CS F342 COMPUTER ARCHITECTURE

ASSIGNMENT 1

Implement a 5-stage pipelined processor in Verilog. This processor supports the instructions: load upper immediate (lui), or immediate (ori), set on less than (slt), register left shift (LSR), register right shift (RSR), jump (j). The processor should implement forwarding to resolve data hazards. The processor has Reset, CLK as inputs and no outputs. The processor has instruction fetch, decode, execution, memory, and writeback units. The processor also contains four pipelined registers IF/ID, ID/EX, EX/MEM, and MEM/WB. When reset is activated, the PC, IF/ID, ID/EX, EX/MEM, and MEM/WB registers are initialized to 0. The instruction memory and register file get loaded by predefined values.

The pipelined registers contain unknown values when the instruction unit starts fetching the first instruction. When the second instruction is being fetched in the IF unit, the IF/ID register will hold the instruction code for the first instruction. When the third instruction is being fetched by the IF unit, the IF/ID register contains the instruction code of the second instruction, the ID/RR register contains information related to the first instruction, and so on. (Assume a 32-bit PC. Also, Assume Address and Data size as 32-bit).

The instruction and its 32-bit instruction format are shown below:

lui destReg, imm (loads the highest 16 bits of the register rt with a constant (immediate value), and clears the lowest 16 bits to zeros. Opcode for lui is **001111**.)

(Note: This instruction does not use bits 25-21 (rs), preferably set them to 00000)

op	rs	rt	imm
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

ori destReg, sourceReg, imm ("or immediate" will perform a bitwise OR operation on the register rs and the immediate value, and store the result in register rt. Opcode for ori is **001101**.)

op	rs	rt	imm
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	16-bits (15-0)

slt destReg , sourceReg1, sourceReg2 (“set less than” sets rd value as 1 if the value in the rs register is less than the rt value, else sets it to 0. Opcode for slt is **000000** and func value is **101010**).

op	rs	rt	rd	shamt	func
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	5-bits (15-11)	5 bits (10-6)	6 bits (5-0)

LSR destinationReg, sourceReg 1, sourceReg 2 (Perform left shift operation on the register specified by register number in RS field by the value stored in the register specified by register number in RT in and save the result in the register specified by register number in RD field. OPCODE for LSR is 110010, SHAMT is 00000, and FUNCT is 000000).
[NOTE: You must solve this part, without using the SHAMT section of the instruction]

op	rs	rt	rd	shamt	funct
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	5-bits (15-11)	5-bits (10-6)	6-bits (5-0)

RSR destinationReg, sourceReg 1, sourceReg 2 (Perform right shift operation on the register specified by register number in RS field by the value stored in the register specified by register number in RT in and save the result in the register specified by register number in RD field. Opcode for RSR is 111011, SHAMT is 00000, and FUNCT is 000000.
[NOTE: You must solve this part, without using the SHAMT section of the instruction]

op	rs	rt	rd	shamt	funct
6 bits (31-26)	5-bits (25-21)	5-bits (20-16)	5-bits (15-11)	5-bits (10-6)	6-bits (5-0)

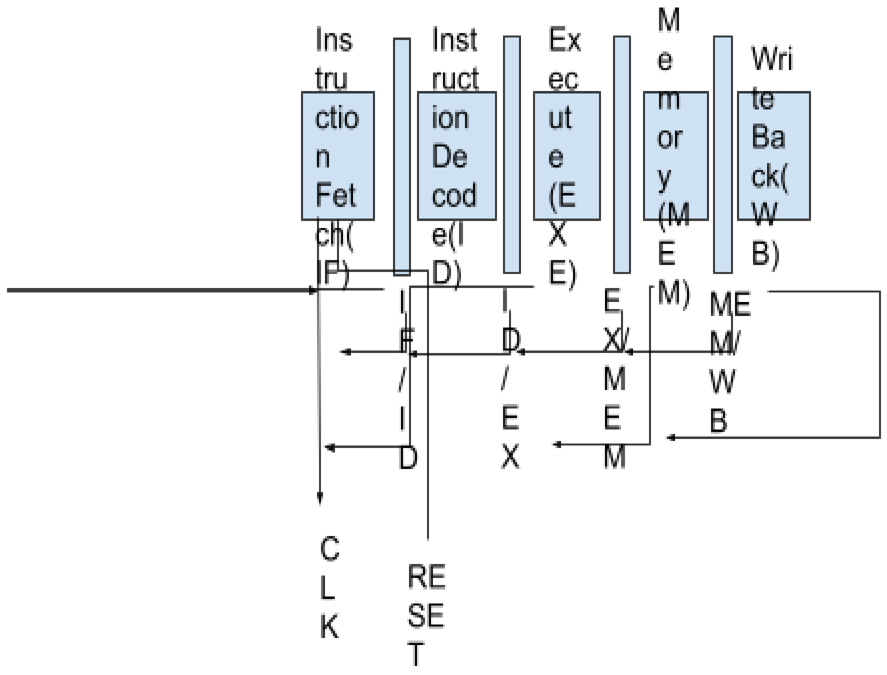
j target (jumps to an address generated by appending 00 to the right and 4 higher order bits of the program counter [31:28] to the left, of the 26 bit address field in the instruction. Opcode for j is **000010**). For clarity, **jump address(32-bit) = {PC[31:28], 26-bit immediate value, 00}**

op	address
6 bits (31-26)	26-bits (25-0)

Assume the register file contains 32 registers (R0-R31) each register can hold 32-bit data. On reset, PC and all register file registers should get initialized to 0. Ensure r0 is always zero. Each location in DMEM has 8-bit data. So, to store a 32-bit value, you need 4 locations in the DMEM, stored in big-endian format. Also ensure that on reset, the instruction memory gets initialized with the following instructions, starting at address 0:

LSR R1, R2, R8
LSR R3, R4, R9
SLT R5, R1, R3
ORI R6, R5, 200
RSR R6, R7, R10
j L1
LSR R7, R7, R8
L1: LUI R6, 128
The above code should run correctly on the processor implementation. Ensure that you handle the data hazards present if any.

A partial block-level representation of the 5-stage pipelined processor is shown below. **Please note that for register file implementation, write should be on the positive edge, and read should be on the negative edge of the clock.** Write operation depends on the control signal.



As part of the assignment, three files should be submitted in a zipped folder.

1. PDF version of this Document with all the Questions below answered with the file name **IDNO_NAME.pdf.**
2. Design Verilog Files for all the Sub-modules (instruction fetch, Register file, forwarding unit).

