COMPUTER ARCHITECTURE (CS F342: PROJECT DEMONSTRATION)

GROUP MEMBERS WITH ID NUMBER:

1. ARYAMAN SEN (2022A3PS0667H)
2. VITTHAL TIWARY (2022A3PS0553H)
3. RAJAT PORWAL (2022A3PS0725H)
4. HARISH. R. RAMACHANDRAN (2022AAPS0310H)

GROUP NUMBER: 18

LAB SECTION: P2

SUBJECT: Computer Architecture: Project

TOPIC: Pipelining- with appropriate Forwarding and Data Hazard Detection Units

CONTRIBUTION OF EACH MEMBER:

All members have contributed equally, and are equally knowledgeable and competent enough to explain any work or portion of the project, taking into account each individual's knowledge, as well as collective group effort.

PROBLEM STATEMENT:

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

0000 NOR reg1, reg2, reg3

0004 ADDI reg4, reg1, ECD1

0008 SUB reg6, reg4, reg5

0012 LW reg7, 9(reg8)

0016 XNOR reg10, reg9, reg7

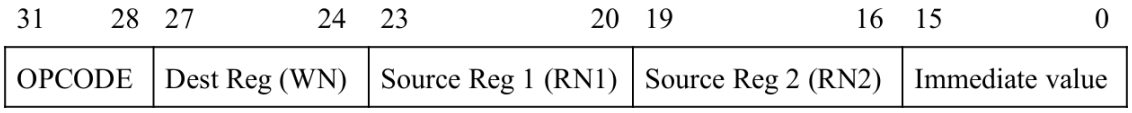
The register file is initialized with the following data

reg2 = 437AC reg3 = 03EC2

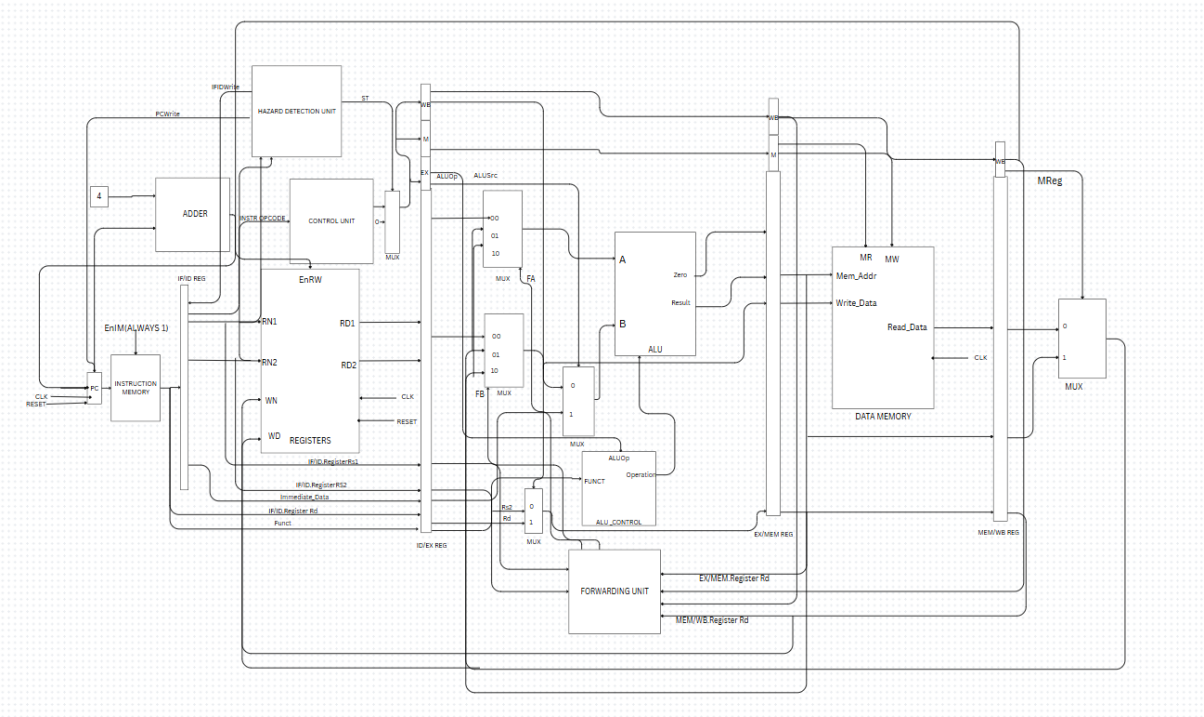
reg5 = CE331 reg8 = 3

reg9 = 24442

SPECIFIED OPCODE FORMAT:



BLOCK DIAGRAM



HEXADECIMAL DECODING OF ALL INSTRUCTIONS ACCORDING TO THE SPECIFIED OPCODE FORMAT:

1. NOR reg1, reg2, reg3

Hexadecimal representation of this instruction:

0000 0001 0010 0011 0000 0000 0000 0000*

2. ADDI reg4, reg1, ECD1

0001 0100 0100 0000 1110 1100 1101 0001

3. SUB reg6, reg4, reg5

0011 0110 0100 0101 0000 0000 0000 0000

4. LW reg7, 9(reg8)

0111 0111 1000 0000 0000000000001001

5. XNOR reg10, reg9, reg7

1111 1010 1001 0111 0000 0000 0000 0000

*** 0000 has been used to indicate values of fields that are not applicable (or can take arbitrary values) for each instruction, in accordance with the specific**

opcode format followed for all instructions specified as part of the problem statement.

