BITS PILANI
CS F372 COMPUTER ARCHITECTURE
PROJECT REPORT

GROUP NO:-28
QUESTION SET 3

BOJJA KARTHIK	2022AAPS0414H
SHREYAS DEVULAPALLI	2021B3A80604H
JATIN YADAV KAVETI	2022AAPS0404H
SUMANTH ABHINAV ARSHANAPALLY	2022A8PS0528H

Question:

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 LW reg1, 7(reg2)
0004 XORI reg3, reg1, AABB
0008 NAND reg5, reg4, reg3
0012 ADD reg8, reg7, reg6
0016 SUB reg10, reg9, reg5
Initialize the register file with the following data
reg2 = 5 reg4 = 03EC2
reg6 = 469BC reg7 = 964FF
reg9 = 11111
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)

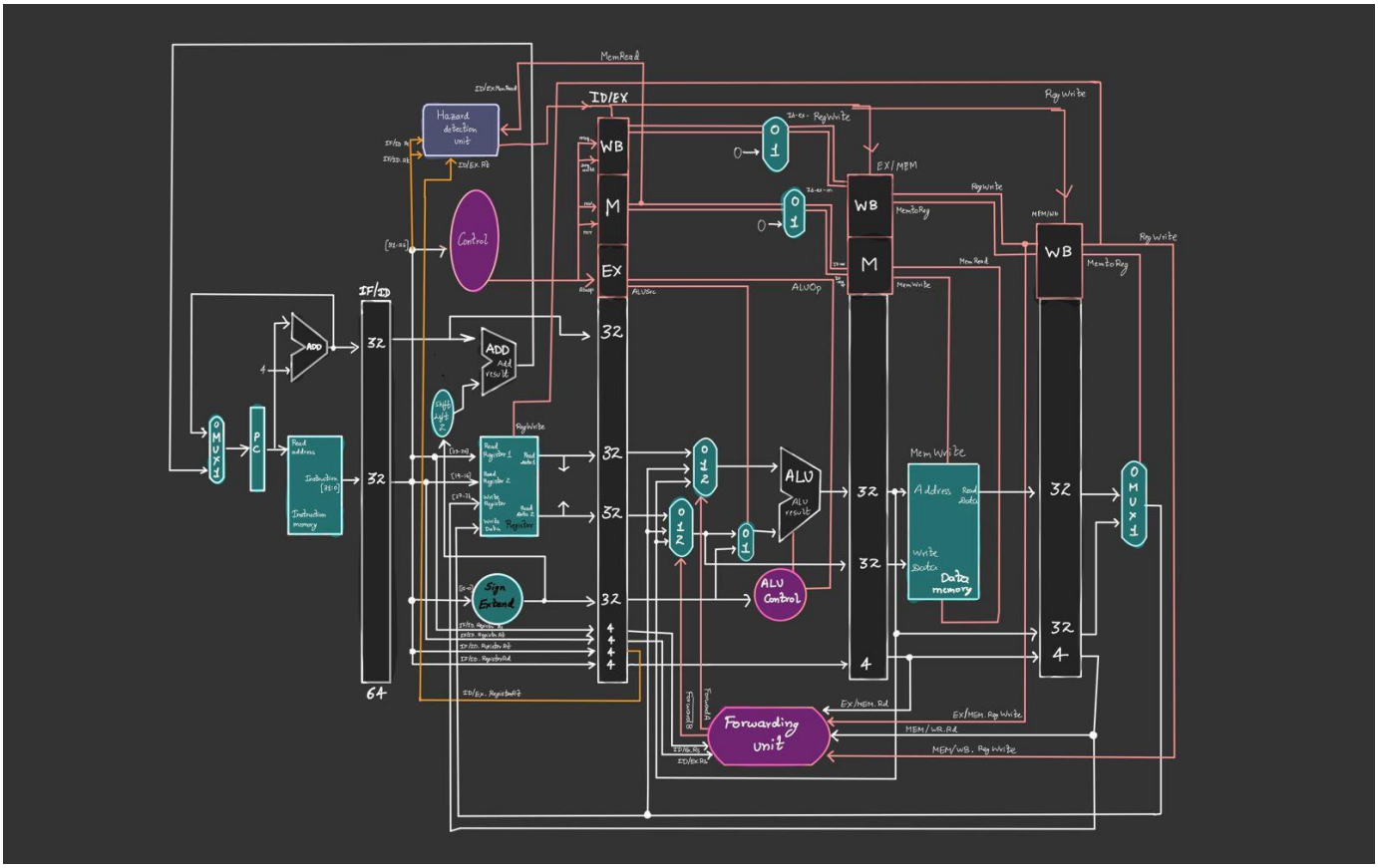
● PCWrite - Disable PC change (used in hazard detection circuit)
● ST - Control signal of mux which changes all control signals to zero (used in hazard detection circuit)
Initialize PC with all zeros. The instruction memory is of size 32 bytes, and the processor has 16 registers, numbered from reg0 to 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.