3. Design a Verilog file for the main processor.

   The name of the zipped folder should be in the format IDNO_NAME.zip

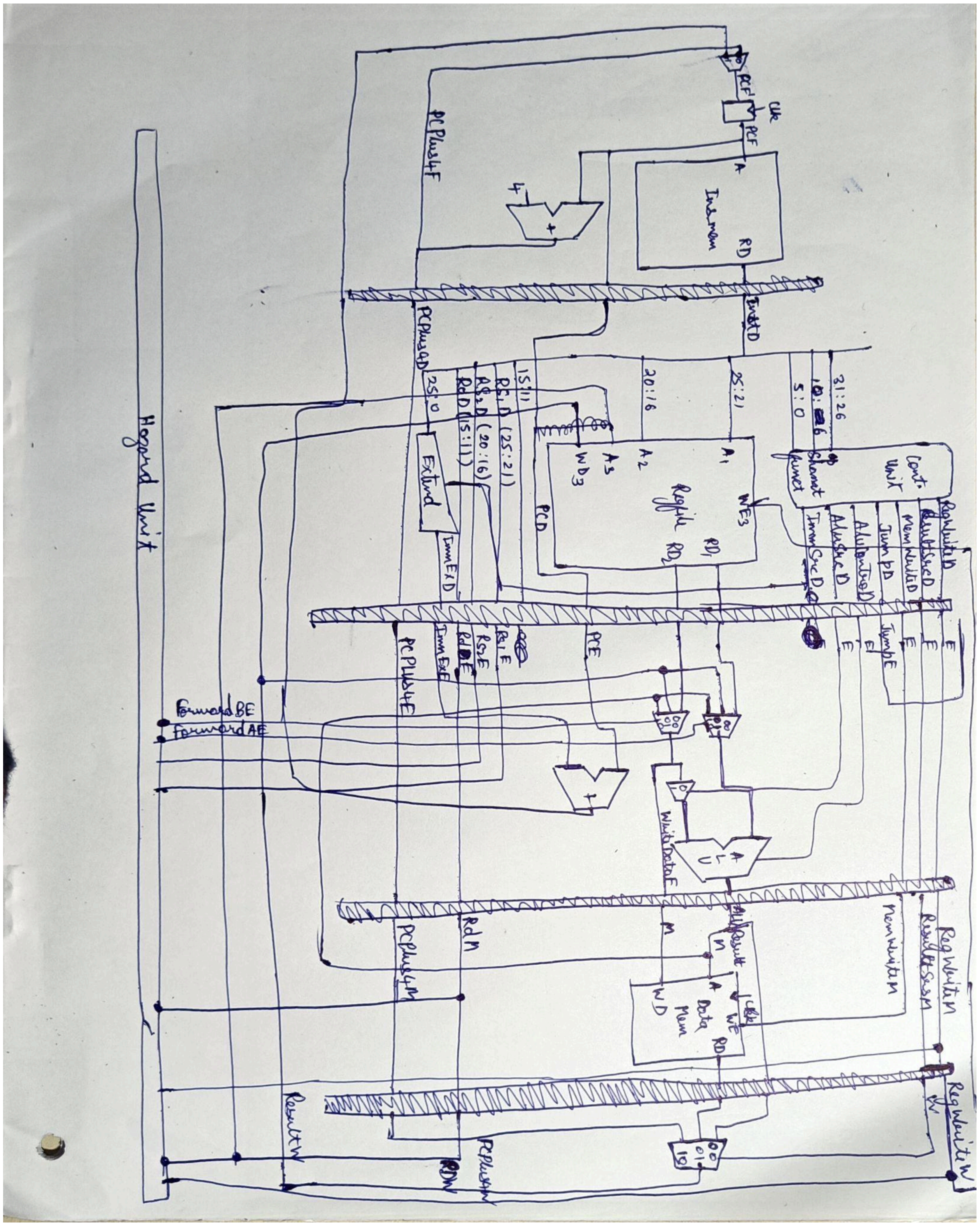
NAME: **ID No:**

Questions Related to Assignment

1. **Draw the complete Datapath and show control signals of the 5-stage pipelined processor. A sample Datapath for a 5-stage pipelined MIPS processor has been discussed in class. A ppt named Assignmenthelp.ppt contains this 5-stage processor and is shared with you. You can modify this according to your specification.**

Note: the adder in the execution unit is transposed to the decode unit to prevent the stall of when a jump instruction is decoded on the decode stage though it is not updated in the design.

Answer:


PCF'
clk
PCF
A
Instr mem
RD
PCPlus4F
+
4
InstrD
PCPlus4D
31:26
5:0
25:21
20:16
15:11
25:0
Opcode
Funct
Cont. Unit
RegWriteD
MemWriteD
JumpD
ALUControlD
ALUSrcD
ImmSrcD
A1
A2
A3
WD3
WE3
RD1
RD2
Regfile
Extend
ImmExtD
RtD
RsD
RdD
PCD
PCE
RtE
RsE
RdE
ImmExtE
PCPlus4E
JumpE
ForwardBE
ForwardAE
ALU
WriteDataE
ALUResultM
WriteDataM
Data Mem
WE
RD
WD
A
RdM
PCPlus4M
MemWriteM
RegWriteM
ResultSrcM
RegWriteW
ResultW
RdW
PCPlus4W
Hazard Unit

2. **Suppose we have to load a 32-bit constant value in one of the registers. (A single instruction only supports giving a 16-bit immediate value.) Implement the functionality to load a 32-bit constant value into the register, and then store it in memory. [Hint: Use the instructions discussed in this file]**
For example, store 1101010010101110 1010110000110100 in reg R1, then store R1 in a memory location
Answer

FOR THIS WE CAN USE LUI INSTRUCTION AND FIRST LOAD THE IMMEDIATE VALUE IN THE UPPER 16 BITS OF THE REGISTER R1 THEN SIMILARLY LOAD IN THE LOWER 16 BITS AND FINALLY STORING THE REGISTER VALUE IN MEMORY

LUI R1, R6, 1101010010101110
LUI R2, R6, 1010110000110100
RSR R2, R8, R2 **(GIVEN THE VALUE OF R8 = 16h)**
ORI R1, R1, 0h
ORI R1, R2, 0h

:

3. **List the control signals used and also the values of control signals for different instructions in a tabular format as follows:**

Answer:

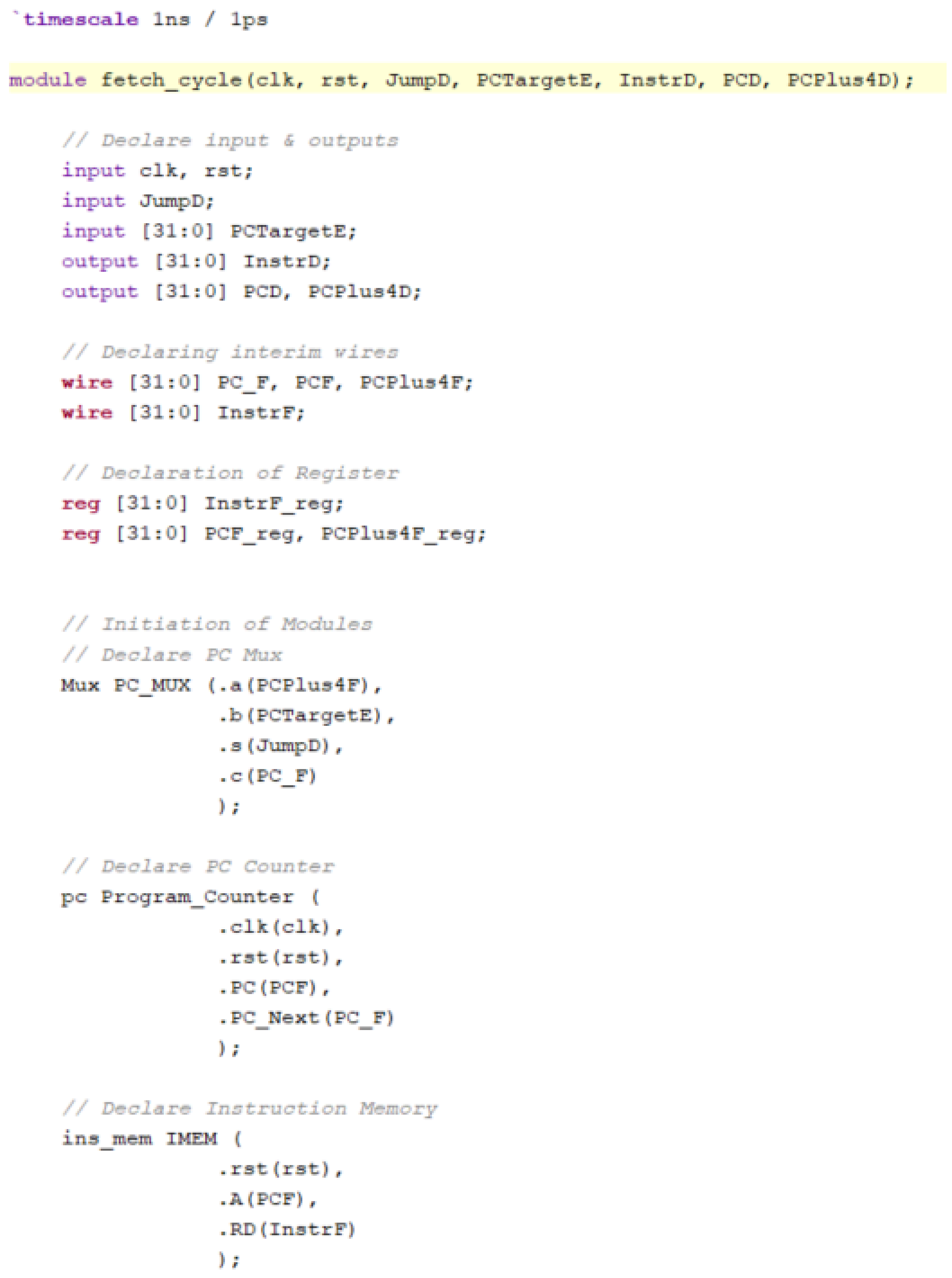
Instructions	Control Signals						
	Reg write	Result src	Mem write	jump	Alu control	Imm src	Alu src
lsr	1	0	0	0	010	11	0
rsr	1	0	0	0	011	11	0
slt	1	0	0	0	110	11	0
lui	1	0	0	0	000	11	1
ori	1	0	0	0	001	00	1
j	0	0	0	1	100	01	0

