Assignment -3 (Computer Architecture Lab)

Group no. 05

Name and ID Student 1: **Vinayak Tulsyan (2020AAPS0442H)**

Name and ID Student 2: **Anshuman Mishra (2020AAPS0448H)**

Implement a pipeline-based MIPS processor in Verilog that can execute at least 12 instructions, including R-type, I-type, and J-type (conditional and unconditional). The instructions should be chosen so that the three hazards, namely data, structural, and control hazards, should arise, and these issues should be resolved using techniques such as stalling, flushing, and forwarding based on the optimal solution.

Part-A

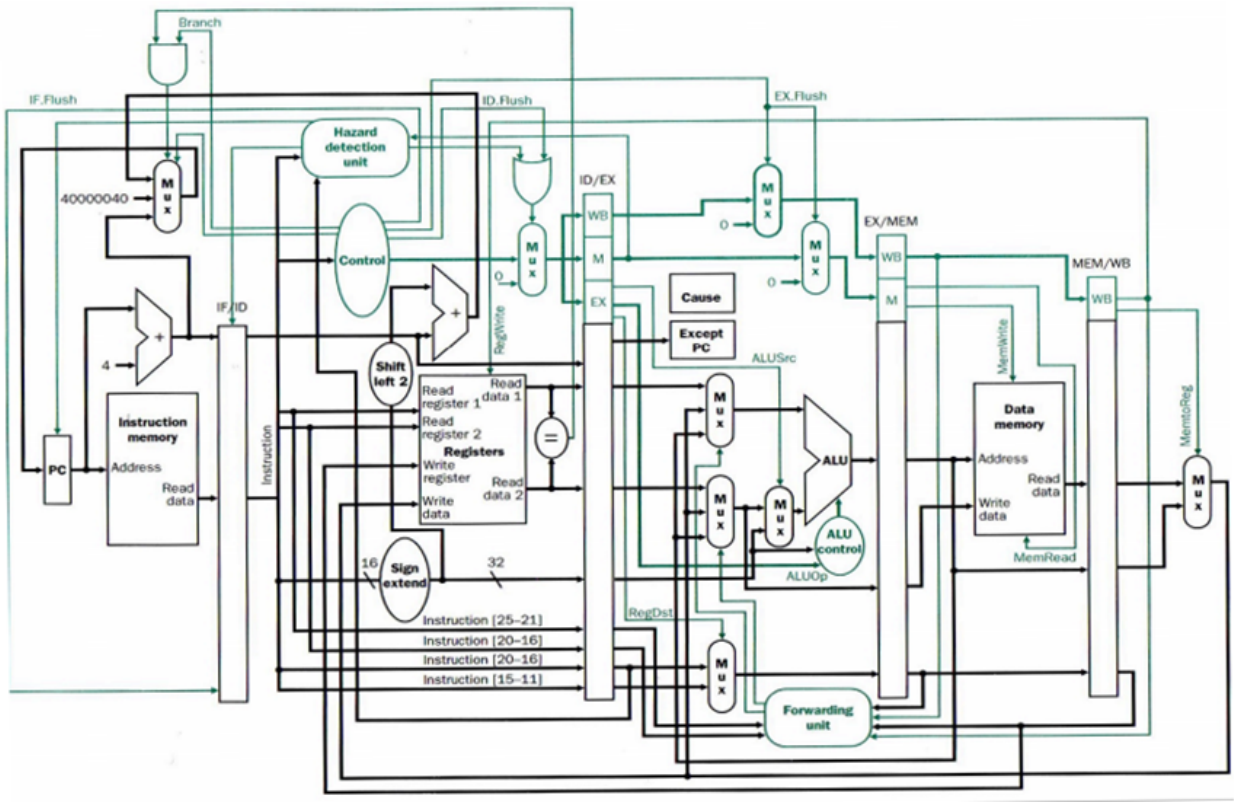
Edit this document containing the following once you are done with your Verilog implementation:

1) The program should indicate the hazard points and the result stored in each register

2) Mention the techniques used to overcome the hazards encountered in the program chosen.

   Stalling and forwarding is used to overcome the hazards.

3) The block diagram of the processor showing the required data path, control path, and hazard detection unit.



4) The Truth Table for the control unit.

	RegDst	Jump	Branch	MemRead	MemtoReg	ALUOp	Memwrite	ALUSrc	RegWrite	jr	reg1	jal	bne
R-type	1	0	0	0	0	10	0	0	1	0	0	0	0
Lw	0	0	0	1	1	00	0	1	1	0	0	0	0
sw	0	0	0	0	0	00	1	1	0	0	0	0	0
beq	0	0	1	0	0	01	0	0	0	0	0	0	0
bne	0	0	0	0	0	01	0	0	0	0	0	0	1
addi	0	0	0	0	0	00	0	1	1	0	0	0	0
j	0	1	0	0	0	0	0	0	0	0	0	0	0
jr	0	0	0	0	0	0	0	0	0	1	1	0	0
jal	0	1	0	0	0	0	0	0	1	0	0	1	0
jalr	0	0	0	0	0	0	0	0	1	1	1	1	0

The control unit's signals were all concatenated to be sent into the pipeline register ID-EX.