FUNCTIONS OF ALL THE CORE PROCESSING UNITS:

1. **Instruction Memory-** stores the program instructions in addressable memory locations. The memory address comes from the Program Counter (PC), and upon each clock cycle (CLK), it outputs the instruction at the current PC address. The RESET signal initializes the memory state, while EnIM (Enable Instruction Memory) activates or deactivates the memory based on pipeline conditions.
2. **Adder-** a specialized arithmetic component that increments the Program Counter by 4 (for word-aligned instructions) to generate the next sequential instruction address. This value feeds back to the PC register for the next instruction fetch cycle, unless a branch or jump instruction alters the control flow.
3. **Control Unit-** decodes each instruction to determine the required operations. It extracts operation codes and generates the appropriate control signals for the datapath components. These signals include ALU operation codes

(ALUOPC), memory operation signals (MR, MW), register write signals (WN, WD), and selectors for the various multiplexers. It also handles condition codes and status flags for conditional execution.

4. **Register Bank-**A bank of general-purpose registers that store temporary data during program execution. It has dual read ports that output the values of two source registers (RD1, RD2) simultaneously based on register addresses from the instruction. It also has a write port controlled by WN (Write Number, to select the destination register) and WD (Write Data, the actual data to be written). Writing occurs during the clock edge when write is enabled.
5. **ALU (Arithmetic Logic Unit)-**performs arithmetic and logical operations based on the ALU_Op and ALU_Src signals, based on the instruction. The inputs A and B come from multiplexers that select either direct register values or forwarded values from later pipeline stages. Operations include ADD, SUB, AND, OR, NOR, XNOR, and shifts. The ALU outputs the result of the Operation, labelled as output **Result,** and has a zero flag labelled **Zero,** normally used for branch decisions.
6. **ALU Control Unit-** generates specific ALU operation codes based on the FUNCT field (6-bit

functional field obtained from the the instruction) and the 2-bit ALUOp control signal. This allows the ALU to perform the precise operation required by the current instruction, whether it's an R-type, I-type, or any other instruction format.

7. **Hazard Detection Unit-** monitors instruction dependencies to identify situations where the pipeline must stall. It receives register information from current and preceding instructions (PC01inc input) and detects load-use hazards where an instruction tries to use data that hasn't yet been loaded from memory. When a hazard is detected, it asserts the stall signal (ST) to freeze parts of the pipeline and prevents the PC from incrementing, effectively inserting a bubble in the pipeline.

8. **Forwarding Unit-** detects and resolves data hazards caused by instruction dependencies in the pipeline. It compares the source register numbers (Rs and Rt) of the instruction in the execute stage with the destination registers of instructions in later pipeline stages (EX/MEM Register Rd and MEM/WB Register Rd). When a match is found, indicating that a previous instruction is producing a result needed by the current instruction, the

forwarding unit controls the multiplexers (FA and FB) to route the data directly from the pipeline registers rather than waiting for it to be written back to the register file. This unit has inputs from multiple pipeline registers and outputs control signals to the ALU input multiplexers, enabling execution to continue without unnecessary stalls.

9. **Data Memory-**stores program data (as opposed to instructions) in addressable memory locations. Address input comes from Mem_Addr (typically ALU Result), and data input/output happens through Write_Data and Read_Data ports respectively. Memory access is controlled by MR (Memory Read) and MW (Memory Write) signals, according to whether data is **read** from memory or **written** into memory. Memory operations are synchronized with the clock (CLK).

Values of control signals during each instruction:

1. 0000 NOR reg1, reg2, reg3

   AluOp=00

   AluSrc=0

   MR=0

   MW=0

   EnIM=1

EnRW=1

MReg=0

RegDst=1

IFIDWRITE: 1 (allowed to write to IF/ID pipeline register)

FA: 0 (no forwarding needed for first source operand, reg2)

FB: 0 (no forwarding needed for second source operand, reg3)

PCWrite: 1 (PC can be incremented normally)

ST: 0 (no store operation)

2. 0004 ADDI reg4, reg1, ECD1

AluOp=01

AluSrc=1

MR=0

MW=0

EnIM=1

EnRW=1

MReg=0

RegDst=1

IFIDWRITE: 1

FA: 2

FB: Not used

PCWrite: 1

ST: 0

3. 0008 SUB reg6, reg4, reg5

   AluOp=10

   AluSrc=0

   MR=0

   MW=0

   EnIM=1

   EnRW=1

   MReg=0

   RegDst=1

IFIDWRITE: 1

FA: 1

FB: 0

PCWrite: 1

ST: 0

4. 0012 LW reg7, 9(reg8)

   AluOp=01

   AluSrc=1

   MR=1

   MW=0

   EnIM=1

   EnRW=1

   MReg=1

   RegDst=0

   IFIDWRITE: 1

FA: 0

FB: Not used

PCWrite: 1

ST: 0

5. 0016 XNOR reg10, reg9, reg7

AluOp=11

AluSrc=0

MR=0

MW=0

EnIM=1

EnRW=1

MReg=0

RegDst=1

IFIDWRITE: 0

FA: 0

FB: 0

PCWrite: 0

ST: 0

Default values of the control signals:

AluOp=01

AluSrc=0

MR=0

MW=0

EnIM=1

EnRW=0

MReg=0

RegDst=0

IFIDWRITE: 1

FA: 0

FB: 0

PCWrite: 1

ST: 0

THE CODE

The top module:

```
`timescale 1ns / 1ps

module datapath(
    input clk,rst,
    output [7:0]Clk_Cycle
    );
    counter COUNTER(.clk(clk),
                    .rst(rst),
                    .count(Clk_Cycle));

    wire AluSrc,MR,MW,MReg,EnIm,EnRW,IFIDWrite,PCWrite,ST,Zero;
    wire [1:0]AluOp,FA,FB;
    //Signals for IF_ID
    wire AluSrc_IF_ID,MR_IF_ID,MW_IF_ID,MReg_IF_ID,EnRW_IF_ID;
    wire [1:0]AluOp_IF_ID;
    //Signals for ID_EX
    wire AluSrc_ID_EX,MR_ID_EX,MW_ID_EX,MReg_ID_EX,EnRW_ID_EX;
    wire [1:0]AluOp_ID_EX;
    //Signals for EX_MEM
    wire MR_EX_MEM,MW_EX_MEM,MReg_EX_MEM,EnRW_EX_MEM;
    //Signals for MEM_WB
    wire MReg_MEM_WB,EnRW_MEM_WB;
    //Signals End

    wire [31:0]pc_to_instr_mem;

    reg_pc PC ( .clk(clk),
                .rst(rst),
                .PCWrite(PCWrite),
                .pc_out(pc_to_instr_mem));

     wire [31:0]Instruction_to_IF_ID;

     mem_instr INSTRUCTION_MEMORY (  .EnIM(EnIm),
                                     .Address(pc_to_instr_mem),
                                     .Instruction(Instruction_to_IF_ID));

     //Register for IF ID
     wire [3:0]Opcode,reg_Rs_IF_ID,reg_Rt_IF_ID,reg_Rd_IF_ID;
     //Register for ID EX
     wire [3:0]reg_Rs_ID_EX,reg_Rt_ID_EX,reg_Rd_ID_EX,reg_dest_ID_EX;
     //REgister for EX MEM
     wire [3:0]reg_Rd_EX_MEM;
     //Register for MEM WB
     wire [3:0]reg_Rd_MEM_WB;
     //Registers Done

     wire [15:0]Immediate;
     wire [31:0]Imm_Extended,Imm_Extended_ID_EX;

     if_id IF_ID (   .clk(clk),
                     .rst(rst),
                     .IF_ID_Write(IFIDWrite),
                     .Instruction(Instruction_to_IF_ID),
                     .Opcode(Opcode),
                     .reg_dest(reg_Rd_IF_ID),
                     .reg_source_1(reg_Rs_IF_ID),
                     .reg_source_2(reg_Rt_IF_ID),
                     .Immediate_value(Immediate));
```

```
61      hazard_detection_unit HDU ( .ID_EX_RegisterRd(reg_Rt_ID_EX),
62                                  .IF_ID_RegisterRs1(reg_Rs_IF_ID),
63                                  .IF_ID_RegisterRs2(reg_Rt_IF_ID),
64                                  .ID_EX_MemRead(MR_ID_EX),
65                                  .ST(ST),
66                                  .IFIDWrite(IFIDWrite),
67                                  .PCWrite(PCWrite));
68
69      control_unit CONTROL_UNIT ( .clk(clk),
70                                  .rst(rst),
71                                  .Opcode(Opcode),
72                                  .AluSrc(AluSrc),
73                                  .MW(MW),
74                                  .MR(MR),
75                                  .MReg(MReg),
76                                  .EnIM(EnIm),
77                                  .EnRW(EnRW),
78                                  .RegDst(RegDst),
79                                  .AluOp(AluOp));
80
81      mux_control_unit MUX_ST (   .AluSrc(AluSrc),
82                                  .MW(MW),
83                                  .MR(MR),
84                                  .MReg(MReg),
85                                  .EnRW(EnRW),
86                                  .ST(ST),
87                                  .AluOp(AluOp),
88                                  .AluSrc_out(AluSrc_IF_ID),
89                                  .MW_out(MW_IF_ID),
90                                  .MR_out(MR_IF_ID),
91                                  .MReg_out(MReg_IF_ID),
92                                  .EnRW_out(EnRW_IF_ID),
93                                  .AluOp_out(AluOp_IF_ID));

95      wire [31:0]Read_data_1,Read_data_2,Write_Data,Read_data_1_ID_EX,Read_data_2_ID_EX;
96
97      registers REGISTERS (   .clk(clk),
98                              .rst(rst),
99                              .EnRW(EnRW_MEM_WB),
100                             .reg_dest(reg_Rd_MEM_WB),
101                             .reg_source_1(reg_Rs_IF_ID),
102                             .reg_source_2(reg_Rt_IF_ID),
103                             .Write_data(Write_Data),
104                             .read_data_1(Read_data_1),
105                             .read_data_2(Read_data_2));
106
107     sign_extender SIGN_EXTEND (.offset(Immediate),
108                                .sign_ext_offset(Imm_Extended));

107     sign_extender SIGN_EXTEND (.offset(Immediate),
108                                .sign_ext_offset(Imm_Extended));
109
110     id_ex ID_EX (   .clk(clk),
111                     .rst(rst),
112                     .AluSrc(AluSrc_IF_ID),
113                     .MW(MW_IF_ID),
114                     .MR(MR_IF_ID),
115                     .MReg(MReg_IF_ID),
116                     .EnRW(EnRW_IF_ID),
117                     .AluOp(AluOp_IF_ID),
118                     .reg_data_1(Read_data_1),
119                     .reg_data_2(Read_data_2),
120                     .Imm_Extended(Imm_Extended),
121                     .if_id_reg_rs(reg_Rs_IF_ID),
122                     .if_id_reg_rt(reg_Rt_IF_ID),
123                     .if_id_reg_rd(reg_Rd_IF_ID),
124                     .AluSrc_out(AluSrc_ID_EX),
125                     .MW_out(MW_ID_EX),
126                     .MR_out(MR_ID_EX),
127                     .MReg_out(MReg_ID_EX),
128                     .EnRW_out(EnRW_ID_EX),
129                     .AluOp_out(AluOp_ID_EX),
130                     .reg_data_1_out(Read_data_1_ID_EX),
131                     .reg_data_2_out(Read_data_2_ID_EX),
132                     .Imm_Extended_out(Imm_Extended_ID_EX),
133                     .id_ex_reg_rs(reg_Rs_ID_EX),
134                     .id_ex_reg_rt(reg_Rt_ID_EX),
135                     .id_ex_reg_rd(reg_Rd_ID_EX));
136
137     wire [31:0]Alu_out,Alu_out_EX_MEM,Alu_out_MEM_WB,Mem_data,Mem_Data_MEM_WB,Alu_In_1,Alu_In_2,Alu_In_2_3Mux_2Mux;
```

```
    mux_2_to_1_4_bit MUX3 ( .a(reg_Rd_ID_EX),
                            .b(reg_Rt_ID_EX),
                            .sel(MReg_ID_EX),
                            .out(reg_dest_ID_EX));

    forwarding_unit FORWARDING_UNIT (   .ID_EX_RegisterRs1(reg_Rs_ID_EX),
                                        .ID_EX_RegisterRs2(reg_Rt_ID_EX),
                                        .EX_MEM_RegisterRd(reg_Rd_EX_MEM),
                                        .MEM_WB_RegisterRd(reg_Rd_MEM_WB),
                                        .EX_MEM_RegWrite(EnRW_EX_MEM),
                                        .MEM_WB_RegWrite(MReg_MEM_WB),
                                        .ForwardA(FA),
                                        .ForwardB(FB));

    mux_3_to_1 MUX_3TO1_1 ( .a(Read_data_1_ID_EX),
                            .b(Write_Data),
                            .c(Alu_out_EX_MEM),
                            .sel(FA),
                            .out(Alu_In_1));

    mux_3_to_1 MUX_3TO1_2 ( .a(Read_data_2_ID_EX),
                            .b(Write_Data),
                            .c(Alu_out_EX_MEM),
                            .sel(FB),
                            .out(Alu_In_2_3Mux_2Mux));

    mux_2_to_1 MUX1 (   .a(Alu_In_2_3Mux_2Mux),
                        .b(Imm_Extended_ID_EX),
                        .sel(AluSrc_ID_EX),
                        .out(Alu_In_2));

    arithmetic_logic_unit ALU ( .data_in_1(Alu_In_1),
                                .data_in_2(Alu_In_2),
                                .control(AluOp_ID_EX),
                                .zero(Zero),
                                .alu_out(Alu_out));

    ex_mem EX_MEM ( .clk(clk),
                    .rst(rst),
                    .MW(MW_ID_EX),
                    .MR(MR_ID_EX),
                    .MReg(MReg_ID_EX),
                    .EnRW(EnRW_ID_EX),
                    .Alu_out_in(Alu_out),
                    .id_ex_reg_rd(reg_dest_ID_EX),
                    .ex_mem_reg_rd(reg_Rd_EX_MEM),
                    .Alu_out_out(Alu_out_EX_MEM),
                    .MW_out(MW_EX_MEM),
                    .MR_out(MR_EX_MEM),
                    .MReg_out(MReg_EX_MEM),
                    .EnRW_out(EnRW_EX_MEM));

    mem_data DATA_MEMORY (  .clk(clk),
                            .rst(rst),
                            .Address(Alu_out_EX_MEM),
                            .Write_data(),
                            .MR(MR_EX_MEM),
                            .MW(MW_EX_MEM),
                            .Mem_Data(Mem_data));

    mem_wb MEM_WB ( .clk(clk),
                    .rst(rst),
                    .MReg(MReg_EX_MEM),
                    .EnRW(EnRW_EX_MEM),
                    .Alu_out_in(Alu_out_EX_MEM),
                    .Mem_Data_in(Mem_data),
                    .ex_mem_reg_rd(reg_Rd_EX_MEM),
                    .MReg_out(MReg_MEM_WB),
                    .EnRW_out(EnRW_MEM_WB),
                    .mem_wb_reg_rd(reg_Rd_MEM_WB),
                    .Alu_out_out(Alu_out_MEM_WB),
                    .Mem_Data_out(Mem_Data_MEM_WB));

    mux_2_to_1 MUX2 (   .a(Alu_out_MEM_WB),
                        .b(Mem_Data_MEM_WB),
                        .sel(MReg_MEM_WB),
                        .out(Write_Data));


endmodule
```