4. **In a program, there are 25% stores instructions, 1/x of which are immediately followed by an instruction that uses a result, requiring a stall. 10% are loads. 50% are R-type. 10% are branch, 1/y of which are taken. 5% are jumps. What is the average CPI of this program? If the number of instructions is 10^9, and the clock cycle is 100 ps, how much time does a MIPS single-cycle pipelined processor take to execute all instructions? Assume the processor always predicts branch not-taken.**
   Where x, y, z are related to last 3 digits of your ID No.
   If ID number: 20XXXXXXABCG, then x = (A % 8) + 1, y = ((B + 2) % 8) + 1, and z= ((C + 3) % 8) + 1.

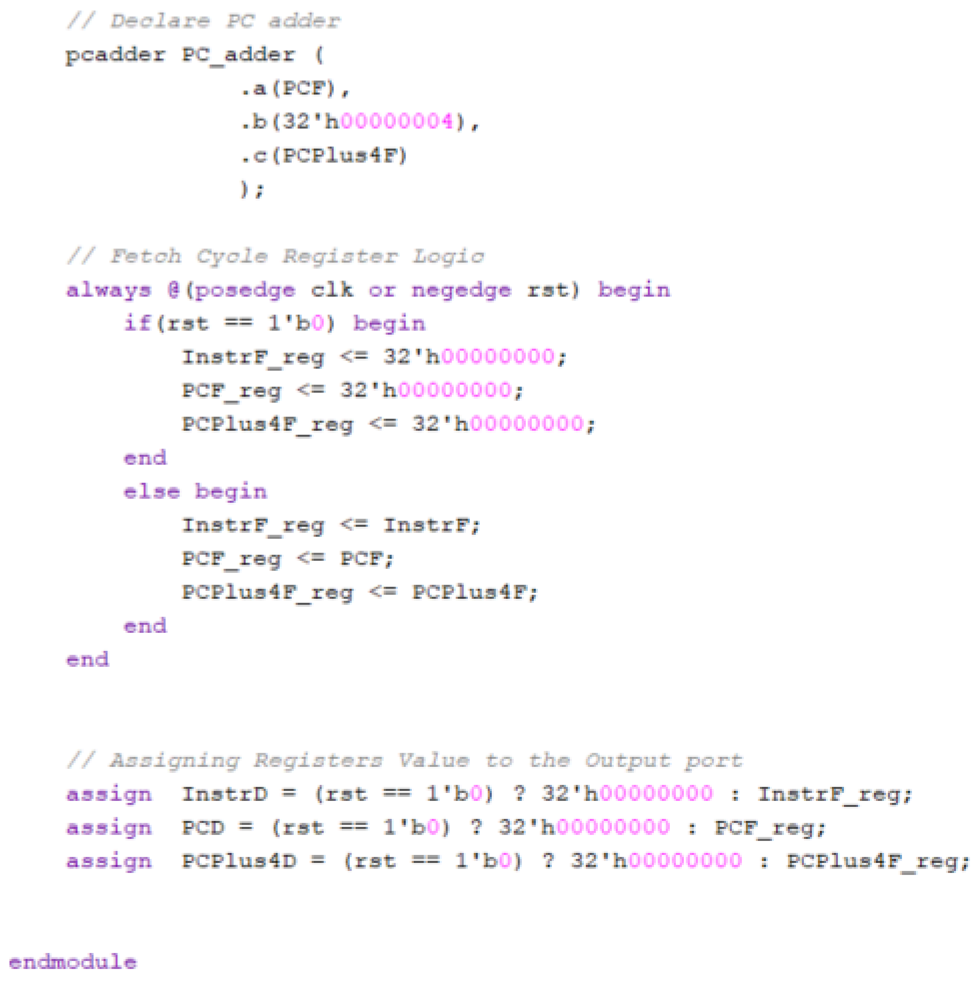
   Answer: 1.3176 (ASSUMING ONLY TAKEN BRANCHES TAKE TWO PIPELINE STALLS)

5. **Implement the Instruction Fetch block. Copy the <u>image</u> of Verilog code of the Instruction fetch block here**

Answer:



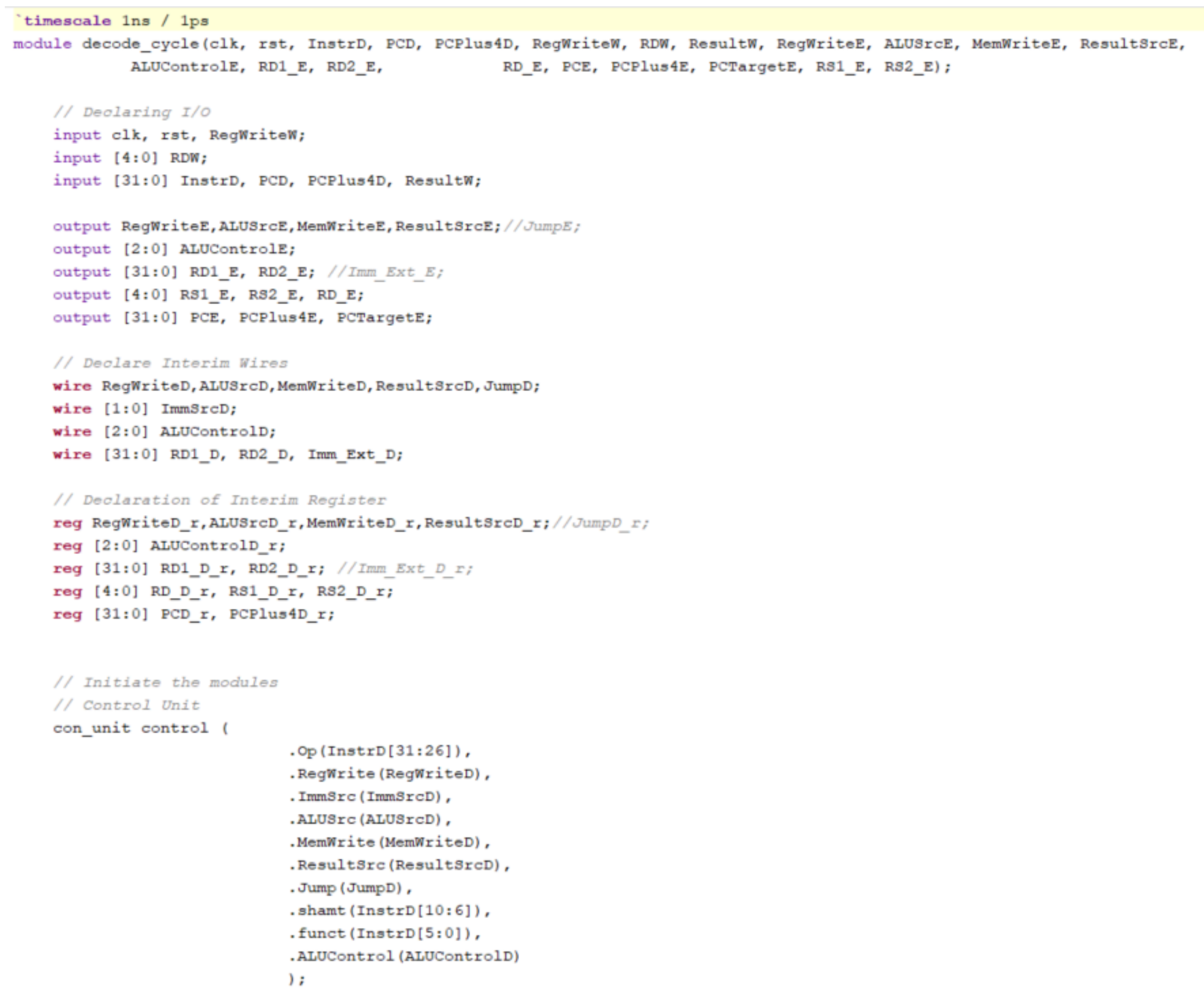
```verilog
`timescale 1ns / 1ps

module fetch_cycle(clk, rst, JumpD, PCTargetE, InstrD, PCD, PCPlus4D);

    // Declare input & outputs
    input clk, rst;
    input JumpD;
    input [31:0] PCTargetE;
    output [31:0] InstrD;
    output [31:0] PCD, PCPlus4D;

    // Declaring interim wires
    wire [31:0] PC_F, PCF, PCPlus4F;
    wire [31:0] InstrF;

    // Declaration of Register
    reg [31:0] InstrF_reg;
    reg [31:0] PCF_reg, PCPlus4F_reg;


    // Initiation of Modules
    // Declare PC Mux
    Mux PC_MUX (.a(PCPlus4F),
                .b(PCTargetE),
                .s(JumpD),
                .c(PC_F)
                );

    // Declare PC Counter
    pc Program_Counter (
                .clk(clk),
                .rst(rst),
                .PC(PCF),
                .PC_Next(PC_F)
                );

    // Declare Instruction Memory
    ins_mem IMEM (
                .rst(rst),
                .A(PCF),
                .RD(InstrF)
                );
```



```verilog
    // Declare PC adder
    pcadder PC_adder (
                .a(PCF),
                .b(32'h00000004),
                .c(PCPlus4F)
                );

    // Fetch Cycle Register Logic
    always @(posedge clk or negedge rst) begin
        if(rst == 1'b0) begin
            InstrF_reg <= 32'h00000000;
            PCF_reg <= 32'h00000000;
            PCPlus4F_reg <= 32'h00000000;
        end
        else begin
            InstrF_reg <= InstrF;
            PCF_reg <= PCF;
            PCPlus4F_reg <= PCPlus4F;
        end
    end


    // Assigning Registers Value to the Output port
    assign  InstrD = (rst == 1'b0) ? 32'h00000000 : InstrF_reg;
    assign  PCD = (rst == 1'b0) ? 32'h00000000 : PCF_reg;
    assign  PCPlus4D = (rst == 1'b0) ? 32'h00000000 : PCPlus4F_reg;


endmodule
```

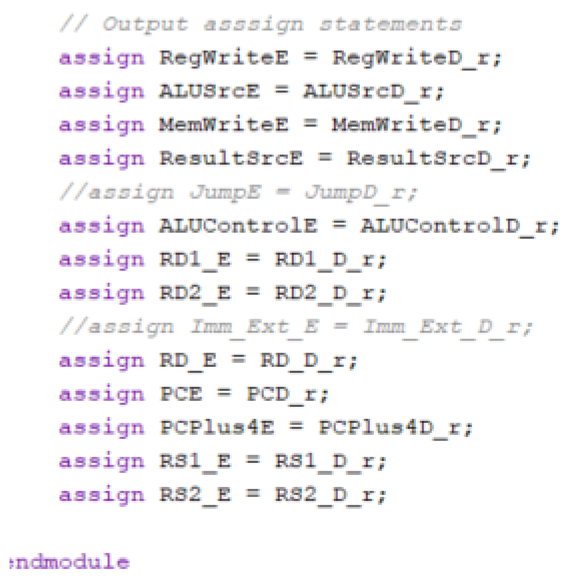
6. **Implement the Instruction Decode block. Copy the <u>image</u> of Verilog code of the Instruction decode block here**