5) The program that you load in the instruction memory. (4 instructions minimum for each type)

The instructions used:

```
00000000000000000000000000000000 //nop
10001100000000010000000000000100 //lw $1, 4($0)
10001100000000010000000000001000 //lw $2, 8($0)
00000000001000100001100000000000 //add $3, $1, $2
10101100000000110000000000001100 //sw $3, 12($0)
00000000001000100010100000000100 //and $5, $1, $2
10101100000001010000000000010000 //sw $5, 16($0)
00000000001000100011000000000101 //or $6, $1, $2
10101100000001100000000000010100 //sw $6, 20($0)
00000000001000100101000000000010 //sub $10, $1, $2
10001100000000010000000000000100 //lw $1, 4($0)
10001100000000010000000000000100 //lw $2, 4($0)
00010000010000010000000000000001 //beq $1, $2, 1
10101100000001010000000000011000 //sw $10, 24($0)
00000000110000000011100010001010 //sll $7,$6,2
00000000110000000100000010001011 //srl $8,$6,2
00000100110010010000000001000011 //addi $9,$6,67
10000000011001010000000000000001 //bne $3, $5, 1
00000000001000100001100000000000 //add $3, $1, $2
00111100000000000000000000001001 //jalr $9
00000000001000100001100000000000 //add $3, $1, $2
00011100000000000000000000011000 //jal 24
00000000001000100001100000000000 //add $3, $1, $2
00000000001000100001100000000000 //add $3, $1, $2
00000100110010010000000001011111 //addi $9,$6,95
00001100000000000000000000001001 //jr $9
10001100000000010000000000000100 //lw $1, 4($0)
10001100000000010000000000001000 //lw $2, 8($0)
00001000000000000000000000000110 //j 6
```

6) Show the pipeline diagram showing pipelining, the instructions, hazards, etc. An example is shown below.