Counter and mux initialisation

```
`timescale 1ns / 1ps

module mux_3_to_1(
    input [31:0]a,b,c,
    input [1:0]sel,
    output reg [31:0]out
    );
    always@(*) begin
        out<=(sel==2'b10)?c:(sel==2'b01)?b:a;
    end
endmodule

module mux_2_to_1 (
    input [31:0]a,b,
    input sel,
    output reg [31:0]out
    );
    always@(*) begin
        out<=(sel)?b:a;
    end

endmodule

module mux_2_to_1_4_bit (
    input [3:0]a,b,
    input sel,
    output reg [3:0]out
    );
    always@(*) begin
        out<=(sel)?b:a;
    end

endmodule

module mux_control_unit(
    input AluSrc,MW,MR,MReg,EnRW,ST,
    input [1:0]AluOp,
    output reg AluSrc_out,MW_out,MR_out,MReg_out,EnRW_out,
    output reg [1:0]AluOp_out
    );
    always@(*) begin
        if(ST) begin
            AluSrc_out<=1'b0;
            MW_out<=1'b0;
            MR_out<=1'b0;
            MReg_out<=1'b0;
            EnRW_out<=1'b0;
            AluOp_out<=2'd0;
        end
        else begin
            AluSrc_out<=AluSrc;
            MW_out<=MW;
            MR_out<=MR;
            MReg_out<=MReg;
            EnRW_out<=EnRW;
            AluOp_out<=AluOp;
        end
    end

endmodule

module counter (
    input clk,
    input rst,
    output reg [7:0] count
);
```

```
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count <= 8'b0;
        end else begin
            count <= count + 1;
        end
    end

endmodule
```