Contributions:

Equal Contribution from all team members.

Block Diagram:



Core Components:

1.Instruction Memory

The instruction memory module is responsible for storing all the machine instructions that the processor executes. During the Instruction Fetch (IF) stage of the pipeline, the processor sends the current value of the Program Counter (PC) to the instruction memory, and it returns the instruction located at that address. This instruction is then passed to the next stage of the pipeline for decoding and execution. The memory is typically read-only and pre-loaded with a program. Its simplicity allows for fast access, ensuring that the processor can fetch instructions quickly and efficiently during each clock cycle.

2.Control Unit

The control unit interprets the instruction received from the instruction memory and generates the appropriate control signals needed to guide the operation of various modules in the processor. Based on the opcode and instruction type, the control unit decides whether the operation is an ALU operation, memory access, or a control transfer like a branch. It produces signals such as RegWrite, MemRead, MemWrite, ALUSrc, and ALUOp, which control the data paths and functional units. This module plays a central role in ensuring that the processor behaves correctly and executes each instruction according to its intended semantics.

3.Hazard Detection Unit

The hazard detection unit is essential in pipelined processors where multiple instructions are in-flight simultaneously. It detects data hazards, particularly read-after-write (RAW) hazards, where an instruction depends on the result of a previous one that has not yet completed. If a hazard is detected, the unit stalls the pipeline by disabling writes to pipeline registers and holding the PC. This ensures that the instructions are executed in the correct order and that no incorrect or partial data is used. Without this unit, pipelined execution could result in incorrect outputs due to timing conflicts between dependent instructions.

4.Register File

The register file is a fast memory component consisting of a fixed number of registers used to store temporary data and results. Most RISC architectures have 32 general-purpose registers.

Each clock cycle, two registers can be read simultaneously while another can be written. In the decode stage, the register file reads the source operands for the current instruction using register addresses from the instruction fields. At the write-back stage, the result of an operation is written back into the destination register. This unit enables efficient data handling and is crucial for most ALU and memory instructions.

5.Forwarding Unit

The forwarding unit solves the problem of data hazards without stalling the pipeline. It checks if a required operand for an instruction in the execute stage is being written back by a later instruction still in the pipeline. If so, it forwards the result directly from the EX or MEM stage to the ALU inputs. This is known as bypassing. The forwarding unit compares destination and source register addresses between pipeline stages and enables the selection of the correct forwarded data. It significantly boosts processor performance by allowing continuous instruction flow without waiting for registers to update.

6.ALU

The Arithmetic Logic Unit (ALU) performs all arithmetic and logical computations required by the instructions. Common operations include addition, subtraction, bitwise AND/OR/XOR, and comparisons. It takes two inputs—usually from the register file or immediate values—and produces an output used for branching decisions or writing back into registers. The ALU is active during the execution stage of the pipeline. Its operation is determined by control signals generated by the ALU control unit. As the core computational engine of the processor, it handles most data processing tasks.