7) Paste the code from all the Verilog files/modules/mem files that are implemented.

```
`timescale 1ns / 1ps



module adder(

input [31:0] A,

input [31:0] B,

output [31:0] sum

);



assign sum = A+B;







endmodule
```

```
`timescale 1ns / 1ps



module ALU_control(

    input [1:0] ALUOp,

    input [5:0] funct,

    output reg [2:0] ALU_control

    );

always@(*)

begin

if(ALUOp == 2'b00)
```

```
    ALU_control = 3'b010;
else if(ALUOp == 2'b01)
    ALU_control = 3'b110;
else if(ALUOp[1]) begin
    case(funct[3:0])
    4'b0000 : ALU_control = 3'b010;  //add
    4'b0010 : ALU_control = 3'b110;  //sub
    4'b0100 : ALU_control = 3'b000;  //and
    4'b0101 : ALU_control = 3'b001;  //or
    4'b1010 : ALU_control = 3'b111;  //sll
    4'b1011 : ALU_control = 3'b011;  //srl

    endcase
end
end
endmodule
```

```
`timescale 1ns / 1ps


module ALU(
    input [31:0] ALU_src_1,
```

```
    input [31:0] ALU_src_2,
    input [2:0] ALU_control,
    input [4:0] shamt,
    output reg [31:0] ALU_out,
    output reg zero,
    input clk
    );


always @(*) begin
    case(ALU_control)
    3'b000 : ALU_out = ALU_src_1 & ALU_src_2;
    3'b001 : ALU_out = ALU_src_1 | ALU_src_2;
    3'b010 : ALU_out = ALU_src_1 + ALU_src_2;
    3'b110 : ALU_out = ALU_src_1 - ALU_src_2;
    3'b111 : ALU_out = ALU_src_1 << shamt;
    3'b011 : ALU_out = ALU_src_1 >> shamt;
    default: ALU_out = 32'b 0;
    endcase
end


always@(*)
begin
if(ALU_out == 0)
```

```
    zero = 1;
else
    zero = 0;
end


endmodule
```

```
`timescale 1ns / 1ps


module control(
    input [5:0] opcode,
    output reg RegDst,
    output reg Jump,
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg [1:0] ALUOp,
    output reg MemWrite,
    output reg ALUSrc,
    output reg RegWrite,
    output reg jr,
    output reg reg1,
```

```
    output reg jal,
    output reg bne,
    input stall,
    input clk
    );
always@(*)
begin
case(opcode)
    6'b000000 : begin       //  R - type
                        RegDst = 1;
                        ALUSrc = 0;
                        MemtoReg = 0;
                        RegWrite = 1;
                        MemRead = 0;
                        MemWrite = 0;
                        Branch = 0;
                        ALUOp[1] = 1;
                        ALUOp[0] = 0;
                        Jump = 0;
                        jr = 0;
                        reg1 = 0;
                        jal = 0;
                        bne = 0;
```

```
                    end
    6'b100011 : begin           //LW
                    RegDst = 0;
                    ALUSrc = 1;
                    MemtoReg = 1;
                    RegWrite = 1;
                    MemRead = 1;
                    MemWrite = 0;
                    Branch = 0;
                    ALUOp[1] = 0;
                    ALUOp[0] = 0;
                    Jump = 0;
                    jr = 0;
                    reg1 = 0;
                    jal = 0;
                    bne = 0;
                    end
    6'b101011 : begin           //SW
                    RegDst = 0;
                    ALUSrc = 1;
                    MemtoReg = 0;
                    RegWrite = 0;
                    MemRead = 0;
```

```
                        MemWrite = 1;
                        Branch = 0;
                        ALUOp[1] = 0;
                        ALUOp[0] = 0;
                        Jump = 0;
                        jr = 0;
                        reg1 = 0;
                        jal = 0;
                        bne = 0;
                        end
    6'b000100 : begin           //beq
                        RegDst = 0;
                        ALUSrc = 0;
                        MemtoReg = 0;
                        RegWrite = 0;
                        MemRead = 0;
                        MemWrite = 0;
                        Branch = 1;
                        ALUOp[1] = 0;
                        ALUOp[0] = 1;
                        Jump = 0;
                        jr = 0;
                        reg1 = 0;
```

```
                    jal = 0;
                    bne = 0;
                    end
    6'b100000 : begin           //bne
                    RegDst = 0;
                    ALUSrc = 0;
                    MemtoReg = 0;
                    RegWrite = 0;
                    MemRead = 0;
                    MemWrite = 0;
                    Branch = 0;
                    bne = 1;
                    ALUOp[1] = 0;
                    ALUOp[0] = 1;
                    Jump = 0;
                    jr = 0;
                    reg1 = 0;
                    jal = 0;
                    end
    6'b000001 : begin           //addi
                    RegDst = 0;
                    ALUSrc = 1;
                    MemtoReg = 0;
```

```
                    RegWrite = 1;
                    MemRead = 0;
                    MemWrite = 0;
                    Branch = 0;
                    ALUOp[1] = 0;
                    ALUOp[0] = 0;
                    Jump = 0;
                    jr = 0;
                    reg1 = 0;
                    jal = 0;
                    bne = 0;
                    end
6'b000010 : begin           //j
                    RegDst = 0;
                    ALUSrc = 0;
                    MemtoReg = 0;
                    RegWrite = 0;
                    MemRead = 0;
                    MemWrite = 0;
                    Branch = 0;
                    ALUOp[1] = 0;
                    ALUOp[0] = 0;
                    Jump = 1;
```

```
                    jr = 0;
                    reg1 = 0;
                    jal = 0;
                    bne = 0;
                    end
    6'b000011 : begin           //jr
                    RegDst = 0;