   Answer:



```verilog
`timescale 1ns / 1ps
module decode_cycle(clk, rst, InstrD, PCD, PCPlus4D, RegWriteW, RDW, ResultW, RegWriteE, ALUSrcE, MemWriteE, ResultSrcE,
            ALUControlE, RD1_E, RD2_E,            RD_E, PCE, PCPlus4E, PCTargetE, RS1_E, RS2_E);

    // Declaring I/O
    input clk, rst, RegWriteW;
    input [4:0] RDW;
    input [31:0] InstrD, PCD, PCPlus4D, ResultW;

    output RegWriteE,ALUSrcE,MemWriteE,ResultSrcE;//JumpE;
    output [2:0] ALUControlE;
    output [31:0] RD1_E, RD2_E; //Imm_Ext_E;
    output [4:0] RS1_E, RS2_E, RD_E;
    output [31:0] PCE, PCPlus4E, PCTargetE;

    // Declare Interim Wires
    wire RegWriteD,ALUSrcD,MemWriteD,ResultSrcD,JumpD;
    wire [1:0] ImmSrcD;
    wire [2:0] ALUControlD;
    wire [31:0] RD1_D, RD2_D, Imm_Ext_D;

    // Declaration of Interim Register
    reg RegWriteD_r,ALUSrcD_r,MemWriteD_r,ResultSrcD_r;//JumpD_r;
    reg [2:0] ALUControlD_r;
    reg [31:0] RD1_D_r, RD2_D_r; //Imm_Ext_D_r;
    reg [4:0] RD_D_r, RS1_D_r, RS2_D_r;
    reg [31:0] PCD_r, PCPlus4D_r;


    // Initiate the modules
    // Control Unit
    con_unit control (
                            .Op(InstrD[31:26]),
                            .RegWrite(RegWriteD),
                            .ImmSrc(ImmSrcD),
                            .ALUSrc(ALUSrcD),
                            .MemWrite(MemWriteD),
                            .ResultSrc(ResultSrcD),
                            .Jump(JumpD),
                            .shamt(InstrD[10:6]),
                            .funct(InstrD[5:0]),
                            .ALUControl(ALUControlD)
                            );
    // Register File
    reg_fl rf (
                        .clk(clk),
                        .rst(rst),
                        .WE3(RegWriteW),
                        .WD3(ResultW),
                        .A1(InstrD[25:21]),
                        .A2(InstrD[20:16]),
                        .A3(RDW),
                        .RD1(RD1_D),
                        .RD2(RD2_D)
                        );

    // Sign Extension
    extend extension (
                        .In(InstrD[31:0]),
                        .Imm_Ext(Imm_Ext_D),
                        .ImmSrc(ImmSrcD)
                        );

  // Adder
pcadder branch_adder (
    .a(PCD),
    .b(Imm_Ext_D),
    .c(PCTargetE)
    );
```

```verilog
// Declaring Register Logic
always @(posedge clk or negedge rst) begin
    if(rst == 1'b0) begin
        RegWriteD_r <= 1'b0;
        ALUSrcD_r <= 1'b0;
        MemWriteD_r <= 1'b0;
        ResultSrcD_r <= 1'b0;
        //JumpD_r <= 1'b0;
        ALUControlD_r <= 3'b000;
        RD1_D_r <= 32'h00000000;
        RD2_D_r <= 32'h00000000;
        //Imm_Ext_D_r <= 32'h00000000;
        RD_D_r <= 5'h00;
        PCD_r <= 32'h00000000;
        PCPlus4D_r <= 32'h00000000;
        RS1_D_r <= 5'h00;
        RS2_D_r <= 5'h00;
    end
    else begin
        RegWriteD_r <= RegWriteD;
        ALUSrcD_r <= ALUSrcD;
        MemWriteD_r <= MemWriteD;
        ResultSrcD_r <= ResultSrcD;
        //JumpD_r <= JumpD;
        ALUControlD_r <= ALUControlD;
        RD1_D_r <= RD1_D;
        RD2_D_r <= RD2_D;
        //Imm_Ext_D_r <= Imm_Ext_D;
        RD_D_r <= InstrD[15:11];
        PCD_r <= PCD;
        PCPlus4D_r <= PCPlus4D;
        RS1_D_r <= InstrD[25:21];
        RS2_D_r <= InstrD[20:16];
    end
end
```



```
    // Output asssign statements
    assign RegWriteE = RegWriteD_r;
    assign ALUSrcE = ALUSrcD_r;
    assign MemWriteE = MemWriteD_r;
    assign ResultSrcE = ResultSrcD_r;
    //assign JumpE = JumpD_r;
    assign ALUControlE = ALUControlD_r;
    assign RD1_E = RD1_D_r;
    assign RD2_E = RD2_D_r;
    //assign Imm_Ext_E = Imm_Ext_D_r;
    assign RD_E = RD_D_r;
    assign PCE = PCD_r;
    assign PCPlus4E = PCPlus4D_r;
    assign RS1_E = RS1_D_r;
    assign RS2_E = RS2_D_r;

ndmodule
```

7. **What is the data hazard in the given instruction snippet? Determine the condition that can detect the data hazard in the given instruction snippet.**

Answer:

LSR R1, R2, R8

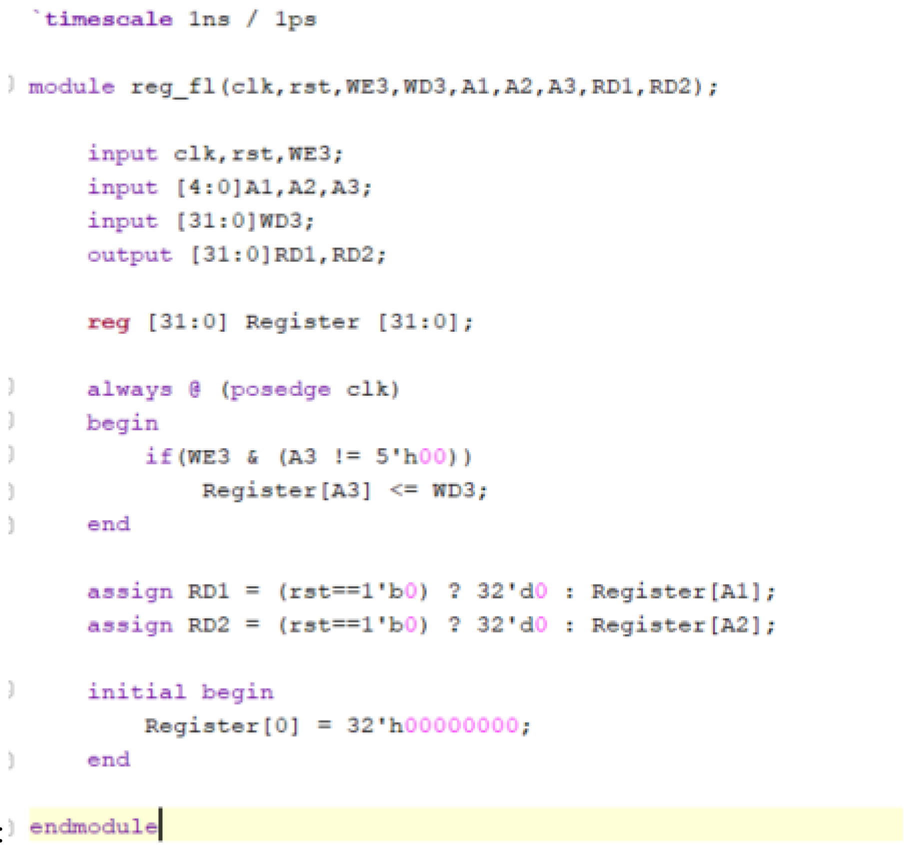
SLT R5, R1, R3
R1: data dependency. Require forwarding from memory stage

LSR R3, R4, R9
SLT R5, R1, R3
R3: data dependency. Require forwarding from execution stage

SLT R5, R1, R3
ORI R6, R5, 200
R5: data dependency. Require forwarding from execution stage

RSR R6, R7, R10
L1: LUI R6, 128
R6: data dependency. Require forwarding from memory stage as unconditional jump takes no flushes as per the designed architecture.

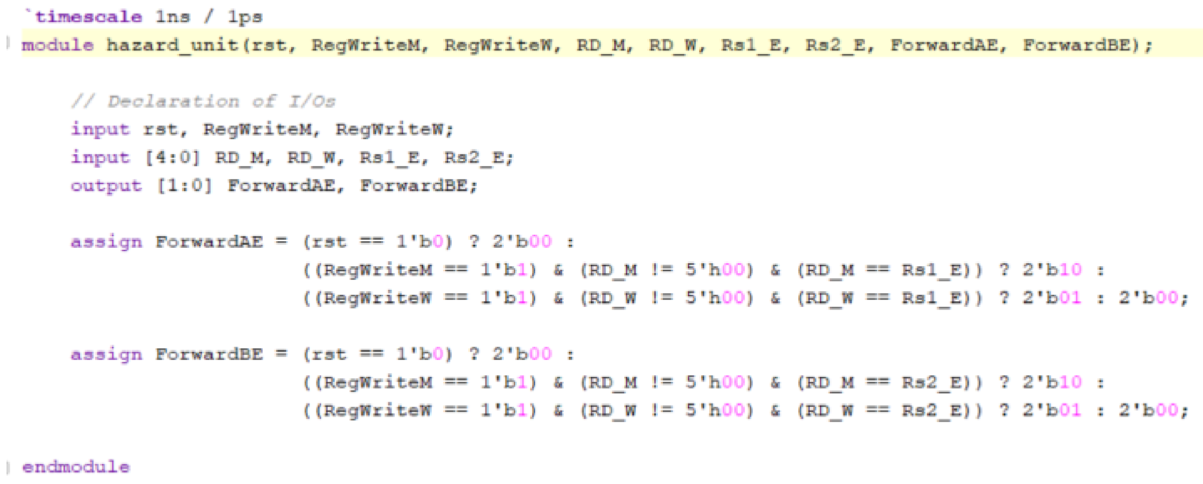
8. **Implement the Register File and copy the image of Verilog code of Register file unit here.**