Program counter register:

```
`timescale 1ns / 1ps

module reg_pc(
    input clk,rst,PCWrite,
    output  [31:0]pc_out
    );
    reg [31:0] pc_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      pc_reg <= 32'd0; // Reset PC to the beginning of memory (address 0)
    end else if (PCWrite) begin
        pc_reg <= pc_reg + 32'd4; // Increment PC by 4 (assuming byte addressing)
      end
    // If PCWrite is low, the PC holds its current value
  end

  assign pc_out = pc_reg; // Output the current PC value
endmodule
```

Instruction memory:

```
`timescale 1ns / 1ps

module mem_instr(
    input EnIM,
    input [31:0]Address,
    output reg [31:0]Instruction
    );
    integer i;
    reg [7:0]Instruction_Memory[0:255];
    initial begin
        Instruction_Memory[3]=8'h01;Instruction_Memory[2]=8'h23;Instruction_Memory[1]=8'h00;Instruction_Memory[0]=8'h00;
        Instruction_Memory[7]=8'h14;Instruction_Memory[6]=8'h10;Instruction_Memory[5]=8'hEC;Instruction_Memory[4]=8'hD1;
        Instruction_Memory[11]=8'h36;Instruction_Memory[10]=8'h45;Instruction_Memory[9]=8'h00;Instruction_Memory[8]=8'h00;
        Instruction_Memory[15]=8'h70;Instruction_Memory[14]=8'h87;Instruction_Memory[13]=8'h00;Instruction_Memory[12]=8'h09;
        Instruction_Memory[19]=8'hFA;Instruction_Memory[18]=8'h97;Instruction_Memory[17]=8'h00;Instruction_Memory[16]=8'h00;
        for(i=20;i<256;i=i+1) begin
            Instruction_Memory[i]=8'd0;
        end
    end
    always@(*) begin
        if(EnIM)begin
            Instruction<={Instruction_Memory[Address+3],Instruction_Memory[Address+2],Instruction_Memory[Address+1],Instruction_Memory[Address]};
        end
    end
endmodule
```

IF/ID interface:

```
`timescale 1ns / 1ps

module if_id(
    input clk,rst,IF_ID_Write,
    input [31:0] Instruction,
    output reg [3:0]Opcode,reg_dest,reg_source_1,reg_source_2,
    output reg [15:0]Immediate_value
    );
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            Opcode<=4'd0;
            reg_dest<=4'd0;
            reg_source_1<=4'd0;
            reg_source_2<=4'd0;
            Immediate_value<=16'd0;
        end
        else if (IF_ID_Write) begin
            Opcode<=Instruction[31:28];
            reg_dest<=Instruction[27:24];
            reg_source_1<=Instruction[23:20];
            reg_source_2<=Instruction[19:16];
            Immediate_value<=Instruction[15:0];
        end
    end
endmodule
```

Hazard Detection unit:

```
`timescale 1ns / 1ps

module hazard_detection_unit(
    input [3:0] ID_EX_RegisterRd, /* Destination register in the ID/EX Pipeline register of the previous instruction*/
    input [3:0] IF_ID_RegisterRs1, /* One source register of the current instruction in the IF/ID pipeline register */
    input [3:0] IF_ID_RegisterRs2, /* Second source register of the current instruction in the IF/ID pipeline register */
    input ID_EX_MemRead, /* This is used to check whether the instruction in the instructions set is a load instruction or not */
    output reg ST, /* This effectively performs the stall operation, wherein all the other control signals are
initialized/brought back to zero */
    output reg IFIDWrite, /* This control signal is used to flush changes in the IF/ID pipeline register */
    output reg PCWrite /* This is used to disable PC changes */
);

    // Load-use hazard detection
    always @(*) begin
        // Default values - no hazard detected
        ST = 1'b0;
        IFIDWrite = 1'b1;  // Enable IF/ID register updates
        PCWrite = 1'b1;    // Enable PC updates

        // Load-use hazard: When a load instruction is followed by an instruction that uses the loaded data
        // In our instruction set, this occurs between LW and XNOR
        if (ID_EX_MemRead &&
            ((ID_EX_RegisterRd == IF_ID_RegisterRs1) || (ID_EX_RegisterRd == IF_ID_RegisterRs2))) begin
            ST = 1'b1;      // We activate the stall signal over here //
            IFIDWrite = 1'b0;  // Disable further updates in IF/ID
            PCWrite = 1'b0;    // Disable updates in PC by not allowing it to fetch a new instruction or increment //
        end
    end
endmodule
```