                    ALUSrc = 0;
                    MemtoReg = 0;
                    RegWrite = 0;
                    MemRead = 0;
                    MemWrite = 0;
                    Branch = 0;
                    ALUOp[1] = 0;
                    ALUOp[0] = 0;
                    Jump = 0;
                    jr = 1;
                    reg1 = 1;
                    jal = 0;
                    bne = 0;
                    end
    6'b000111 : begin           //jal
                    RegDst = 0;
```

```
                        ALUSrc = 0;
                        MemtoReg = 0;
                        RegWrite = 1;
                        MemRead = 0;
                        MemWrite = 0;
                        Branch = 0;
                        ALUOp[1] = 0;
                        ALUOp[0] = 0;
                        Jump = 1;
                        jr = 0;
                        reg1 = 0;
                        jal = 1;
                        bne = 0;
                        end
    6'b001111 : begin           //jalr
                        RegDst = 0;
                        ALUSrc = 0;
                        MemtoReg = 0;
                        RegWrite = 1;
                        MemRead = 0;
                        MemWrite = 0;
                        Branch = 0;
                        ALUOp[1] = 0;
```

```
                        ALUOp[0] = 0;
                        Jump = 0;
                        jr = 1;
                        reg1 = 1;
                        jal = 1;
                        bne = 0;
                        end
    default : begin
                        RegDst = 0;
                        ALUSrc = 0;
                        MemtoReg = 0;
                        RegWrite = 0;
                        MemRead = 0;
                        MemWrite = 0;
                        Branch = 0;
                        ALUOp[1] = 0;
                        ALUOp[0] = 0;
                        Jump = 0;
                        jr = 0;
                        reg1 = 0;
                        jal = 0;
                        bne = 0;
                        end
```

```
endcase



end



/*always@(*)

begin

if(stall) begin

    Branch = 0;

    MemRead = 0;

    MemWrite = 0;

    RegWrite = 0;

    MemtoReg = 0;

    RegDst = 0;

    ALUSrc = 0;

    ALUOp = 0;

end

end*/




endmodule
```

```
`timescale 1ns / 1ps


module data_memory(
    input [31:0] address,
    input [31:0] write_data,
    output reg [31:0] read_data,
    input mem_read,
    input mem_write,
    input clk
    );


reg [7:0] dataMem [127:0];
initial begin
    $readmemh("data_mem.mem", dataMem);
    end


always@(posedge clk)
begin
    if(mem_write)
    begin
              {dataMem[address+3],  dataMem[address+2],  dataMem[address+1],
   dataMem[address]} <= write_data;
    end
```

```
    if(mem_read)

    begin

                read_data  <=  {dataMem[address+3],  dataMem[address+2],
   dataMem[address+1], dataMem[address]};

    end



end



endmodule
```

```
`timescale 1ns / 1ps



module EX_MEM(

      input  id_ex_Branch,  id_ex_MemRead,  id_ex_MemWrite,  id_ex_RegWrite,
   id_ex_MemtoReg, clk, zero,

    input [31:0] id_ex_read_reg_data_2, ALU_out, adder2_result,

    input [4:0] write_reg,

          output   reg   ex_mem_Branch,   ex_mem_MemRead,   ex_mem_MemWrite,
   ex_mem_RegWrite, ex_mem_MemtoReg, ex_mem_zero,

          output   reg   [31:0]   ex_mem_read_reg_data_2,   ex_mem_ALU_out,
   ex_mem_adder2_result,

    output reg [4:0] ex_mem_write_reg,

    input stall

    );
```

```
    always@ (negedge clk) begin

            ex_mem_Branch <= id_ex_Branch;
            ex_mem_MemRead <= id_ex_MemRead;
            ex_mem_MemWrite <= id_ex_MemWrite;
            ex_mem_RegWrite <= id_ex_RegWrite;
            ex_mem_MemtoReg <= id_ex_MemtoReg;
            ex_mem_zero <= zero;
            ex_mem_write_reg <= write_reg;
            ex_mem_read_reg_data_2 <= id_ex_read_reg_data_2;
            ex_mem_ALU_out <= ALU_out;
            ex_mem_adder2_result <= adder2_result;

    end

endmodule
```

```
`timescale 1ns / 1ps

module forwarding_unit(
```

```
 input ex_mem_RegWrite, mem_wb_RegWrite, clk,

    input  [4:0]  id_ex_read_reg_1,  id_ex_read_reg_2,  ex_mem_write_reg,
mem_wb_write_reg,

 output reg [1:0] forwardA, forwardB

 );



 always@ (negedge clk) begin

       if(ex_mem_RegWrite & (ex_mem_write_reg != 0) & (ex_mem_write_reg
== id_ex_read_reg_1))

         forwardA <= 2'b10;

              else  if(mem_wb_RegWrite  &  (mem_wb_write_reg  !=  0)  &
(mem_wb_write_reg == id_ex_read_reg_1))

         forwardA <= 2'b01;

     else

         forwardA <= 2'b00;

 end



 always@ (negedge clk) begin

       if(ex_mem_RegWrite & (ex_mem_write_reg != 0) & (ex_mem_write_reg
== id_ex_read_reg_2))

         forwardB <= 2'b10;

              else  if(mem_wb_RegWrite  &  (mem_wb_write_reg  !=  0)  &
(mem_wb_write_reg == id_ex_read_reg_2))

         forwardB <= 2'b01;

     else
```

```
            forwardB <= 2'b00;

    end




endmodule
```

```
`timescale 1ns / 1ps



module ID_EX(

    input Branch, MemRead, MemWrite, RegWrite, MemtoReg, RegDst, ALUSrc,
  clk,

    input [1:0] ALUOp,

        input  [31:0]  if_id_PC_plus_4,  read_reg_data_1,  read_reg_data_2,
  extended, if_id_instruction,

    input [4:0] read_reg_1,



          output   reg   id_ex_Branch,   id_ex_MemRead,   id_ex_MemWrite,
  id_ex_RegWrite, id_ex_MemtoReg, id_ex_RegDst, id_ex_ALUSrc,

    output reg [1:0] id_ex_ALUOp,

    output reg [4:0] id_ex_read_reg_1,

         output   reg   [31:0]   id_ex_PC_plus_4,   id_ex_read_reg_data_1,
  id_ex_read_reg_data_2, id_ex_extended, id_ex_instruction,

    input stall



```

```
);



always@ (negedge clk) begin

    if(!stall) begin

        id_ex_Branch <= Branch;

        id_ex_MemRead <= MemRead;

        id_ex_MemWrite <= MemWrite;

        id_ex_RegWrite <= RegWrite;

        id_ex_MemtoReg <= MemtoReg;

        id_ex_RegDst <= RegDst;

        id_ex_ALUSrc <= ALUSrc;

        id_ex_ALUOp <= ALUOp;

        id_ex_PC_plus_4 <= if_id_PC_plus_4;

        id_ex_read_reg_data_1 <= read_reg_data_1;

        id_ex_read_reg_1 <= read_reg_1;

        id_ex_read_reg_data_2 <= read_reg_data_2;

        id_ex_extended <= extended;

        id_ex_instruction <= if_id_instruction;

        end

     else

        begin

            id_ex_Branch <= 0;
```

```
                id_ex_MemRead <= 0;

                id_ex_MemWrite <= 0;

                id_ex_RegWrite <= 0;

                id_ex_MemtoReg <= 0;

                id_ex_RegDst <= 0;

                id_ex_ALUSrc <= 0;

                id_ex_ALUOp <= 0;

                id_ex_PC_plus_4 <= if_id_PC_plus_4;

                id_ex_read_reg_data_1 <= read_reg_data_1;

                id_ex_read_reg_1 <= read_reg_1;

                id_ex_read_reg_data_2 <= read_reg_data_2;

                id_ex_extended <= extended;

                id_ex_instruction <= if_id_instruction;

            end


    end


endmodule
```

```
`timescale 1ns / 1ps


module IF_ID(

    input clk,
```

```
    input [31:0] PC_plus_4, instruction,
    output reg [31:0] if_id_PC_plus_4, if_id_instruction,
    input stall
    );



    always@ (negedge clk)
    begin
    if(!stall) begin
            if_id_PC_plus_4 <= PC_plus_4;
            if_id_instruction <= instruction;
    end
    end


endmodule
```

```
`timescale 1ns / 1ps


module instruction_fetch(
    input clk,
    input reset,
    input Jump,
    input jr,
```

```
    input Branch,
    input bne,
    input zero,
    input jal,
    input signed [31:0] shifted,
    input [27:0] Jump_address,
    input [31:0] jr_address,
    output [31:0] jal_address,
    output [31:0] instruction,
    output reg [31:0] PC,
    input stall
    );


instruction_memory I1 (
.read_address(PC),
.reset(reset),
.instruction(instruction)
);


wire [31:0] PC_plus_four;
assign PC_plus_four = PC + 4;
assign jal_address = PC + 8;
```

```
always@(negedge clk)
begin
if(reset == 0)
    PC <= 0;

else if((Branch && zero) & !stall)
    PC <= PC + 4 + shifted;

else if((bne && !zero) & !stall)
    PC <= PC + 4 + shifted;

else if((Jump) & !stall)
    PC <= {PC_plus_four[31:28],Jump_address};

else if((jr) & !stall)
    PC <= jr_address;

else if (!stall)
    PC <= PC + 4;
end
```

```
endmodule
```

```
`timescale 1ns / 1ps


module instruction_memory(
    input [31:0] read_address,
    input reset,
    output [31:0] instruction
    );


reg [7:0] Mem [115:0];
assign    instruction    =    {Mem[read_address],    Mem[read_address+1],
   Mem[read_address+2], Mem[read_address+3]};


initial
begin
            $readmemh("instruction_mem.mem", Mem);
end


endmodule
```

```
`timescale 1ns / 1ps


module MEM_WB(
    input ex_mem_RegWrite, ex_mem_MemtoReg, clk,
    input [31:0] ex_mem_ALU_out, read_data,
    input [4:0] ex_mem_write_reg,
    output reg mem_wb_MemtoReg, mem_wb_RegWrite,
    output reg [4:0] mem_wb_write_reg,
    output reg [31:0] mem_wb_read_data, mem_wb_ALU_out,
    input stall
    );



    always@ (negedge clk)
      begin

            mem_wb_RegWrite <= ex_mem_RegWrite;
            mem_wb_MemtoReg <= ex_mem_MemtoReg;
            mem_wb_read_data <= read_data;
            mem_wb_ALU_out <= ex_mem_ALU_out;
            mem_wb_write_reg <= ex_mem_write_reg;
      end
endmodule
```

```
`timescale 1ns / 1ps


module mux(
    input [31:0] A,
    input [31:0] B,
    input sel,
    output  [31:0] Out
);


assign Out = sel? B:A;
endmodule
```

```
`timescale 1ns / 1ps


module mux2(
    input [31:0] A,
    input [31:0] B,
    input [31:0] C,
    input [1:0] sel,
    output  [31:0] Out
);
```

```
assign Out = sel[1]? C:(sel[0]? B:A);

endmodule
```

```
`timescale 1ns / 1ps



module PC(

    input clk,

    input reset,

    input [31:0] in,

    output [31:0] PC

    );


assign PC = reset ? in: 0;


endmodule
```

```
`timescale 1ns / 1ps


module register_file(

    input [4:0] read_reg_1,
```

```
    input [4:0] read_reg_2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_reg_data_1,
    output [31:0] read_reg_data_2,
    input RegWrite,
    input clk
    );


reg [31:0] regMem [31:0];
initial begin
    $readmemh("register_mem.mem", regMem);
    end



always@(posedge clk)
begin
    if(RegWrite)
        regMem[write_reg] = write_data;
end


assign read_reg_data_1 = regMem[read_reg_1];
assign read_reg_data_2 = regMem[read_reg_2];
```

```
endmodule
```

```
`timescale 1ns / 1ps




module shifter(
    input [31:0] A,
    output reg signed [31:0] Out
    );


always@(*)
begin
Out = A<<2;
end
endmodule



`timescale 1ns / 1ps
```

```
module shifter2(
    input [25:0] A,
    output reg [27:0] Out
    );

always@(*)
begin
Out = {2'b00,(A<<2)};
end
endmodule
```

```
`timescale 1ns / 1ps
module sign_extender(
    input [15:0] A,
    output reg [31:0] OUT
    );
always@(*)

begin
OUT[15:0] = A;
OUT[31:16] = {16{A[15]}};
```

```
end


endmodule
```

```
`timescale 1ns / 1ps




module stalling_unit(
    input clk, id_ex_MemRead,
    input [4:0] if_id_Rs, if_id_Rt, id_ex_Rt,
    output reg [2:0] stall
    );


    always@ (posedge clk) begin
             if(id_ex_MemRead  &  ((id_ex_Rt  ==  if_id_Rs)  |  (id_ex_Rt  ==
   if_id_Rt)))
            stall <= 1;
        else
            stall <= 0;
    end


    /*reg check;
```

```
    always@(*)
       check <= id_ex_MemRead & ((id_ex_Rt == if_id_Rs) | (id_ex_Rt ==
   if_id_Rt));

    always@(*) begin
    if(check)
            stall <= 1;
        else
            stall <= 0;
    end*/

       //assign stall = (id_ex_MemRead && ((id_ex_Rt == if_id_Rs) ||
   (id_ex_Rt == if_id_Rt)))? 1:0;

endmodule
```

```
`timescale 1ns / 1ps

module top(
    input clk,
    input reset,
    output [31:0] aluout,
```

```
    output [31:0] Instruction,
    output [31:0] ALU_A,
    output [31:0] ALU_B,


    output memread,
    output memtoreg,
    output [1:0] aluop,
    output [2:0] alucontrol,
    output memwrite,
    output alusrc,
    output regdst,
    output regwrite,
    output [4:0] writereg,
    output [31:0] Read_reg_data_2,
    output [31:0] Readdata


    );
wire [31:0] instruction;
wire [31:0] PC_plus_4;
wire [31:0] PC;
wire Jump;
wire jr;
wire reg1;
```

```
wire jal;

wire Branch;

wire bne;

wire MemRead;

wire MemtoReg;

wire [1:0] ALUOp;

wire MemWrite;

wire ALSrc;

wire RegDst;

wire [31:0] write_data;

wire RegWrite;

wire [4:0] write_reg;

wire [31:0]read_reg_data_1;

wire [31:0]read_reg_data_2;

wire [31:0] extended;

wire zero;

wire [31:0] ALU_out;

wire [2:0] ALU_control;

wire [31:0] read_data;

wire signed [31:0] shifted;

wire [31:0] adder2_result;

wire [31:0] mux4_result;

wire [27:0] Out;
```

```
wire [31:0] jr_address;

wire [4:0] read_reg_1;

wire [31:0] jal_address;

wire [4:0] write_reg_1;

wire [31:0] write_data_1;



wire    ex_mem_Branch,  ex_mem_MemRead,  ex_mem_MemWrite,  ex_mem_RegWrite,
   ex_mem_MemtoReg, ex_mem_zero;

wire [31:0] ex_mem_read_reg_data_2, ex_mem_ALU_out, ex_mem_adder2_result;

wire  [4:0] ex_mem_write_reg;



wire [2:0] stall;



instruction_fetch if1(

.clk(clk),

.reset(reset),

.Jump(Jump),

.jr(jr),

.jal(jal),

.Branch(Branch),

.bne(bne),

.zero(zero),

.shifted(shifted),

.Jump_address(Out),
```

```
.jr_address(jr_address),
.jal_address(jal_address),
.instruction(instruction),
.PC(PC),
.stall(stall)
);


wire [31:0] read_address;
wire [31:0] in;


mux m0(
    .A(PC_plus_4),
    .B(ex_mem_adder2_result),
    .sel(ex_mem_Branch & ex_mem_zero),
    .Out(in)
);


PC pc (
    .clk(clk),
    .reset(reset),
    .in(in),
    .PC(PC)
);
```

```
wire [31:0] if_id_instruction;
wire [31:0] if_id_PC_plus_4;



IF_ID if_id_uut(
    .clk(clk),
    .PC_plus_4(PC_plus_4),
    .instruction(instruction),
    .if_id_PC_plus_4(if_id_PC_plus_4),
    .if_id_instruction(if_id_instruction),
    .stall(stall[2])
);


wire     id_ex_Branch,   id_ex_MemRead,   id_ex_MemWrite,   id_ex_RegWrite,
   id_ex_MemtoReg, id_ex_RegDst, id_ex_ALUSrc;
wire  [1:0] id_ex_ALUOp;
wire [4:0] id_ex_read_reg_1;
wire             [31:0]      id_ex_PC_plus_4,      id_ex_read_reg_data_1,
   id_ex_read_reg_data_2, id_ex_extended, id_ex_instruction;


ID_EX id_ex_uut (
    .clk(clk),
```

```
    .ALUSrc(ALUSrc),
    .RegDst(RegDst),
    .MemtoReg(MemtoReg),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemWrite(MemWrite),
    .RegWrite(RegWrite),
    .ALUOp(ALUOp),
    .if_id_PC_plus_4(if_id_PC_plus_4),
    .read_reg_data_1(read_reg_data_1),
    .read_reg_1(read_reg_1),
    .read_reg_data_2(read_reg_data_2),
    .extended(extended),
    .if_id_instruction(if_id_instruction),
    .id_ex_ALUSrc(id_ex_ALUSrc),
    .id_ex_RegDst(id_ex_RegDst),
    .id_ex_MemtoReg(id_ex_MemtoReg),
    .id_ex_Branch(id_ex_Branch),
    .id_ex_MemRead(id_ex_MemRead),
    .id_ex_MemWrite(id_ex_MemWrite),
    .id_ex_RegWrite(id_ex_RegWrite),
    .id_ex_ALUOp(id_ex_ALUOp),
    .id_ex_PC_plus_4(id_ex_PC_plus_4),
```

```
    .id_ex_read_reg_data_1(id_ex_read_reg_data_1),
    .id_ex_read_reg_1(id_ex_read_reg_1),
    .id_ex_read_reg_data_2(id_ex_read_reg_data_2),
    .id_ex_extended(id_ex_extended),
    .id_ex_instruction(id_ex_instruction),
    .stall(stall[2])

);

EX_MEM ex_mem_uut (
    .clk(clk),
    .id_ex_Branch(id_ex_Branch),
    .id_ex_MemRead(id_ex_MemRead),
    .id_ex_MemWrite(id_ex_MemWrite),
    .id_ex_RegWrite(id_ex_RegWrite),
    .id_ex_MemtoReg(id_ex_MemtoReg),
    .zero(zero),
    .ALU_out(ALU_out),
    .id_ex_read_reg_data_2(id_ex_read_reg_data_2),
    .adder2_result(adder2_result),
    .write_reg(write_reg),
    .ex_mem_Branch(ex_mem_Branch),
```

```
    .ex_mem_MemRead(ex_mem_MemRead),
    .ex_mem_MemWrite(ex_mem_MemWrite),
    .ex_mem_RegWrite(ex_mem_RegWrite),
    .ex_mem_MemtoReg(ex_mem_MemtoReg),
    .ex_mem_zero(ex_mem_zero),
    .ex_mem_ALU_out(ex_mem_ALU_out),
    .ex_mem_read_reg_data_2(ex_mem_read_reg_data_2),
    .ex_mem_adder2_result(ex_mem_adder2_result),
    .ex_mem_write_reg(ex_mem_write_reg),
    .stall(stall[2])
);


wire mem_wb_MemtoReg, mem_wb_RegWrite;
wire [4:0] mem_wb_write_reg;
wire [31:0] mem_wb_read_data, mem_wb_ALU_out, mem_wb_write_data;


MEM_WB mem_wb_uut (
    .clk(clk),
    .ex_mem_RegWrite(ex_mem_RegWrite),
    .ex_mem_MemtoReg(ex_mem_MemtoReg),
    .ex_mem_ALU_out(ex_mem_ALU_out),
    .read_data(read_data),
    .ex_mem_write_reg(ex_mem_write_reg),
```

```
    .mem_wb_RegWrite(mem_wb_RegWrite),
    .mem_wb_MemtoReg(mem_wb_MemtoReg),
    .mem_wb_write_reg(mem_wb_write_reg),
    .mem_wb_ALU_out(mem_wb_ALU_out),
    .mem_wb_read_data(mem_wb_read_data),
    .stall(stall[2])
);


control c1(
    .opcode(if_id_instruction[31:26]),
    .RegDst(RegDst),
    .Jump(Jump),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .ALUOp(ALUOp),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .RegWrite(RegWrite),
    .jr(jr),
    .reg1(reg1),
    .jal(jal),
    .bne(bne),
```

```
    .stall(stall[0]),
    .clk(clk)
);
//mux m_c(
//    .A(
//    .B(
//);


adder a1(
.A(PC),
.B(4),
.sum(PC_plus_4)
);


mux m1(
    .A(id_ex_instruction[20:16]),
    .B(id_ex_instruction[15:11]),
    .sel(id_ex_RegDst),
    .Out(write_reg)
);


mux m6(
    .A(if_id_instruction[25:21]),
```

```
    .B(if_id_instruction[4:0]),
    .sel(reg1),
    .Out(read_reg_1)
);


mux m7(
    .A(mem_wb_write_reg),
    .B(5'b11111),
    .sel(jal),
    .Out(write_reg_1)
);


mux m8(
    .A(mem_wb_write_data),
    .B(jal_address),
    .sel(jal),
    .Out(write_data_1)
);


mux m3(
    .A(mem_wb_ALU_out),
    .B(mem_wb_read_data),
    .sel(mem_wb_MemtoReg),
```

```
    .Out(mem_wb_write_data)

);



data_memory d1(

    .address(ex_mem_ALU_out),

    .write_data(ex_mem_read_reg_data_2),

    .read_data(read_data),

    .mem_read(ex_mem_MemRead),

    .mem_write(ex_mem_MemWrite),

    .clk(clk)

);



register_file r1(

    .read_reg_1(read_reg_1),

    .read_reg_2(if_id_instruction[20:16]),

    .write_reg(write_reg_1),

    .write_data(write_data_1),

    .read_reg_data_1(read_reg_data_1),

    .read_reg_data_2(read_reg_data_2),

    .RegWrite(mem_wb_RegWrite),

    .clk(clk)

);
```

```
assign jr_address = read_reg_data_1;


sign_extender ex1(
    .A(if_id_instruction[15:0]),
    .OUT(extended)
);


mux m2(
    .A(id_ex_read_reg_data_2),
    .B(id_ex_extended),
    .sel(id_ex_ALUSrc),
    .Out(ALU_B)
);


ALU_control ALUc(
    .ALUOp(id_ex_ALUOp),
    .funct(id_ex_instruction[5:0]),
    .ALU_control(ALU_control)
);


wire [1:0] forwardA, forwardB;


forwarding_unit fu (
```