Answer: 

```verilog
`timescale 1ns / 1ps

module reg_fl(clk,rst,WE3,WD3,A1,A2,A3,RD1,RD2);

    input clk,rst,WE3;
    input [4:0]A1,A2,A3;
    input [31:0]WD3;
    output [31:0]RD1,RD2;

    reg [31:0] Register [31:0];

    always @ (posedge clk)
    begin
        if(WE3 & (A3 != 5'h00))
            Register[A3] <= WD3;
    end

    assign RD1 = (rst==1'b0) ? 32'd0 : Register[A1];
    assign RD2 = (rst==1'b0) ? 32'd0 : Register[A2];

    initial begin
        Register[0] = 32'h00000000;
    end

endmodule
```

9. **Implement the forwarding unit and copy the image of Verilog code of forwarding unit here.**

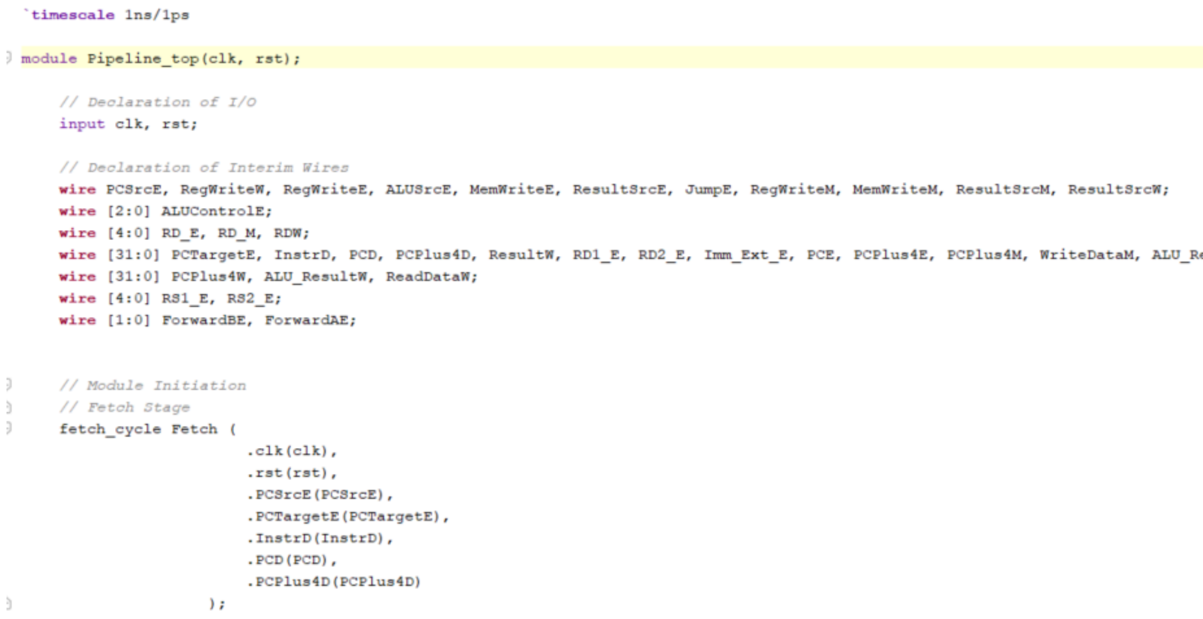
Answer:



```
`timescale 1ns / 1ps
module hazard_unit(rst, RegWriteM, RegWriteW, RD_M, RD_W, Rs1_E, Rs2_E, ForwardAE, ForwardBE);

    // Declaration of I/Os
    input rst, RegWriteM, RegWriteW;
    input [4:0] RD_M, RD_W, Rs1_E, Rs2_E;
    output [1:0] ForwardAE, ForwardBE;

    assign ForwardAE = (rst == 1'b0) ? 2'b00 :
                       ((RegWriteM == 1'b1) & (RD_M != 5'h00) & (RD_M == Rs1_E)) ? 2'b10 :
                       ((RegWriteW == 1'b1) & (RD_W != 5'h00) & (RD_W == Rs1_E)) ? 2'b01 : 2'b00;

    assign ForwardBE = (rst == 1'b0) ? 2'b00 :
                       ((RegWriteM == 1'b1) & (RD_M != 5'h00) & (RD_M == Rs2_E)) ? 2'b10 :
                       ((RegWriteW == 1'b1) & (RD_W != 5'h00) & (RD_W == Rs2_E)) ? 2'b01 : 2'b00;

endmodule
```

10. **Implement a complete processor in Verilog (using all the Datapath blocks). Copy the <u>image</u> of Verilog code of the processor here. (Use comments to describe your Verilog implementation)**