7.Data Memory

The data memory module handles load and store operations. It is accessed during the memory (MEM) stage of the pipeline when instructions like `lw` (load word) or `sw` (store word) are executed. Depending on the control signals, the processor either reads data from memory into a register or writes data from a register into memory. The address for the memory operation is typically generated by the ALU during the previous stage. This memory is separate from instruction memory and is often implemented as a simple RAM block. It plays a key role in supporting instructions that interact with data beyond the register file.

8.ALU Control Unit

The ALU control unit acts as a decoder that translates high-level instruction information into specific signals for the ALU. It receives inputs such as the ALUOp signal from the main control

unit and function bits from the instruction itself. These inputs allow it to determine the exact operation (add, subtract, AND, OR, etc.) that the ALU should perform. For instance, if the instruction is an R-type with a certain function code, the ALU control unit outputs the corresponding control code for that operation. This modular separation makes the ALU versatile and enables support for various instruction types using the same hardware.

9.Pipeline Registers (IF/ID, ID/EX, EX/MEM, MEM/WB)

Pipeline registers are used to pass data and control signals from one stage of the processor to the next in a pipelined architecture. Each register is placed between two stages: IF/ID between Fetch and Decode, ID/EX between Decode and Execute, EX/MEM between Execute and Memory, and MEM/WB between Memory and Write Back. These registers hold intermediate results, control bits, and addresses to ensure each instruction proceeds independently through the pipeline. They are clocked, meaning they update their contents at each clock cycle. These pipeline registers are key to maintaining the flow of multiple instructions simultaneously without interference or data loss.

10.PC to PC+4 Adder

This adder is a simple yet vital component of the instruction fetch stage. It calculates the address of the next instruction by adding 4 to the current PC value, assuming each instruction is 4 bytes long. This helps maintain the sequential flow of instructions. However, if a branch or jump occurs, the PC might be updated with a different address. The PC+4 value is also passed to the next pipeline stage and used in branch calculations. This adder ensures the processor continuously moves forward in the instruction stream unless explicitly directed otherwise by control flow instructions.