Control unit:

```
`timescale 1ns / 1ps

module control_unit(
    input clk, rst,
    input [3:0]Opcode,
    output reg AluSrc,MW,MR,MReg,EnIM,EnRW,RegDst,
    output reg [1:0]AluOp
    );

    parameter NOR=4'b0000;
    parameter ADDI=4'b0001;
    parameter LW=4'b0111;
    parameter SUB=4'b0011;
    parameter XNOR=4'b1111;

    always @(posedge clk or posedge rst) begin
    if (rst) begin
        EnIM = 1'b0;
        EnRW = 1'b0;
    end
    else begin
        EnIM = 1'b1;
        EnRW = 1'b1;
    end
    end

     always@(*) begin
         if(Opcode==NOR) begin
             AluOp<=2'b00;
             AluSrc<=1'b0;
             EnIM = 1'b1;
             EnRW = 1'b1;
             MR<=1'b0;
             MW<=1'b0;
             MReg<=1'b0;
             RegDst<=1'b1;
         end
         else if(Opcode==ADDI)begin
             AluOp<=2'b01;
             AluSrc<=1'b1;
             EnIM = 1'b1;
             EnRW = 1'b1;
             MR<=1'b0;
             MW<=1'b0;
             MReg<=1'b0;
             RegDst<=1'b1;
         end
         else if(Opcode==LW)begin
             AluOp<=2'b01;
             AluSrc<=1'b1;
             EnIM = 1'b1;
             EnRW = 1'b1;
             MR<=1'b1;
             MW<=1'b0;
             MReg<=1'b1;
             RegDst<=1'b0;
         end
```

```
57          end
58          else if(Opcode==SUB)begin
59              AluOp<=2'b10;
60              AluSrc<=1'b0;
61              EnIM = 1'b1;
62              EnRW = 1'b1;
63              MR<=1'b0;
64              MW<=1'b0;
65              MReg<=1'b0;
66              RegDst<=1'b1;
67          end
68          else if(Opcode==XNOR)begin
69              AluOp<=2'b11;
70              AluSrc<=1'b0;
71              EnIM = 1'b1;
72              EnRW = 1'b1;
73              MR<=1'b0;
74              MW<=1'b0;
75              MReg<=1'b0;
76              RegDst<=1'b1;
77          end
78          else begin
79              AluOp<=2'b01;
80              AluSrc<=1'b0;
81              EnIM = 1'b1;
82              EnRW = 1'b1;
83              MR<=1'b0;
84              MW<=1'b0;
85              MReg<=1'b0;
86              RegDst<=1'b0;
87          end
88
89      end
90  endmodule
```

Register File:

```
1  `timescale 1ns / 1ps
2
3  module registers(
4      input clk,rst,EnRW,
5      input [3:0] reg_dest,reg_source_1,reg_source_2,
6      input [31:0]Write_data,
7      output reg [31:0]read_data_1,read_data_2
8      );
9      integer i;
10     reg [31:0]registers[0:15];
11
12     initial begin
13         registers[0]=32'd0;
14         registers[1]=32'd0;
15         registers[2]=32'h000437ac;
16         registers[3]=32'h00003ec2;
17         registers[4]=32'd0;
18         registers[5]=32'h000ce331;
19         registers[6]=32'd0;
20         registers[7]=32'd0;
21         registers[8]=32'd3;
22         registers[9]=32'h00024442;
23         for(i=10;i<16;i=i+1) begin
24             registers[i]=i+1;
25         end
26     end

28     always@(*) begin
29         if(rst) begin
30             read_data_1<=32'd0;
31             read_data_2<=32'd0;
32         end
33         else begin
34             read_data_1<=registers[reg_source_1];
35             read_data_2<=registers[reg_source_2];
36         end
37     end
38     always@(negedge clk) begin
39         if(EnRW) begin
40             registers[reg_dest]<=Write_data;
41         end
42     end
43 endmodule
```

Sign Extender:

```
`timescale 1ns / 1ps

module sign_extender(
    input [15:0] offset,
    output reg [31:0] sign_ext_offset
    );

    always @(*) begin
    sign_ext_offset = {{16{offset[15]}}, offset};
    end

endmodule
```

ID/EX interface:

```
`timescale 1ns / 1ps

module id_ex(
    input clk,rst,AluSrc,MW,MR,MReg,EnRW,
    input [1:0]AluOp,
    input [31:0] reg_data_1,reg_data_2,Imm_Extended,
    input [3:0]if_id_reg_rs,if_id_reg_rt,if_id_reg_rd,
    output reg AluSrc_out,MW_out,MR_out,MReg_out,EnRW_out,
    output reg [1:0]AluOp_out,
    output reg [31:0] reg_data_1_out,reg_data_2_out,Imm_Extended_out,
    output reg [3:0]id_ex_reg_rs,id_ex_reg_rt,id_ex_reg_rd
    );
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            AluSrc_out<=1'b0;
            MW_out<=1'b0;
            MR_out<=1'b0;
            MReg_out<=1'b0;
            EnRW_out<=1'b0;
            AluOp_out<=2'd0;
            reg_data_1_out<=32'd0;
            reg_data_2_out<=32'd0;
            id_ex_reg_rs<=4'd0;
            id_ex_reg_rt<=4'd0;
            id_ex_reg_rd<=4'd0;
            Imm_Extended_out<=32'd0;
        end
```

```
        else begin
            AluSrc_out<=AluSrc;
            MW_out<=MW;
            MR_out<=MR;
            MReg_out<=MReg;
            EnRW_out<=EnRW;
            AluOp_out<=AluOp;
            reg_data_1_out<=reg_data_1;
            reg_data_2_out<=reg_data_2;
            id_ex_reg_rs<=if_id_reg_rs;
            id_ex_reg_rt<=if_id_reg_rt;
            id_ex_reg_rd<=if_id_reg_rd;
            Imm_Extended_out<=Imm_Extended;
        end
    end
endmodule
```