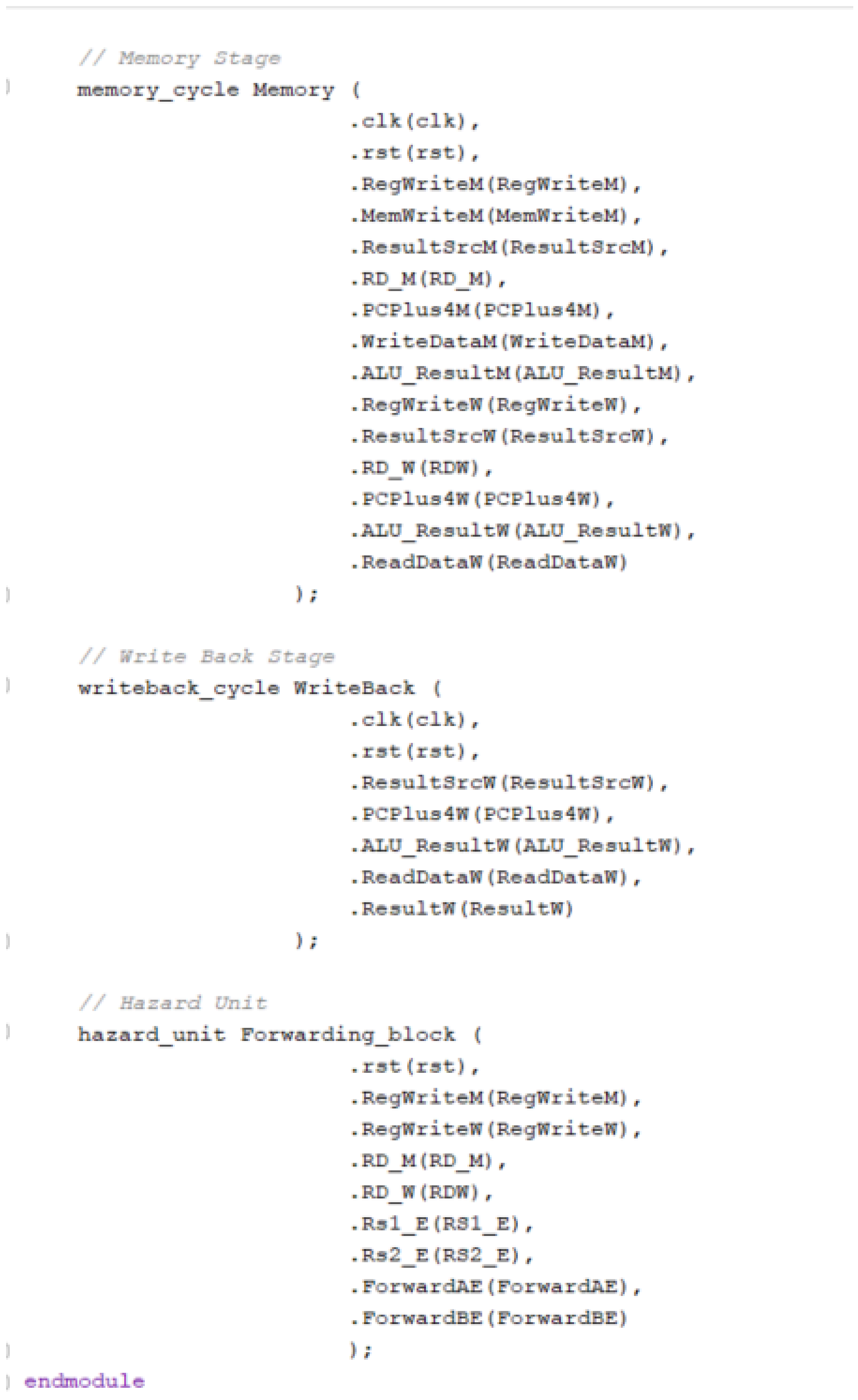
Answer:



```
`timescale 1ns/1ps

module Pipeline_top(clk, rst);

    // Declaration of I/O
    input clk, rst;

    // Declaration of Interim Wires
    wire PCSrcE, RegWriteW, RegWriteE, ALUSrcE, MemWriteE, ResultSrcE, JumpE, RegWriteM, MemWriteM, ResultSrcM, ResultSrcW;
    wire [2:0] ALUControlE;
    wire [4:0] RD_E, RD_M, RDW;
    wire [31:0] PCTargetE, InstrD, PCD, PCPlus4D, ResultW, RD1_E, RD2_E, Imm_Ext_E, PCE, PCPlus4E, PCPlus4M, WriteDataM, ALU_Re
    wire [31:0] PCPlus4W, ALU_ResultW, ReadDataW;
    wire [4:0] RS1_E, RS2_E;
    wire [1:0] ForwardBE, ForwardAE;


    // Module Initiation
    // Fetch Stage
    fetch_cycle Fetch (
                        .clk(clk),
                        .rst(rst),
                        .PCSrcE(PCSrcE),
                        .PCTargetE(PCTargetE),
                        .InstrD(InstrD),
                        .PCD(PCD),
                        .PCPlus4D(PCPlus4D)
                    );
```

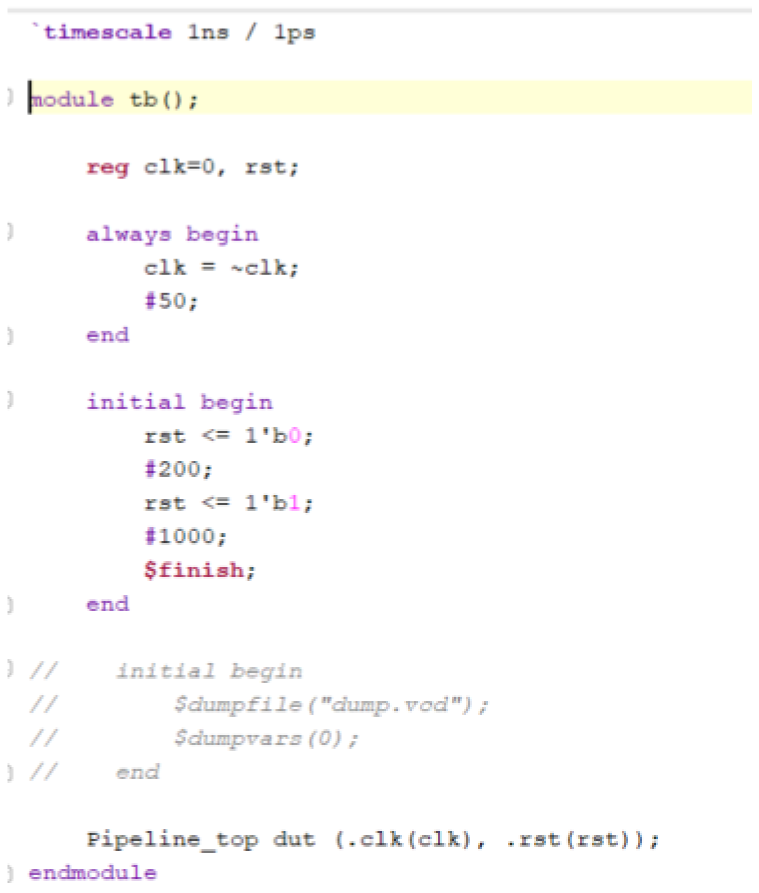
```
// Decode Stage
decode_cycle Decode (
                    .clk(clk),
                    .rst(rst),
                    .InstrD(InstrD),
                    .PCD(PCD),
                    .PCPlus4D(PCPlus4D),
                    .RegWriteW(RegWriteW),
                    .RDW(RDW),
                    .ResultW(ResultW),
                    .RegWriteE(RegWriteE),
                    .ALUSrcE(ALUSrcE),
                    .MemWriteE(MemWriteE),
                    .ResultSrcE(ResultSrcE),
                    .JumpE(JumpE),
                    .ALUControlE(ALUControlE),
                    .RD1_E(RD1_E),
                    .RD2_E(RD2_E),
                    .Imm_Ext_E(Imm_Ext_E),
                    .RD_E(RD_E),
                    .PCE(PCE),
                    .PCPlus4E(PCPlus4E),
                    .RS1_E(RS1_E),
                    .RS2_E(RS2_E)
                );
```

```
// Execute Stage
execute_cycle Execute (
                    .clk(clk),
                    .rst(rst),
                    .RegWriteE(RegWriteE),
                    .ALUSrcE(ALUSrcE),
                    .MemWriteE(MemWriteE),
                    .ResultSrcE(ResultSrcE),
                    .JumpE(JumpE),
                    .ALUControlE(ALUControlE),
                    .RD1_E(RD1_E),
                    .RD2_E(RD2_E),
                    .Imm_Ext_E(Imm_Ext_E),
                    .RD_E(RD_E),
                    .PCE(PCE),
                    .PCPlus4E(PCPlus4E),
                    .PCSrcE(PCSrcE),
                    .PCTargetE(PCTargetE),
                    .RegWriteM(RegWriteM),
                    .MemWriteM(MemWriteM),
                    .ResultSrcM(ResultSrcM),
                    .RD_M(RD_M),
                    .PCPlus4M(PCPlus4M),
                    .WriteDataM(WriteDataM),
                    .ALU_ResultM(ALU_ResultM),
                    .ResultW(ResultW),
                    .ForwardA_E(ForwardAE),
                    .ForwardB_E(ForwardBE)
                );
```



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

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

    // Hazard Unit
    hazard_unit Forwarding_block (
                        .rst(rst),
                        .RegWriteM(RegWriteM),
                        .RegWriteW(RegWriteW),
                        .RD_M(RD_M),
                        .RD_W(RDW),
                        .Rs1_E(RS1_E),
                        .Rs2_E(RS2_E),
                        .ForwardAE(ForwardAE),
                        .ForwardBE(ForwardBE)
                        );
endmodule
```

11. Test the processor design by generating the appropriate clock and reset. Copy the <u>image</u> of your testbench code here.



```
`timescale 1ns / 1ps

module tb();

    reg clk=0, rst;

    always begin
        clk = ~clk;
        #50;
    end

    initial begin
        rst <= 1'b0;
        #200;
        rst <= 1'b1;
        #1000;
        $finish;
    end

//    initial begin
//        $dumpfile("dump.vcd");
//        $dumpvars(0);
//    end

    Pipeline_top dut (.clk(clk), .rst(rst));
endmodule
```

Answer:

12. **Verify if the register file and data memory is getting updated according to the set of instructions (mentioned earlier).**

Copy verified **Register file** waveform here (show only the Registers that get updated, CLK, and RESET), along with the **Data Memory** location getting updated:

(FACING PROBLEM IN CHECKING THE VALUES OF REGISTER FILE)

13. **What is the total number of cycles needed to issue the program given above on the pipelined MIPS Processor? What is the CPI of the program?**

Answer:

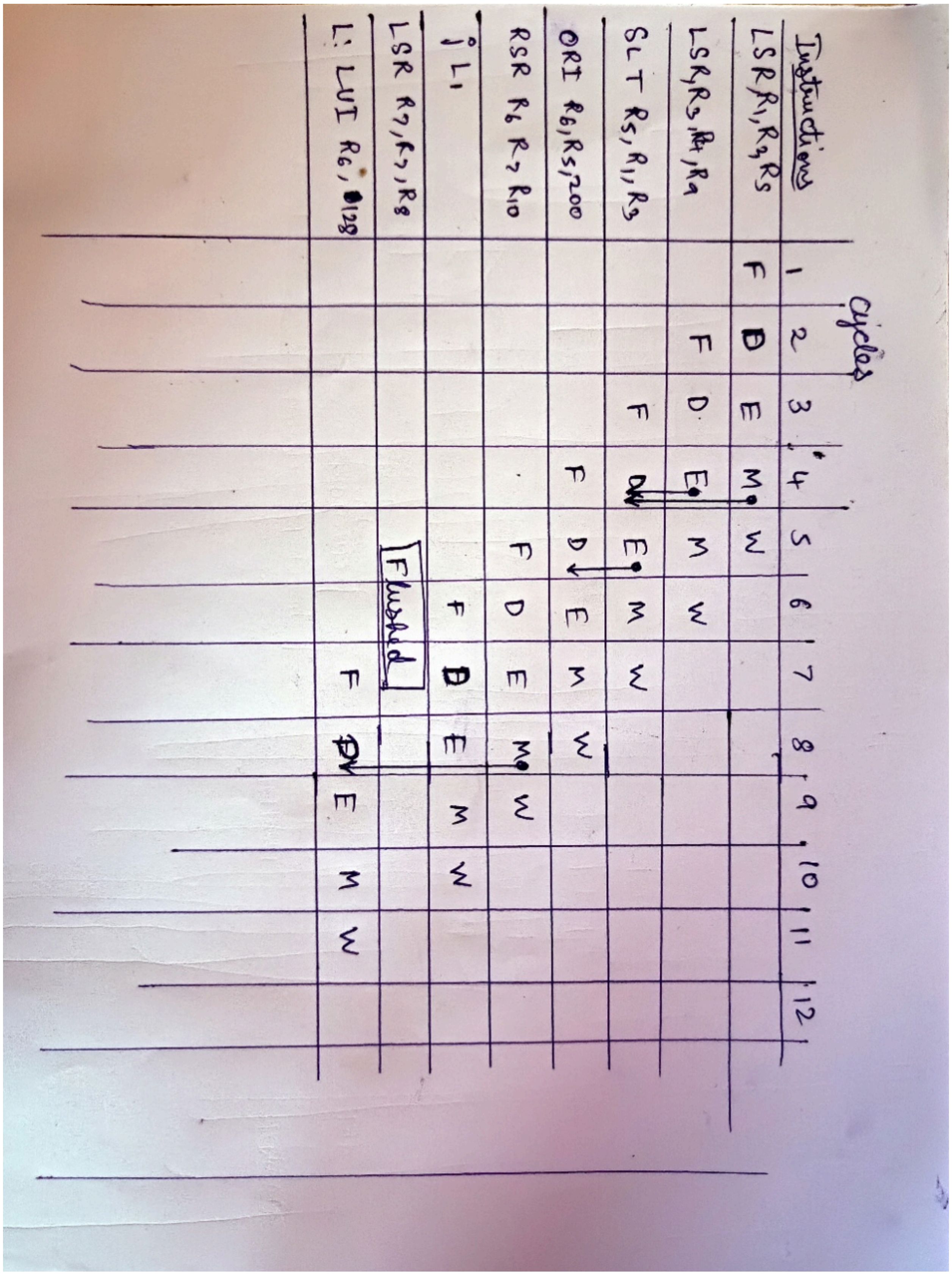
cycles require : 11

No of instructions : 8

Cpi = 11/8 = 1.375

14. **Make a diagram showing the clock by clock execution of each instruction, indicating stalling, forwarding etc wherever necessary.**

Answer:



Instructions	1	2	3	4	5	6	7	8	9	10	11	12
LSR R1, R3, R5	F	D	E	M	W							
LSR R3, R4, R9		F	D	E	M	W						
SLT R5, R11, R3			F	D	E	M	W					
ORI R6, R5, 200				F	D	E	M	W				
RSR R6 R7 R10					F	D	E	M	W			
j L1						F	D	E	M	W		
LSR R7, R7, R8						Flushed						
L: LUI R6, 128							F	D	E	M	W	

15. Your design synthesisable? Which target FPGA was used for synthesis?

Answer: yes. Fpga is Artix-7 (xc7a100tcsg324-1 (active))

16. Provide the synthesis report in tabular form (resources consumed)?

Answer:

```
Starting Synthesize : Time (s): cpu = 00:00:03 ; elapsed = 00:00:06 . Memory (MB): peak =
1252.242 ; gain = 0.000

Finished Synthesize : Time (s): cpu = 00:00:03 ; elapsed = 00:00:08 . Memory (MB): peak =
1252.242 ; gain = 0.000

Detailed RTL Component Info :

+---Adders :

           2 Input   32 Bit       Adders := 2

+---Registers :

                    32 Bit    Registers := 14

                     5 Bit    Registers := 5

                     3 Bit    Registers := 1

                     1 Bit    Registers := 9

+---Multipliers :

                   32x32  Multipliers := 1

+---RAMs :

                  1024 Bit         (32 X 32 bit)         RAMs := 1

+---Muxes :

           2 Input   32 Bit        Muxes := 12

          10 Input   32 Bit        Muxes := 1

           4 Input   32 Bit        Muxes := 2

           5 Input    3 Bit        Muxes := 1

           2 Input    3 Bit        Muxes := 1
```

3 Input	3 Bit	Muxes := 1
2 Input	2 Bit	Muxes := 5
3 Input	2 Bit	Muxes := 1
2 Input	1 Bit	Muxes := 4

Unrelated Questions

What were the problems you faced during the implementation of the processor?

Answer: too many syntax errors. Jump instruction implementation. How to resolve data hazards. How to avoid stalling when jump is executed

Did you implement the processor on your own? If you took help from someone whose help did you take? Which part of the design did you take help for?

Answer: yes i did most of the things by myself. Took help from chat gpt for resolving syntax errors. Some youtube videos wherever i stuck.

Honor Code Declaration by student:

- My answers to the above questions are my own work.
- I have not shared the codes/answers written by me with any other students. (I might have helped clear doubts of other students).
- I have not copied other's code/answers to improve my results. (I might have got some doubts cleared from other students).

Name: Priyank Khandelwal **Date:** 14/4/24
ID No.: 2021A3PS2450G