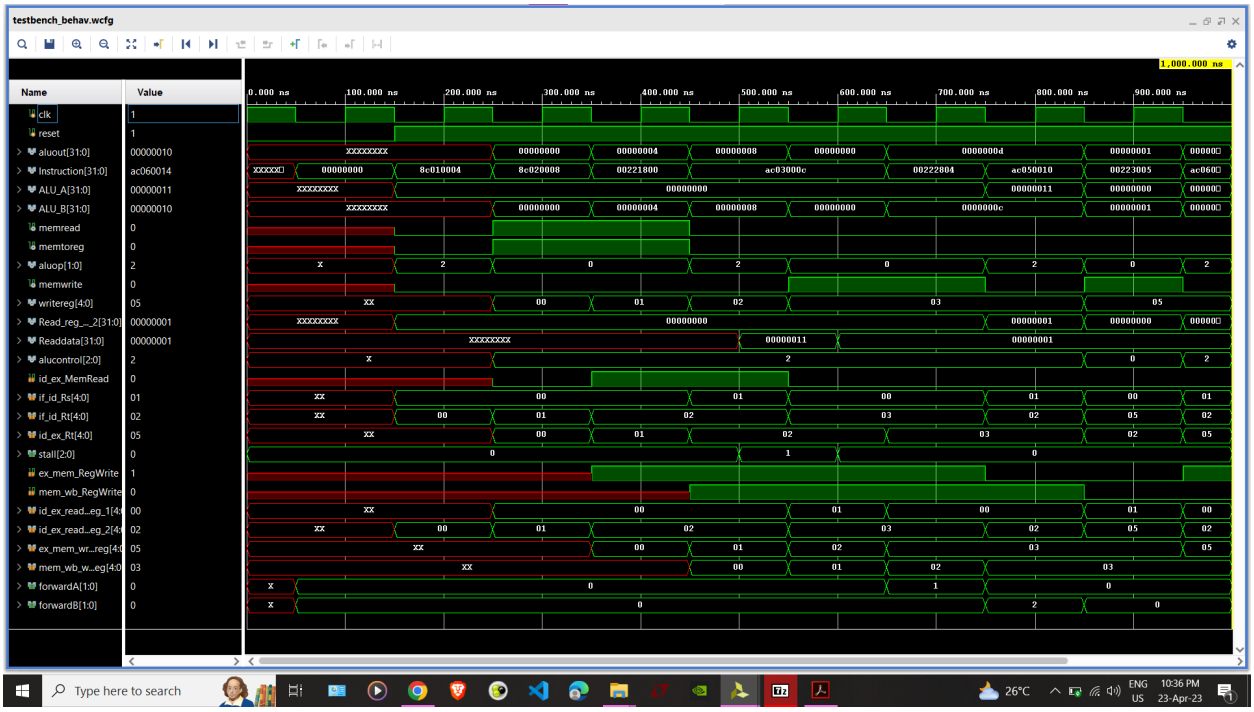
```
    .ex_mem_RegWrite(ex_mem_RegWrite),
    .mem_wb_RegWrite(mem_wb_RegWrite),
    .clk(clk),
    .id_ex_read_reg_1(id_ex_read_reg_1),
    .id_ex_read_reg_2(if_id_instruction[20:16]),
    .ex_mem_write_reg(ex_mem_write_reg),
    .mem_wb_write_reg(write_reg_1),
    .forwardA(forwardA),
    .forwardB(forwardB)
);
wire [31:0] alu_a, alu_b;


mux2 m01 (
    .A(id_ex_read_reg_data_1),
    .B(write_data_1),
    .C(ex_mem_ALU_out),
    .sel(forwardA),
    .Out(alu_a)
);
mux2 m02 (
    .A(ALU_B),
    .B(write_data_1),
    .C(ex_mem_ALU_out),
```

```
    .sel(forwardB),
    .Out(alu_b)
);



ALU alu1(
    .ALU_src_1(alu_a),
    .ALU_src_2(alu_b),
    .ALU_control(ALU_control),
    .shamt(id_ex_instruction[10:6]),
    .ALU_out(ALU_out),
    .zero(zero),
    .clk(clk)
);


shifter s1(
    .A(id_ex_extended),
    .Out(shifted)
);


adder a2(
    .A(id_ex_PC_plus_4),
    .B(shifted),
```

```
    .sum(adder2_result)
);
shifter2 s2(
    .A(instruction[25:0]),
    .Out(Out)
);
stalling_unit su (
    .clk(clk),
    .id_ex_MemRead(id_ex_MemRead),
    .if_id_Rs(read_reg_1),
    .if_id_Rt(if_id_instruction[20:16]),
    .id_ex_Rt(id_ex_instruction[20:16]),
    .stall(stall)
);


assign aluout = ALU_out;
assign Instruction = instruction;
assign ALU_A = read_reg_data_1;




assign memread = MemRead;
assign memtoreg = MemtoReg;
assign aluop = ALUOp;
```

```
assign memwrite = MemWrite;

assign alusrc =ALUSrc;

assign regdst = RegDst;

assign writedata = write_data;

assign regwrite = RegWrite;

assign writereg = write_reg;

assign Read_reg_data_2 = read_reg_data_2;

assign Readdata = read_data;

assign alucontrol = ALU_control;



endmodule
```

8) Paste screenshots of output waveforms.



Pipeline stages

1. IF/ID Stage (Pipeline register)

Variable name	No.of Bits
instruction	32
new PC	32

2. ID/EX Stage (Pipeline register)

Variable Name	No. of Bits
Sign extend	32
Read_Rs	32
Read_Rt	32
Write_Rd	5
lower26	26
new PC	32
RegDst	1
Jump	1
Branch	1
M read	1
M2 Reg	1
ALUOp2	2
MemWrite	1
ALUSrc	1

RegWrite	1
PC Src	2

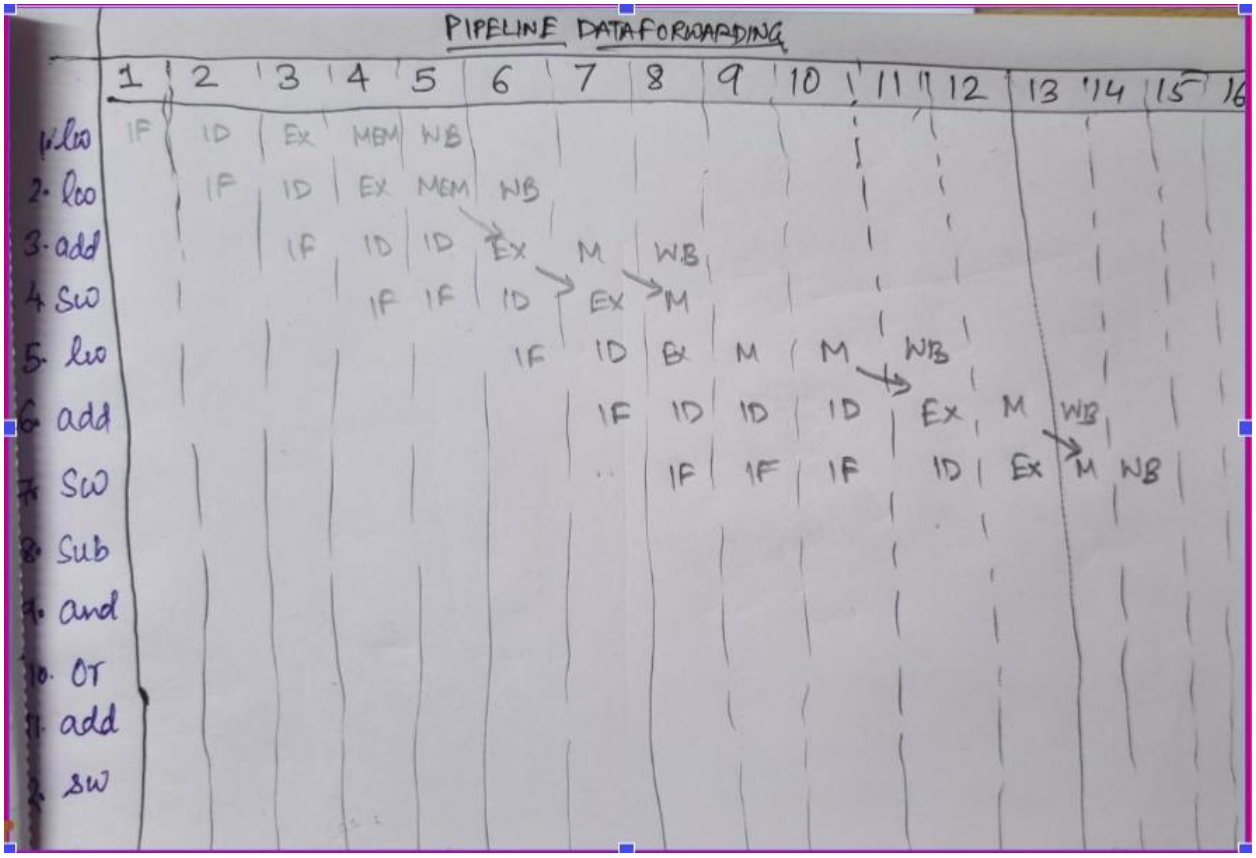
3. EX/MEM Stage (Pipeline register)

Variable name	No.of Bits
Write_Rd	5
Read_Rd	32
ALUOut	32
BT(Branch Target Address)	32
RegWrite	1
MemRead	1
MemWrite	1
Branch	1
Zero	1
PCSrc	2

4. MEM/WB Stage (Pipeline register)

Variable name	No.of Bits
Destination Reg	5
ALU_Output	32
Memory_Data	32

MemtoReg	1
RegWrite	1



Part-B

1) Upload all the. v and .mem files in a single zipped folder.
2) Upload the project file .ise or. prj. (Xilinx ISE/Vivado).
3) Upload this document after editing. (One document per group)

Note

1) Any help taken should be mentioned in the document, without which it will be considered a clear malpractice case, and **no marks** will be awarded.

Had discussions and debugging sessions with other batchmates.

2) Please stick to your respective lab groups.

Deadline: 23.04.2023 (11 p.m)