Forwarding unit:

```
`timescale 1ns / 1ps

module forwarding_unit(
    // Source registers from ID/EX stage
    input [3:0] ID_EX_RegisterRs1,  // First source register in ID/EX stage
    input [3:0] ID_EX_RegisterRs2,  // Second source register in ID/EX stage

    // Destination registers from later pipeline stages
    input [3:0] EX_MEM_RegisterRd,  // Destination register in EX/MEM stage
    input [3:0] MEM_WB_RegisterRd,  // Destination register in MEM/WB stage

    // Control signals indicating register write operations
    input EX_MEM_RegWrite,          // Register write in EX/MEM stage
    input MEM_WB_RegWrite,          // Register write in MEM/WB stage

    // Forwarding control outputs
    output reg [1:0] ForwardA,      // Forwarding control for first ALU operand
    output reg [1:0] ForwardB       // Forwarding control for second ALU operand
);
initial begin
    ForwardA = 2'b00;
    ForwardB = 2'b00;
end
```

```
25      always @(*) begin
26          // ForwardA logic (for first ALU input - Rs1)
27          if (EX_MEM_RegWrite &&
28              (EX_MEM_RegisterRd != 0) &&
29              (EX_MEM_RegisterRd == ID_EX_RegisterRs1)) begin
30              // EX hazard - forward from EX/MEM pipeline register
31              ForwardA = 2'b10;  // Select value from EX/MEM stage
32          end
33          else if (MEM_WB_RegWrite &&
34                  (MEM_WB_RegisterRd != 0) &&
35                  (MEM_WB_RegisterRd == ID_EX_RegisterRs1)) begin
36              // MEM hazard - forward from MEM/WB pipeline register
37              ForwardA = 2'b01;  // Select value from MEM/WB stage
38          end
39          else begin
40              // No forwarding needed
41              ForwardA = 2'b00;  // Select original value from register file
42          end
43
44          // ForwardB logic (for second ALU input - Rs2)
45          if (EX_MEM_RegWrite &&
46              (EX_MEM_RegisterRd != 0) &&
47              (EX_MEM_RegisterRd == ID_EX_RegisterRs2)) begin
48              // EX hazard - forward from EX/MEM pipeline register
49              ForwardB = 2'b10;  // Select value from EX/MEM stage
50          end
51          else if (MEM_WB_RegWrite &&
52                  (MEM_WB_RegisterRd != 0) &&
53                  (MEM_WB_RegisterRd == ID_EX_RegisterRs2)) begin
54              // MEM hazard - forward from MEM/WB pipeline register
55              ForwardB = 2'b01;  // Select value from MEM/WB stage
56          end
57          else begin
58              // No forwarding needed
59              ForwardB = 2'b00;  // Select original value from register file
60          end
61      end
62  endmodule
```

Arithmetic Logic Unit:

```
`timescale 1ns / 1ps

module arithmetic_logic_unit(
    input [31:0] data_in_1, data_in_2,
    input [1:0] control,
    output reg zero,
    output reg [31:0] alu_out
    );

    always @(*) begin
    case (control)
        2'b00 : alu_out <= ~(data_in_1 | data_in_2);
        2'b01 : alu_out <= (data_in_1 + data_in_2);
        2'b10 : alu_out <= (data_in_1 - data_in_2);
        2'b11 : alu_out <= ~(data_in_1 ^ data_in_2);
    endcase
    zero <= (alu_out == 32'd0) ? 1 : 0;
    end
endmodule
```

EX/MEM interface:

```
`timescale 1ns / 1ps

module ex_mem(
    input clk,rst,MW,MR,MReg,EnRW,
    input [31:0]Alu_out_in,
    input [3:0]id_ex_reg_rd,
    output reg [3:0]ex_mem_reg_rd,
    output reg [31:0]Alu_out_out,
    output reg MW_out,MR_out,MReg_out,EnRW_out
     );
     always@(posedge clk or posedge rst) begin
        if(rst) begin
            MW_out<=1'b0;
            MR_out<=1'b0;
            MReg_out<=1'b0;
            EnRW_out<=1'b0;
            Alu_out_out<=32'd0;
            ex_mem_reg_rd<=4'd0;
        end
        else begin
            MW_out<=MW;
            MR_out<=MR;
            MReg_out<=MReg;
            EnRW_out<=EnRW;
            Alu_out_out<=Alu_out_in;
            ex_mem_reg_rd<=id_ex_reg_rd;
        end
     end
endmodule
```

Data memory:

```
`timescale 1ns / 1ps

module mem_data(
    input clk,rst,
    input [31:0]Address,Write_data,
    input MR,MW,
    output reg [31:0]Mem_Data
    );
    integer i;
    reg [7:0]Data_Memory[0:15];

    initial begin
        for(i=0;i<256;i=i+1) begin
            Data_Memory[i]<=i+1;
        end
    end
    always@(posedge clk or posedge rst) begin
        if(MW) begin
            {Data_Memory[Address + 3], Data_Memory[Address + 2], Data_Memory[Address + 1], Data_Memory[Address]} <= Write_data;
        end
    end
    always@(*) begin
        if(MR) begin
        Mem_Data <= {Data_Memory[Address + 3], Data_Memory[Address + 2], Data_Memory[Address + 1], Data_Memory[Address]};
        end
    end
endmodule
```