Final Code:

```
module risc_processor (
   input         clk,
   input         reset,
   output reg [31:0] pc,
   output wire [31:0] r1, r2, r3, r4, r5, r6, r7, r8, r9, r10,
   output reg       alusrc, mr, mw, mreg, enrw, ifidwrite, pcwrite, st,
```

```
    output wire     enim,
    output reg [1:0] fa, fb,
    output reg [3:0] aluop
);
    reg [31:0] if_id_instruction, if_id_pc;
    reg [31:0] id_ex_rd1, id_ex_rd2, id_ex_imm;
    reg [3:0]  id_ex_wn, id_ex_rn1, id_ex_rn2;
    reg        id_ex_alusrc, id_ex_mr, id_ex_mw, id_ex_mreg, id_ex_enrw;
    reg [3:0]  id_ex_aluop;
    reg [31:0] ex_mem_alu_result, ex_mem_rd2;
    reg [3:0]  ex_mem_wn;
    reg        ex_mem_mr, ex_mem_mw, ex_mem_mreg, ex_mem_enrw;
    reg [31:0] mem_wb_mem_data, mem_wb_alu_result;
    reg [3:0]  mem_wb_wn;
    reg        mem_wb_mreg, mem_wb_enrw;

    wire       alusrc_ctrl, mr_ctrl, mw_ctrl, mreg_ctrl, enrw_ctrl, enim_ctrl;
    wire [3:0]  aluop_ctrl;
    wire [1:0]  fa_ctrl, fb_ctrl;
    wire       ifidwrite_ctrl, pcwrite_ctrl, st_ctrl;

    reg [31:0] pc_next;
```

```
reg [7:0]  cycle_count;


wire [31:0] instruction;
instruction_memory imem (
   .clk(clk),
   .enim(enim_ctrl),
   .addr(pc),
   .instruction(instruction)
);


always @(posedge clk or posedge reset) begin
   if (reset) begin
      pc         <= 0;
      pc_next    <= 4;
      cycle_count <= 0;
   end else begin
      cycle_count <= cycle_count + 1;
      if (pcwrite_ctrl) begin
         pc     <= pc_next;
         pc_next <= pc_next + 4;
      end
   end
```

```
end

always @(posedge clk or posedge reset) begin
   if (reset) begin
      if_id_instruction <= 32'h0;
      if_id_pc          <= 32'h0;
   end else if (ifidwrite_ctrl) begin
      if_id_instruction <= instruction;
      if_id_pc          <= pc;
   end
end

wire [3:0] opcode = if_id_instruction[31:28];
wire [3:0] wn     = if_id_instruction[27:24];
wire [3:0] rn1    = if_id_instruction[23:20];
wire [3:0] rn2    = if_id_instruction[19:16];
wire [15:0] imm   = if_id_instruction[15:0];

wire [31:0] imm_se  = {{16{imm[15]}}, imm};
wire [31:0] imm_ze  = {16'h0000,      imm};
wire [31:0] imm_ext = (opcode == 4'b0001) ? imm_ze : imm_se;
```

```
wire [31:0] rd1, rd2;


control_unit ctrl_unit (
    .opcode(opcode),
    .alusrc(alusrc_ctrl),
    .mr(mr_ctrl),
    .mw(mw_ctrl),
    .mreg(mreg_ctrl),
    .enrw(enrw_ctrl),
    .enim(enim_ctrl),
    .aluop(aluop_ctrl)
);
assign enim = enim_ctrl;


hazard_detection hazard_unit (
    .id_ex_memread(id_ex_mr),
    .id_ex_wn(id_ex_wn),
    .if_id_rn1(rn1),
    .if_id_rn2(rn2),
    .ifidwrite(ifidwrite_ctrl),
    .pcwrite(pcwrite_ctrl),
    .st(st_ctrl)
```

```
);

register_file regfile (
    .clk(clk),
    .enrw(mem_wb_enrw),
    .rn1(rn1),
    .rn2(rn2),
    .wn(mem_wb_wn),
    .wd(mem_wb_mreg ? mem_wb_mem_data : mem_wb_alu_result),
    .rd1(rd1),
    .rd2(rd2),
    .r1(r1), .r2(r2), .r3(r3), .r4(r4), .r5(r5),
    .r6(r6), .r7(r7), .r8(r8), .r9(r9), .r10(r10)
);

always @(posedge clk or posedge reset) begin
    if (reset) begin
        id_ex_rd1    <= 32'h0;
        id_ex_rd2    <= 32'h0;
        id_ex_imm    <= 32'h0;
        id_ex_wn     <= 4'h0;
        id_ex_rn1    <= 4'h0;
```

```
    id_ex_rn2    <= 4'h0;
    id_ex_alusrc <= 1'b0;
    id_ex_aluop  <= 4'h0;
    id_ex_mr     <= 1'b0;
    id_ex_mw     <= 1'b0;
    id_ex_mreg   <= 1'b0;
    id_ex_enrw   <= 1'b0;
end else if (st_ctrl) begin
    id_ex_alusrc <= 1'b0;  id_ex_aluop <= 4'h0;
    id_ex_mr     <= 1'b0;  id_ex_mw    <= 1'b0;
    id_ex_mreg   <= 1'b0;  id_ex_enrw  <= 1'b0;
end else begin
    id_ex_rd1    <= rd1;
    id_ex_rd2    <= rd2;
    id_ex_imm    <= imm_ext;
    id_ex_wn     <= wn;
    id_ex_rn1    <= rn1;
    id_ex_rn2    <= rn2;
    id_ex_alusrc <= alusrc_ctrl;
    id_ex_aluop  <= aluop_ctrl;
    id_ex_mr     <= mr_ctrl;
    id_ex_mw     <= mw_ctrl;
```

```
        id_ex_mreg   <= mreg_ctrl;
        id_ex_enrw   <= enrw_ctrl;
      end
    end

    forwarding_unit forward_unit (
      .ex_mem_wn(ex_mem_wn),
      .mem_wb_wn(mem_wb_wn),
      .ex_mem_regwrite(ex_mem_enrw),
      .mem_wb_regwrite(mem_wb_enrw),
      .id_ex_rn1(id_ex_rn1),
      .id_ex_rn2(id_ex_rn2),
      .fa(fa_ctrl),
      .fb(fb_ctrl)
    );

    wire [31:0] alu_a = (fa_ctrl == 2'b10) ? ex_mem_alu_result :
                        (fa_ctrl == 2'b01) ? (mem_wb_mreg ? mem_wb_mem_data :
mem_wb_alu_result) :
                                             id_ex_rd1;
    wire [31:0] fwd_b = (fb_ctrl == 2'b10) ? ex_mem_alu_result :
                        (fb_ctrl == 2'b01) ? (mem_wb_mreg ? mem_wb_mem_data :
mem_wb_alu_result) :
```

```
                                id_ex_rd2;
wire [31:0] alu_b = id_ex_alusrc ? id_ex_imm : fwd_b;
wire [31:0] alu_result;
alu alu_unit (
   .op(id_ex_aluop),
   .a(alu_a),
   .b(alu_b),
   .result(alu_result)
);

always @(posedge clk or posedge reset) begin
   if (reset) begin
      ex_mem_alu_result <= 32'h0;
      ex_mem_rd2       <= 32'h0;
      ex_mem_wn        <= 4'h0;
      ex_mem_mr        <= 1'b0;
      ex_mem_mw        <= 1'b0;
      ex_mem_mreg      <= 1'b0;
      ex_mem_enrw      <= 1'b0;
   end else begin
      ex_mem_alu_result <= alu_result;
      ex_mem_rd2       <= fwd_b;
```

```
        ex_mem_wn       <= id_ex_wn;
        ex_mem_mr       <= id_ex_mr;
        ex_mem_mw       <= id_ex_mw;
        ex_mem_mreg     <= id_ex_mreg;
        ex_mem_enrw     <= id_ex_enrw;
    end
end

wire [31:0] mem_data;
data_memory dmem (
    .clk(clk),
    .mr(ex_mem_mr),
    .mw(ex_mem_mw),
    .addr(ex_mem_alu_result),
    .data_in(ex_mem_rd2),
    .data_out(mem_data)
);

always @(posedge clk or posedge reset) begin
    if (reset) begin
        mem_wb_mem_data  <= 32'h0;
        mem_wb_alu_result <= 32'h0;
```

```
        mem_wb_wn       <= 4'h0;
        mem_wb_mreg     <= 1'b0;
        mem_wb_enrw     <= 1'b0;
    end else begin
        mem_wb_mem_data  <= mem_data;
        mem_wb_alu_result <= ex_mem_alu_result;
        mem_wb_wn       <= ex_mem_wn;
        mem_wb_mreg     <= ex_mem_mreg;
        mem_wb_enrw     <= ex_mem_enrw;
    end
end

always @(posedge clk or posedge reset) begin
    if (reset) begin
        alusrc   <= 1'b0; mr   <= 1'b0; mw   <= 1'b0; mreg <= 1'b0; enrw <= 1'b0;
        ifidwrite <= 1'b1; pcwrite<= 1'b1; st   <= 1'b0; fa <= 2'b00; fb <= 2'b00; aluop <= 4'b0000;
    end else begin
        alusrc   <= alusrc_ctrl;
        mr       <= mr_ctrl;
        mw       <= mw_ctrl;
        mreg     <= mreg_ctrl;
        enrw     <= enrw_ctrl;
```

```
            ifidwrite <= ifidwrite_ctrl;
            pcwrite   <= pcwrite_ctrl;
            st        <= st_ctrl;
            fa        <= fa_ctrl;
            fb        <= fb_ctrl;
            aluop     <= aluop_ctrl;
        end
    end
endmodule

module control_unit (
    input  [3:0] opcode,
    output reg   alusrc, mr, mw, mreg, enrw, enim,
    output reg [3:0] aluop
);
    always @(*) begin
        case (opcode)
            4'b0000: begin
                alusrc=1; mr=1; mw=0; mreg=1; enrw=1; enim=1;
                aluop = 4'b0000;
            end
            4'b0001: begin
```

```
            alusrc=1; mr=0; mw=0; mreg=0; enrw=1; enim=1;
            aluop = 4'b0001;
        end
        4'b0011: begin
            alusrc=0; mr=0; mw=0; mreg=0; enrw=1; enim=1;
            aluop = 4'b0011;
        end
        4'b0111: begin
            alusrc=0; mr=0; mw=0; mreg=0; enrw=1; enim=1;
            aluop = 4'b0111;
        end
        4'b1111: begin
            alusrc=0; mr=0; mw=0; mreg=0; enrw=1; enim=1;
            aluop = 4'b1111;
        end
        default: begin
            alusrc=0; mr=0; mw=0; mreg=0; enrw=0; enim=1;
            aluop = 4'b0000;
        end
      endcase
   end
endmodule
```

```
module instruction_memory (
   input          clk,
   input          enim,
   input      [31:0] addr,
   output reg [31:0] instruction
);
   reg [7:0] mem [0:31];
   integer i;
   initial begin
      mem[0]=8'h01; mem[1]=8'h20; mem[2]=8'h00; mem[3]=8'h07;
      mem[4]=8'h13; mem[5]=8'h10; mem[6]=8'hAA; mem[7]=8'hBB;
      mem[8]=8'h35; mem[9]=8'h43; mem[10]=8'h00; mem[11]=8'h00;
      mem[12]=8'h78; mem[13]=8'h76; mem[14]=8'h00; mem[15]=8'h00;
      mem[16]=8'hFA; mem[17]=8'h95; mem[18]=8'h00; mem[19]=8'h00;
      for (i=20; i<32; i=i+1) mem[i]=8'h00;
   end

   always @(*) begin
      if (enim)
         instruction = {mem[addr],mem[addr+1],mem[addr+2],mem[addr+3]};
      else
```

```
        instruction = 32'h0;
    end
endmodule


module register_file (
    input          clk,
    input          enrw,
    input      [3:0] rn1, rn2, wn,
    input     [31:0] wd,
    output reg[31:0] rd1, rd2,
    output reg[31:0] r1, r2, r3, r4, r5,
                     r6, r7, r8, r9, r10
);
    reg [31:0] registers [0:15];
    initial begin
        registers[2]=32'h00000005;
        registers[4]=32'h00003EC2;
        registers[6]=32'h000469BC;
        registers[7]=32'h000964FF;
        registers[9]=32'h00011111;
    end
```

```
    always @(*) begin
        rd1=registers[rn1];
        rd2=registers[rn2];
        r1=registers[1];  r2=registers[2];  r3=registers[3];
        r4=registers[4];  r5=registers[5];  r6=registers[6];
        r7=registers[7];  r8=registers[8];  r9=registers[9];
        r10=registers[10];
    end

    always @(negedge clk) begin
        if (enrw && (wn!=0)) registers[wn]<=wd;
    end
endmodule

module alu (
    input  [3:0] op,
    input [31:0] a, b,
    output reg [31:0] result
);
    always @(*) begin
        case (op)
            4'b0000: result = a + b;
```

```
            4'b0111: result = a + b;
            4'b0001: result = a ^ b;
            4'b0011: result = ~(a & b);
            4'b1111: result = a - b;
            default: result = 32'h0;
        endcase
    end
endmodule

module data_memory (
    input          clk,
    input          mr, mw,
    input     [31:0] addr,
    input     [31:0] data_in,
    output reg[31:0] data_out
);
    reg [31:0] dmem [0:63];
    integer j;
    initial begin
        for (j=0; j<64; j=j+1) dmem[j]=32'h0;
        dmem[3]=32'hAABBCCDD;
    end
```

```
   always @(*) data_out = mr ? dmem[addr[31:2]] : 32'h0;
   always @(posedge clk) if (mw) dmem[addr[31:2]]<=data_in;
endmodule

module hazard_detection (
   input       id_ex_memread,
   input  [3:0] id_ex_wn,
   input  [3:0] if_id_rn1, if_id_rn2,
   output reg   ifidwrite, pcwrite, st
);
   always @(*) begin
      if (id_ex_memread && (id_ex_wn!=0) &&
        ((id_ex_wn==if_id_rn1)||(id_ex_wn==if_id_rn2))) begin
         ifidwrite=0; pcwrite=0; st=1;
      end else begin
         ifidwrite=1; pcwrite=1; st=0;
      end
   end
endmodule

module forwarding_unit (
```

```
    input  [3:0] ex_mem_wn, mem_wb_wn,
    input        ex_mem_regwrite, mem_wb_regwrite,
    input  [3:0] id_ex_rn1, id_ex_rn2,
    output reg [1:0] fa, fb
);
    always @(*) begin
        if (ex_mem_regwrite && (ex_mem_wn!=0) && (ex_mem_wn==id_ex_rn1))
            fa=2'b10;
        else if (mem_wb_regwrite && (mem_wb_wn!=0) && (mem_wb_wn==id_ex_rn1))
            fa=2'b01;
        else
            fa=2'b00;
        if (ex_mem_regwrite && (ex_mem_wn!=0) && (ex_mem_wn==id_ex_rn2))
            fb=2'b10;
        else if (mem_wb_regwrite && (mem_wb_wn!=0) && (mem_wb_wn==id_ex_rn2))
            fb=2'b01;
        else
            fb=2'b00;
    end
endmodule
```

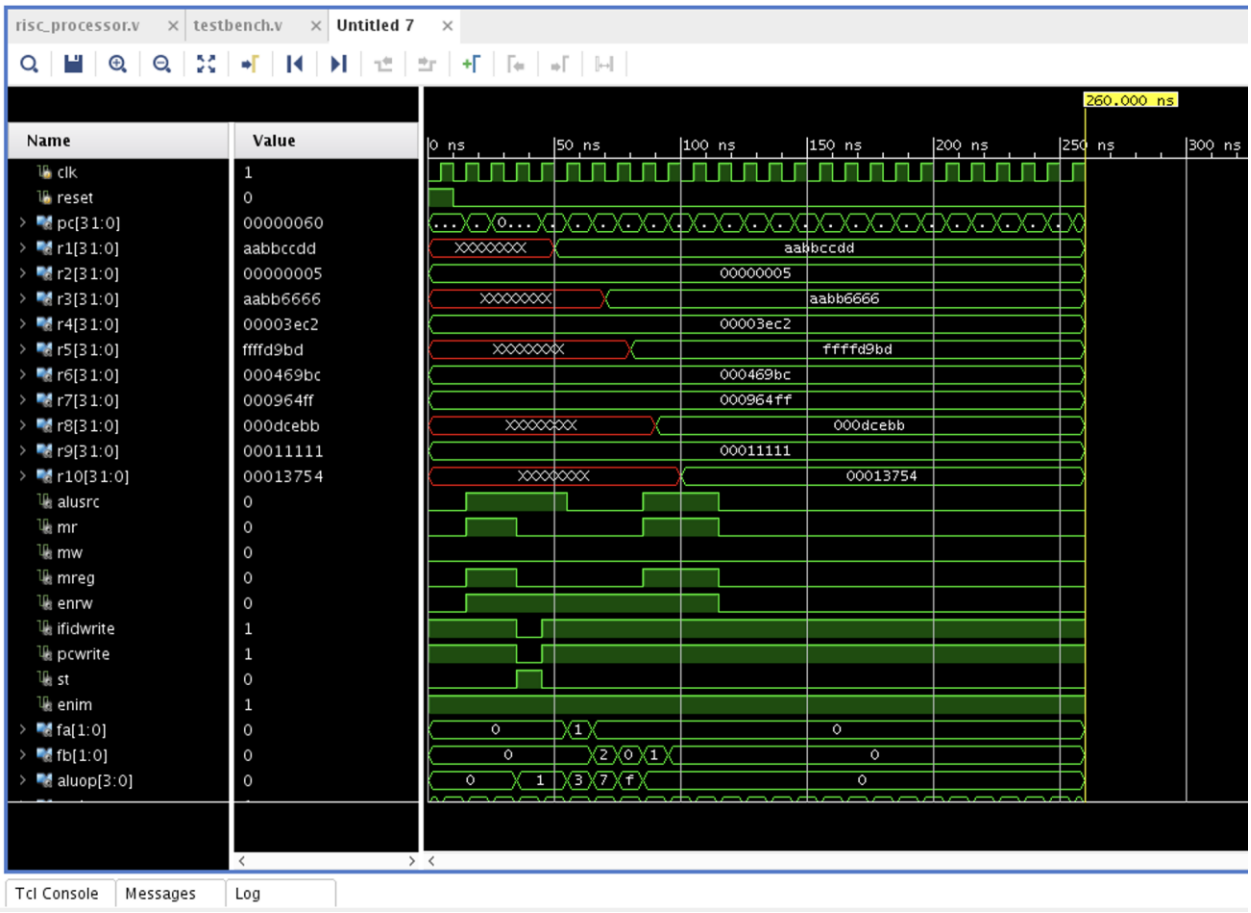
Testbench :

```
module testbench;
    reg         clk, reset;
    wire [31:0] pc;
    wire [31:0] r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;
    wire        alusrc, mr, mw, mreg, enrw, ifidwrite, pcwrite, st, enim;
    wire [1:0]  fa, fb;
    wire [3:0]  aluop;



    risc_processor uut (
        .clk        (clk),
        .reset      (reset),
        .pc         (pc),
        .r1         (r1),
        .r2         (r2),
        .r3         (r3),
        .r4         (r4),
        .r5         (r5),
        .r6         (r6),
        .r7         (r7),
        .r8         (r8),
        .r9         (r9),
        .r10        (r10),
        .alusrc     (alusrc),
```

```
    .mr         (mr),
    .mw         (mw),
    .mreg       (mreg),
    .enrw       (enrw),
    .ifidwrite  (ifidwrite),
    .pcwrite    (pcwrite),
    .st         (st),
    .enim       (enim),
    .fa         (fa),
    .fb         (fb),
    .aluop      (aluop)
);



initial begin
    clk = 0;
    forever #5 clk = ~clk;
end



initial begin
    reset = 1;
    #10 reset = 0;
    #250 $finish;
```

```
    end



    reg [7:0] cycle_counter = 0;
    always @(posedge clk) begin
        cycle_counter <= cycle_counter + 1;
        $display("Cycle %0d | PC = 0x%h | r1=0x%h r3=0x%h r5=0x%h r8=0x%h r10=0x%h",
                 cycle_counter, pc, r1, r3, r5, r8, r10);
        if (cycle_counter == 20) begin
            $display("** Completed 5 instructions by cycle 20 **");
        end
    end
endmodule
```

Output Waveform:


risc_processor.v
testbench.v
Untitled 7
Name
Value
clk
reset
pc[31:0]
r1[31:0]
r2[31:0]
r3[31:0]
r4[31:0]
r5[31:0]
r6[31:0]
r7[31:0]
r8[31:0]
r9[31:0]
r10[31:0]
alusrc
mr
mw
mreg
enrw
ifidwrite
pcwrite
st
enim
fa[1:0]
fb[1:0]
aluop[3:0]
00000060
aabbccdd
00000005
aabb6666
00003ec2
ffffd9bd
000469bc
000964ff
000dcebb
00011111
00013754
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
260.000 ns
Tcl Console
Messages
Log