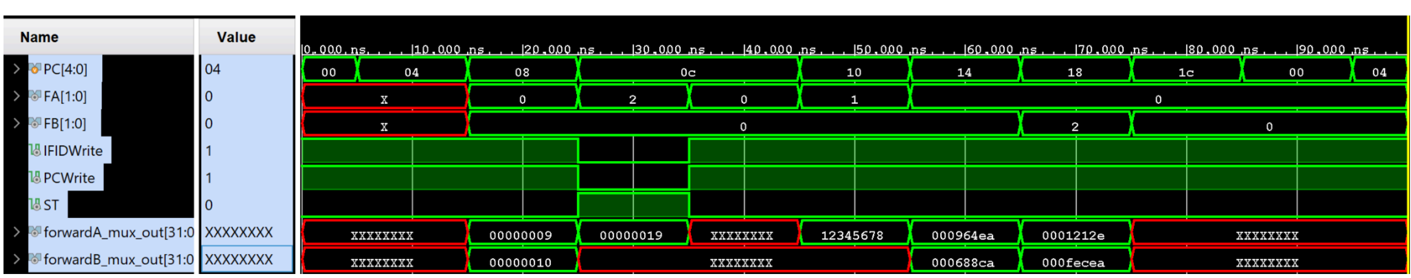
MEM STAGE WAVEFORM



CONTROL SIGNAL

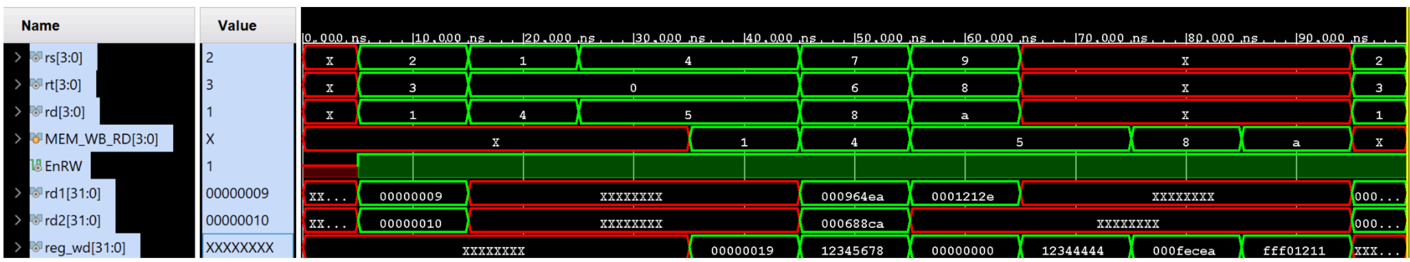


HAZARD DETECTION AND DATA FORWARDING



ALU

REGISTER FILE WAVEFORM



REGISTER FILE

Name	Value	Data Type
register[15:0][31:0]	XXXXXXXX,XXXXXXXX,XXXXXXXX,XXXXXXXX,XXXXXXXX,fff01211,0001212e,000fecea,000964ea,000688ca,12344444,12345678,00000010,00000009,00000019,XXXXXXXX	Array
[15][31:0]	XXXXXXXX	Array
[14][31:0]	XXXXXXXX	Array
[13][31:0]	XXXXXXXX	Array
[12][31:0]	XXXXXXXX	Array
[11][31:0]	XXXXXXXX	Array
[10][31:0]	fff01211	Array
[9][31:0]	0001212e	Array
[8][31:0]	000fecea	Array
[7][31:0]	000964ea	Array
[6][31:0]	000688ca	Array
[5][31:0]	12344444	Array
[4][31:0]	12345678	Array
[3][31:0]	00000010	Array
[2][31:0]	00000009	Array
[1][31:0]	00000019	Array
[0][31:0]	XXXXXXXX	Array