MEM/WB interface:

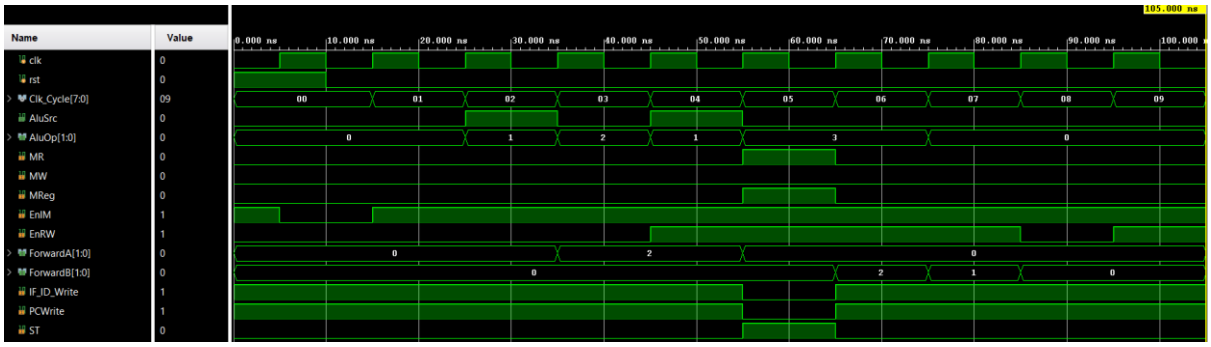
```
`timescale 1ns / 1ps

module mem_wb(
    input clk,rst,MReg,EnRW,
    input [31:0]Alu_out_in,Mem_Data_in,
    input [3:0]ex_mem_reg_rd,
    output reg MReg_out,EnRW_out,
    output reg [3:0]mem_wb_reg_rd,
    output reg [31:0]Alu_out_out,Mem_Data_out
    );
    always@(posedge clk or posedge rst) begin
        if(rst) begin
            MReg_out<=1'b0;
            EnRW_out<=1'b0;
            Alu_out_out<=32'd0;
            Mem_Data_out<=32'd0;
            mem_wb_reg_rd<=4'd0;
        end
        else begin
            MReg_out<=MReg;
            EnRW_out<=EnRW;
            Alu_out_out<=Alu_out_in;
            Mem_Data_out<=Mem_Data_in;
            mem_wb_reg_rd<=ex_mem_reg_rd;
        end
    end
endmodule
```

Testbench:

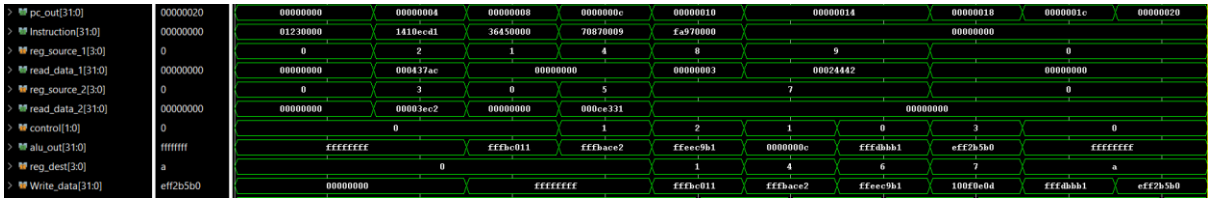
```
3  module datapath_tb;
4
5      reg clk;
6      reg rst;
7      wire [7:0]Clk_Cycle;
8
9      datapath DUT (
10         .clk(clk),
11         .rst(rst),
12         .Clk_Cycle(Clk_Cycle)
13     );
14
15     always begin
16         #5 clk = ~clk;
17     end
18
19     initial begin
20         clk = 0;
21         rst = 0;
22         rst = 1;
23         #10;
24         rst = 0;
25         #95;
26
27         $stop;
28     end
29
30     initial begin
31         $monitor("Time = %0t, Clk_Cycle = %b", $time, Clk_Cycle);
32     end
33
34 endmodule
```

RESULT WAVEFORMS

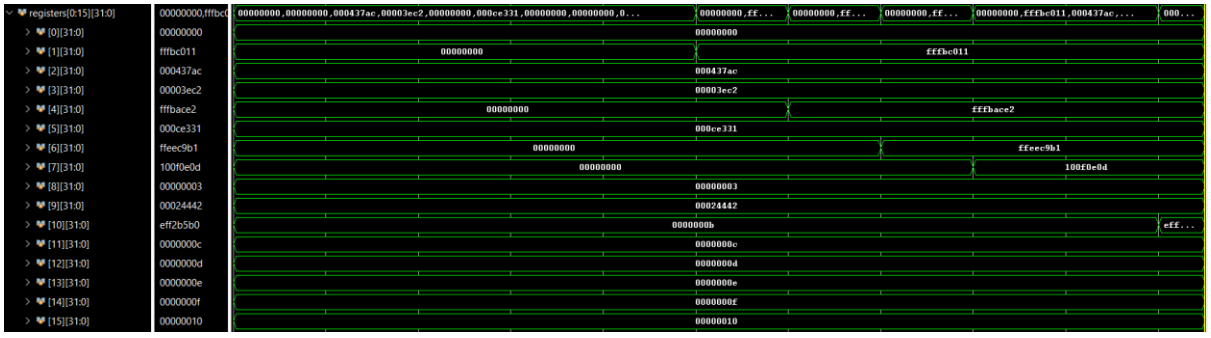
All control signals and clock initialization:



The Register data flow:



The register data:



The data memory: