Computer Architecture
CE/CS-321/330

Final Project Report

RISC V Processor

Group Members
Afeera Umair au08735
Mikaal Imam mi08753
Emaan Naveed Sheikh es09163

Instructor
Saima Shaheen

Contents

1) Introduction

1.1 Objective

The objective of this project is to design and implement a **5-stage pipelined RISC-V processor** that can execute an array sorting algorithm. The project involves converting a previously built **single-cycle processor** into a pipelined one. This pipelined architecture allows instructions to overlap in execution through stages: Fetch, Decode, Execute, Memory, and Write Back.
To test the processor's functionality, we selected the **Bubble Sort algorithm**, a straightforward sorting method well-suited for verifying control flow and data dependencies in the pipeline. All relevant codes can be found in the Appendix at the end of the report.

1.11 Sorting Algorithm

We used the bubble sort algorithm to sort our array. We wrote the code and tested it on https://venus.kvakil.me/

2) Tasks

2.1 Task 1

2.1.1 RISC V Implementation Single Cycle

The algorithm was implemented on the RISC-V single-cycle processor developed in our lab by making some necessary changes. To ensure compatibility with the algorithm, the instruction memory, ALU, and ALU Control were modified. Additionally, the branching mux was adjusted for the jump instructions used in blt.

2.1.2 Changes

The processor implemented in the lab did not work for I-type and SB-type instructions. The following modules were modified to make the bubble sort work:

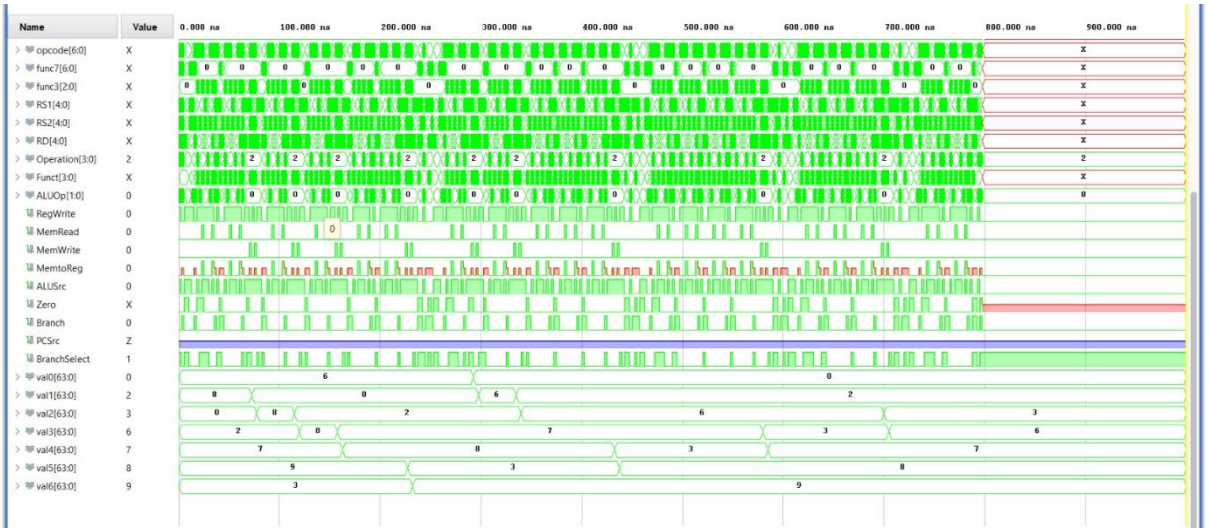
- ALU Control
- Control Unit
- Top level module for RISCV Processor

The slli instruction was added to support left-shifting operations during element comparisons and swaps in Bubble Sort. It is handled within the ALU Control unit when ALUOp = 2'b00 and Funct[2:0] = 4'b001. The blt (branch if less than) instruction is used to enable conditional branching during element comparisons. A branch is triggered if the left element is greater than the right element, prompting a swap.

The ALU Control unit was updated to support both slli and blt, ensuring that proper shifting and branching occur during the sorting process. Support for I-type instructions was also added to handle immediate values and operations.

The Branch Multiplexer was modified to control the flow for the blt instruction, enabling the processor to iterate over the array and perform necessary swaps. Through these modifications, the processor was enabled to efficiently execute Bubble Sort, handling shifts, comparisons, and conditional branching

2.1.3 Waveforms:



The waveform shows the step-by-step sorting of values using Bubble Sort, where initially unsorted values (like 8, 0, 6, etc.) are gradually swapped over time to achieve a sorted order. The values are sorted in ascending order: 0, 2, 6, 7, 8.

2.2 Task 2

2.2.1 Testing 5-Stage/Pipelined RISC V Processor for Sorting Algorithm

In this part, we upgraded our developed processor for Pipeline Execution. We added four new modules and made modifications in three of them.

2.2.2 Changes

To implement pipelining in the processor, new modules were introduced to handle the separation of stages effectively:

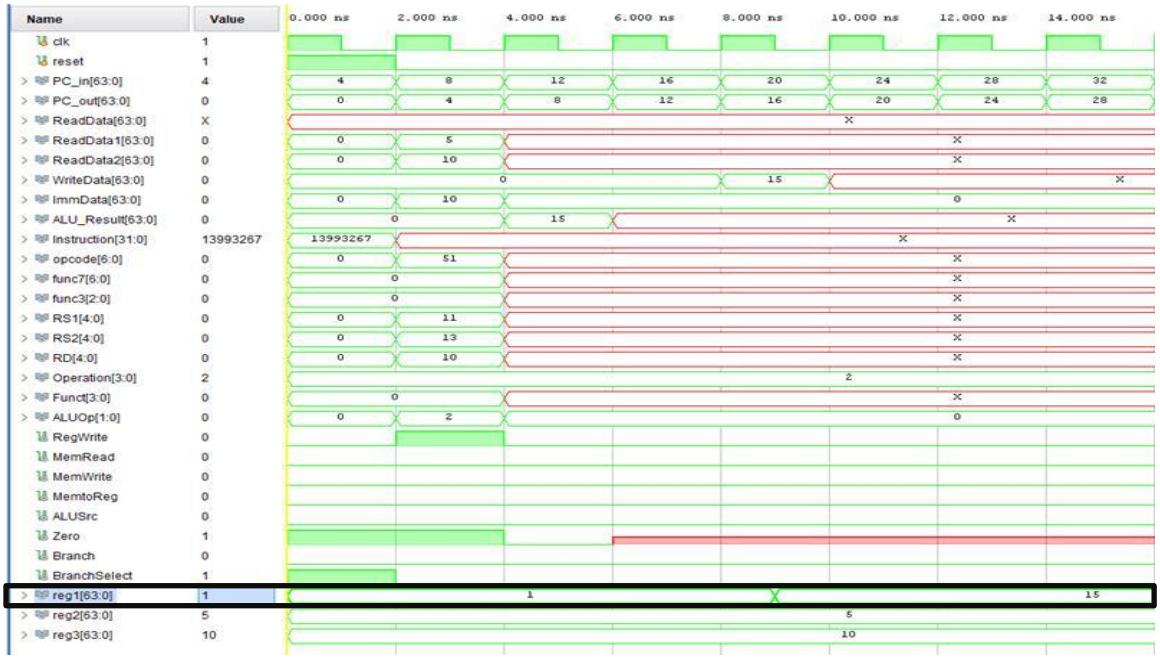
- **IF/ID**
- **ID/EX**
- **EX/MEM**
- **MEM/WB**

These pipeline registers are responsible for transferring both data and control signals between stages, ensuring proper timing and coordination. This structure forms the foundation of the pipelined architecture and helps minimize data hazards. Along with this, updates were made to the **Register File**, **Instruction Memory**, and the main **Processor Logic** to fully support pipelined operation.