SCHEMATIC

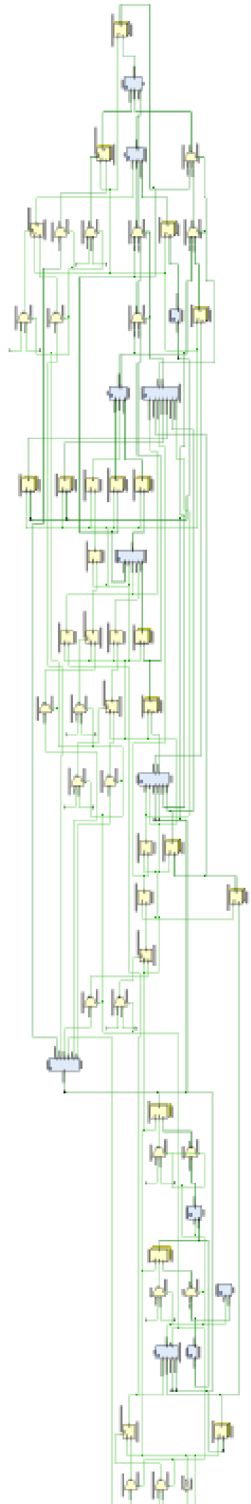


TABLE OF INSTRUCTIONS

Opcode (4-bit)	Instruction	ALUctrl (3-bit)	EnRW	ALUsrc	MReg (MemtoReg)	MR (MemRead)	MW (MemWrite)	Description
0000	ADD	010 (ADD)	1	0	1	0	0	R-type ALU add
0001	LW	010 (ADD)	1	1	0	1	0	Load Word
0010	SW	010 (ADD)	0	1	1	0	1	Store Word
0011	SUBI	110 (SUB)	1	1	1	0	0	I-type Subtract
0111	OR	001 (OR)	1	0	1	0	0	R-type OR
1111	NOR	011 (NOR)	1	0	1	0	0	R-type NOR

TABLE OF CONTROL SIGNALS

Signal	Description	Pipeline Stage	Typical Verilog Use
ALUSrc	Selects second ALU input (register or immediate)	EX (Execute)	assign ALU_input_B = (ALUSrc) ? imm : regB;
ALUOp[1:0]	Determines ALU operation (e.g., add, sub, and, or, slt)	EX (Execute)	Used in ALU control logic for op decoding
MR	Enables reading from data memory	MEM (Memory)	if (MR) data_out = memory[address];
MW	Enables writing to data memory	MEM (Memory)	if (MW) memory[address] = data_in;
MReg	Controls writing data from memory to register	WB (Write Back)	Used in MUX for reg_write_data = (MReg) ? mem_data : alu_result;
EnIM	Enables reading from instruction memory	IF (Fetch)	if (EnIM) instr = imem[PC];
EnRW	Enables writing to register file	WB (Write Back)	if (EnRW) register[rd] = write_data;
FA	Forwarding control for ALU input A	EX (Execute)	MUX select line for ALU_input_A forwarding
FB	Forwarding control for ALU input B	EX (Execute)	MUX select line for ALU_input_B forwarding

IFIDWrite	Controls whether IF/ID pipeline register updates (for stalling)	IF/ID Register	if (IFIDWrite) IF_ID <= new_data;
PCWrite	Controls whether PC is updated (used to stall instruction fetch)	IF (Fetch)	if (PCWrite) PC <= PC + 4;
ST	Control signal to zero out control signals during a stall (hazard detection)	Hazard/Control	control_signal = (ST) ? 0 : normal_value;

CONCLUSION

Through the course of this project, we successfully designed, implemented, and verified a 5-stage pipelined RISC processor capable of executing a specific set of instructions with efficient hazard handling. By carefully structuring the processor into Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back stages, we achieved parallel instruction processing, leading to improved throughput compared to a non-pipelined design.

We addressed key challenges such as data hazards by implementing a hazard detection unit to manage load-use hazards and a forwarding unit to minimize pipeline stalls. These units played a vital role in ensuring the correctness of instruction execution without significantly degrading performance. Furthermore, our modular design approach allowed each functional unit, such as the program counter, register file, ALU, and memory units, to be developed and tested individually before integration into the top-level processor structure.

The simulation results validated the functionality of our design. By using a comprehensive testbench, we were able to observe the behavior of the processor under various scenarios, ensuring that control signals, data forwarding, stalling, and pipeline register updates occurred correctly across clock cycles. The use of waveform visualization and active signal monitoring further helped in debugging and confirming correct data flow.

Overall, this project enhanced our understanding of pipelined processor architecture, control signal management, hazard detection, and real-world issues encountered in hardware design. It also reinforced the importance of systematic module design, timing control, and thorough verification. This project forms a strong foundation for more advanced topics like superscalar execution, out-of-order processing, and deeper pipeline structures.

CONTRIBUTION

We hereby declare that all the 4 members have contributed equally to this group project.