2.2.3 Waveforms

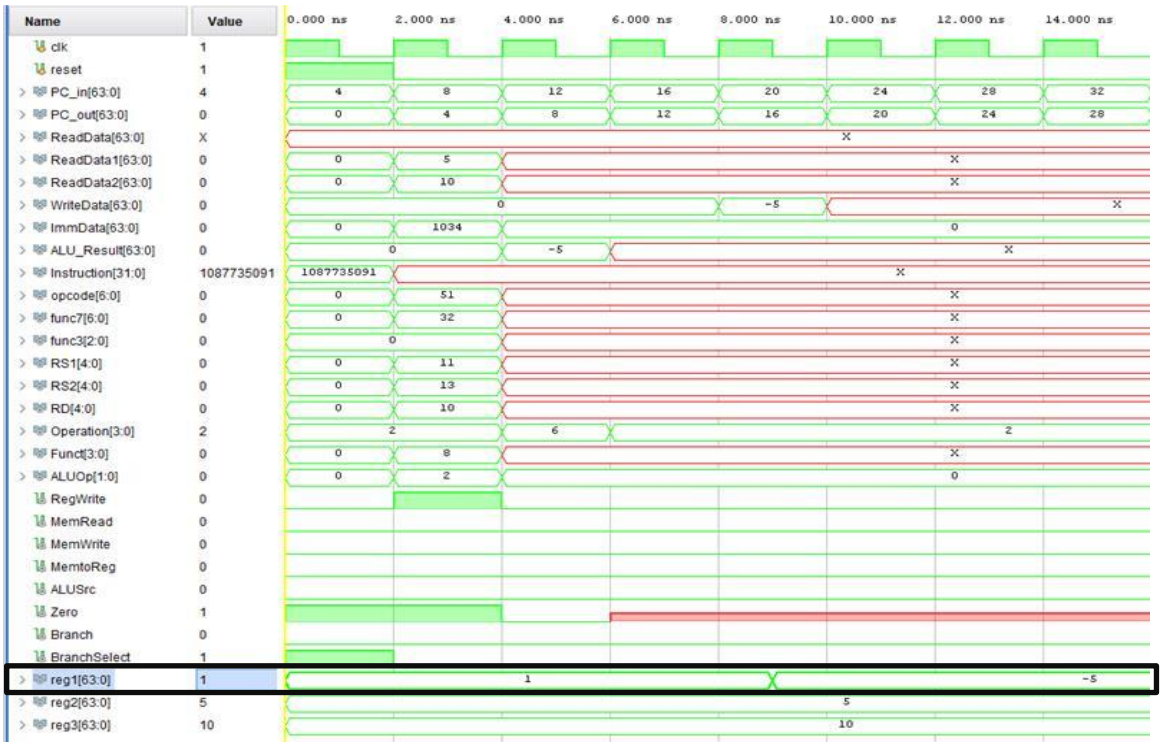
The pipelined processor was tested with individual instructions.

1. **add x10 x11 x13**
   In the waveform below, reg1 = x10, reg2 = x11 and reg3 = x13. The values of reg2 and reg3 are added and stored in reg1 as shown.



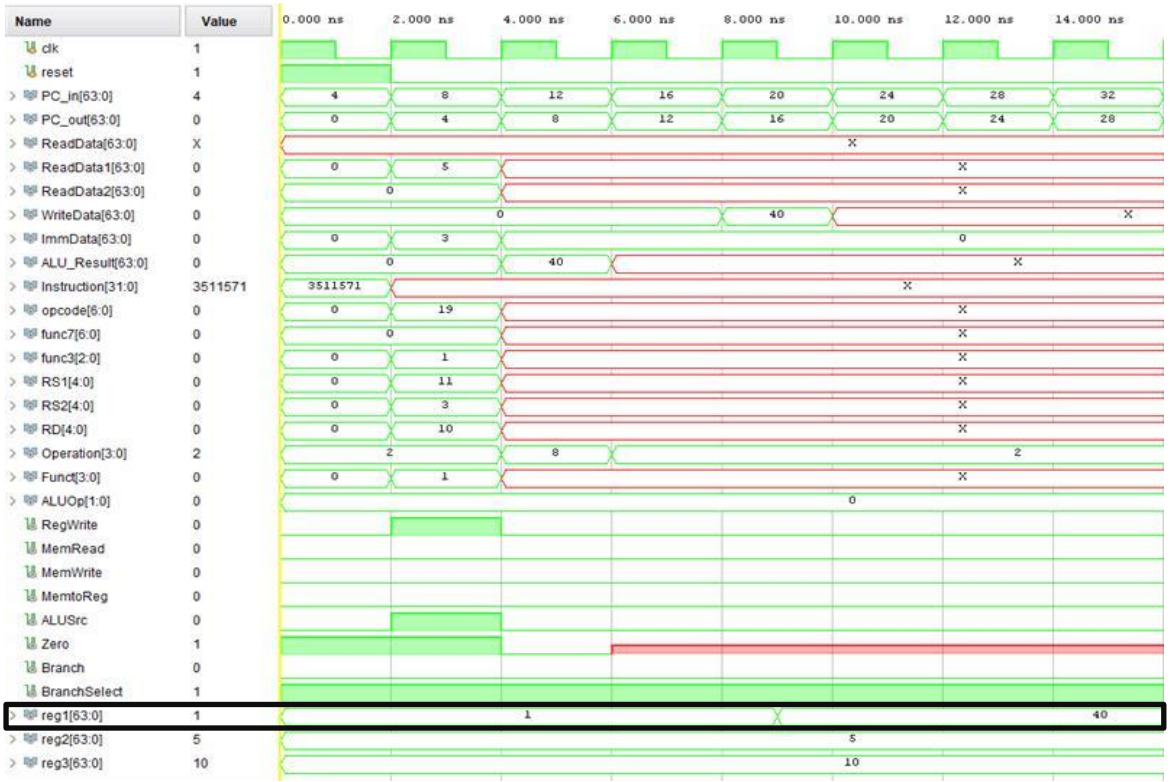
2. **sub x10 x11 x13**

   In the waveform below, reg1 = x10, reg2 = x11 and reg3 = x13. The values of reg2 and reg3 are subtracted and stored in reg1 as shown.



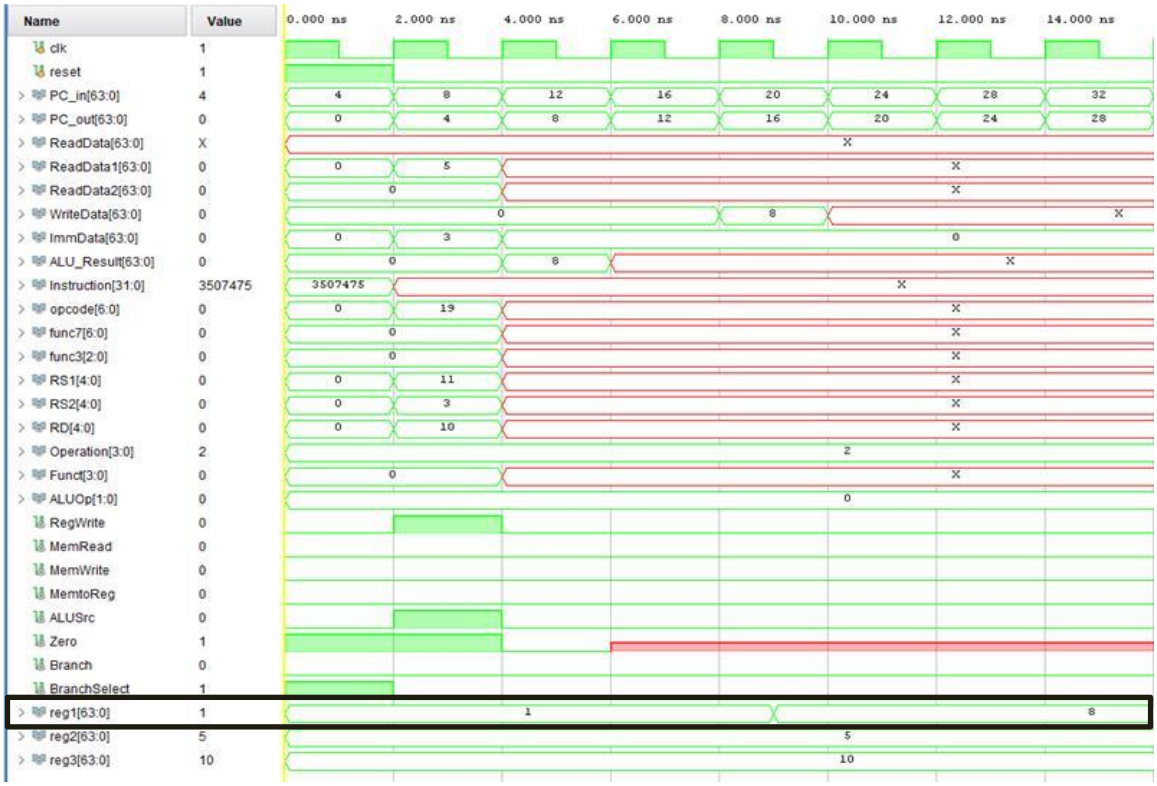
3. **slli x10 x11 3**

   In the waveform below, reg1 = x10, reg2 = x11. The value of reg2 is multiplied by 8 and stored in reg1 as shown.



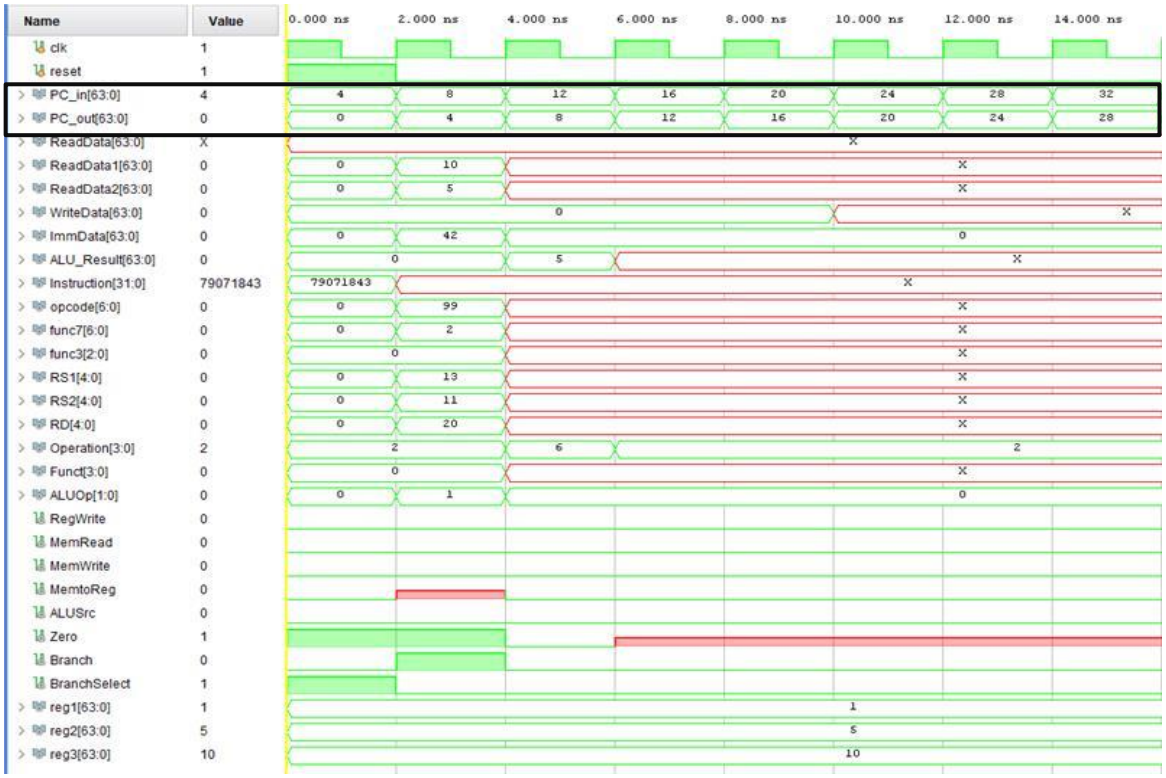
4. addi x10 x11 3

In the waveform below, reg1 = x10, reg2 = x11. The value of reg2 + 3 is stored in reg1 as shown.



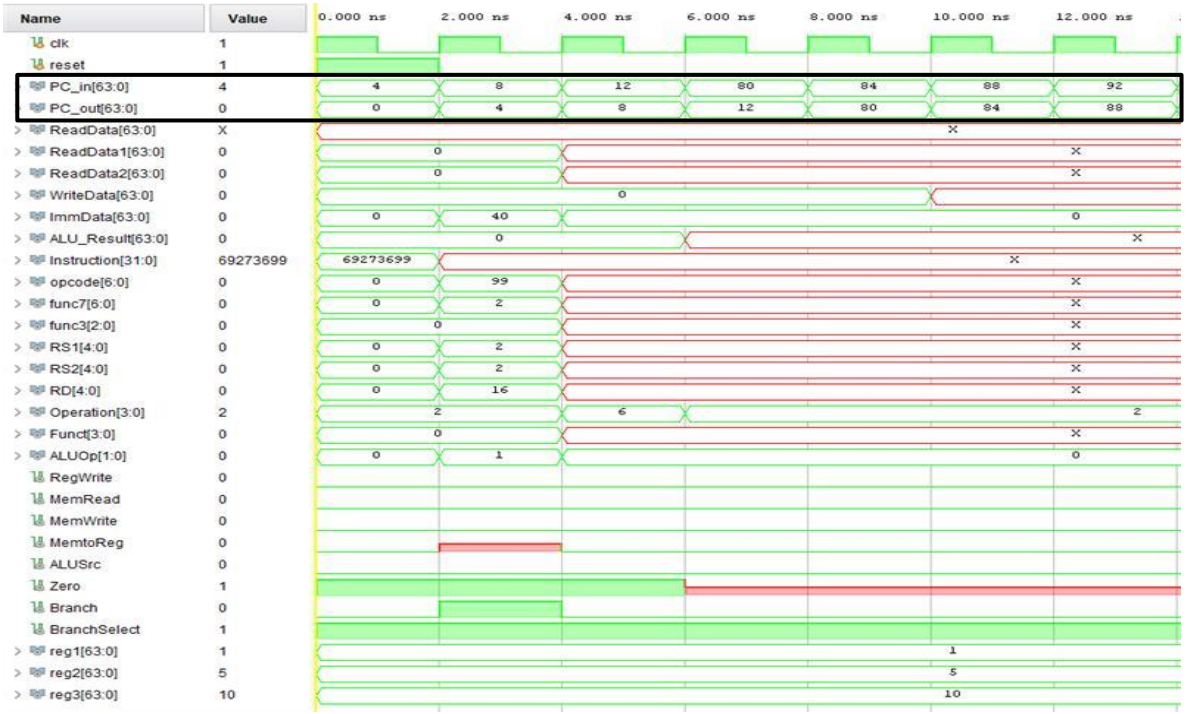
5. beq x13 x11 84 (untaken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the values of reg2 and reg3 are unequal, the branching condition is false, and we observe no change in PC values.



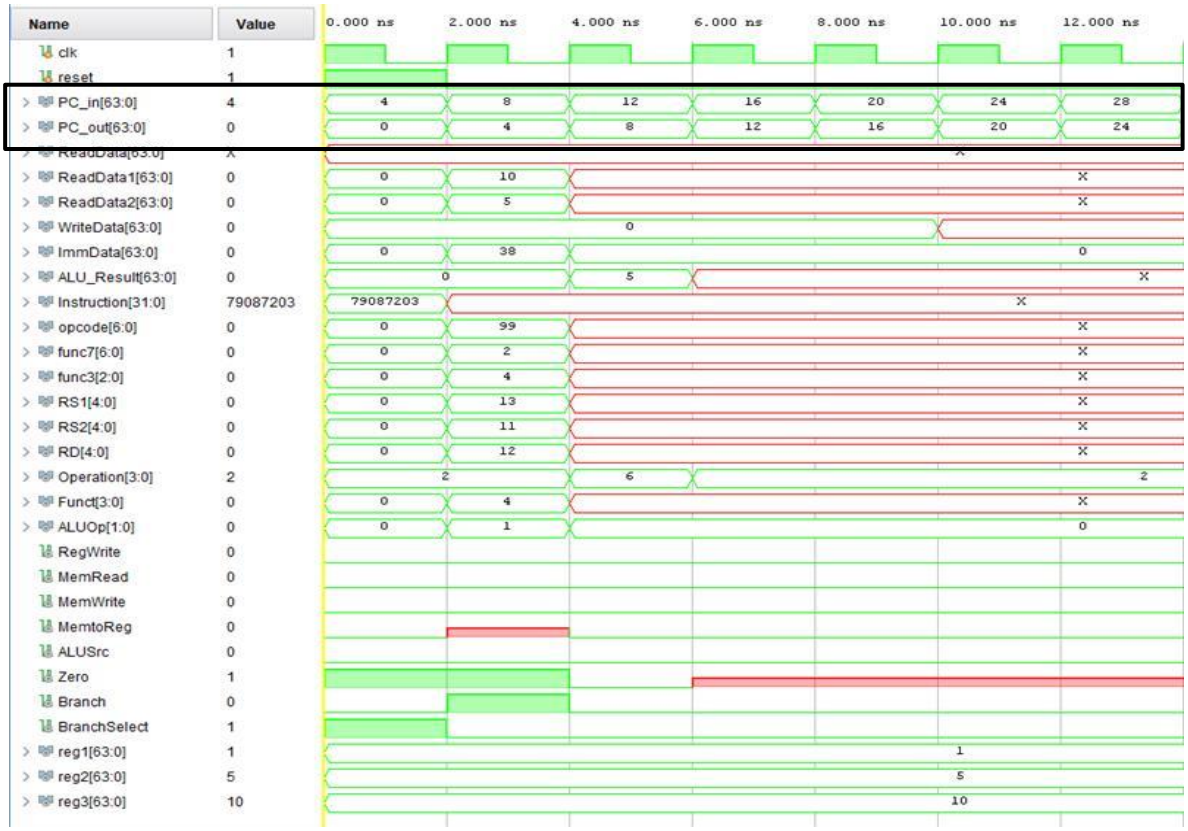
6. beq x2 x2 80 (taken)

The branching condition is true, and we observe a change in PC values.



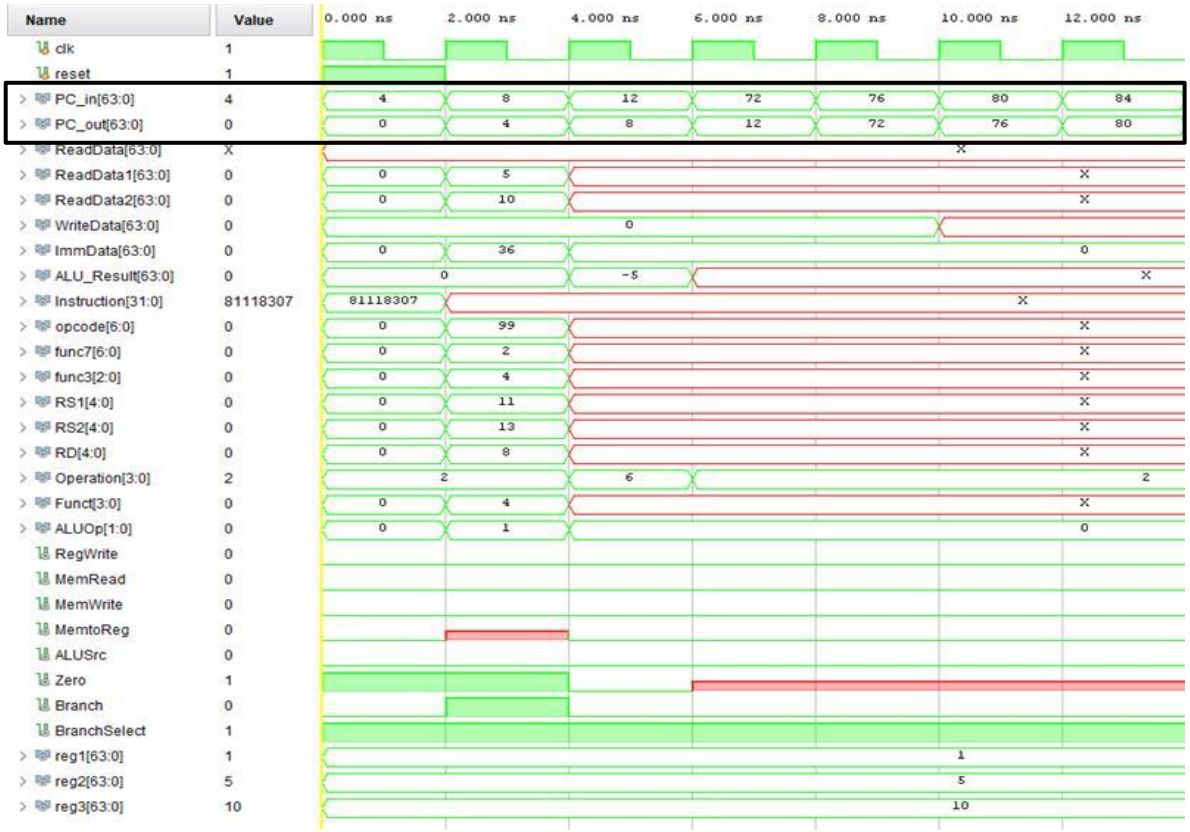
7. blt x13 x11 76 (untaken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the value of reg3 is greater than reg2, the branching condition is false, and we observe no change in PC values.



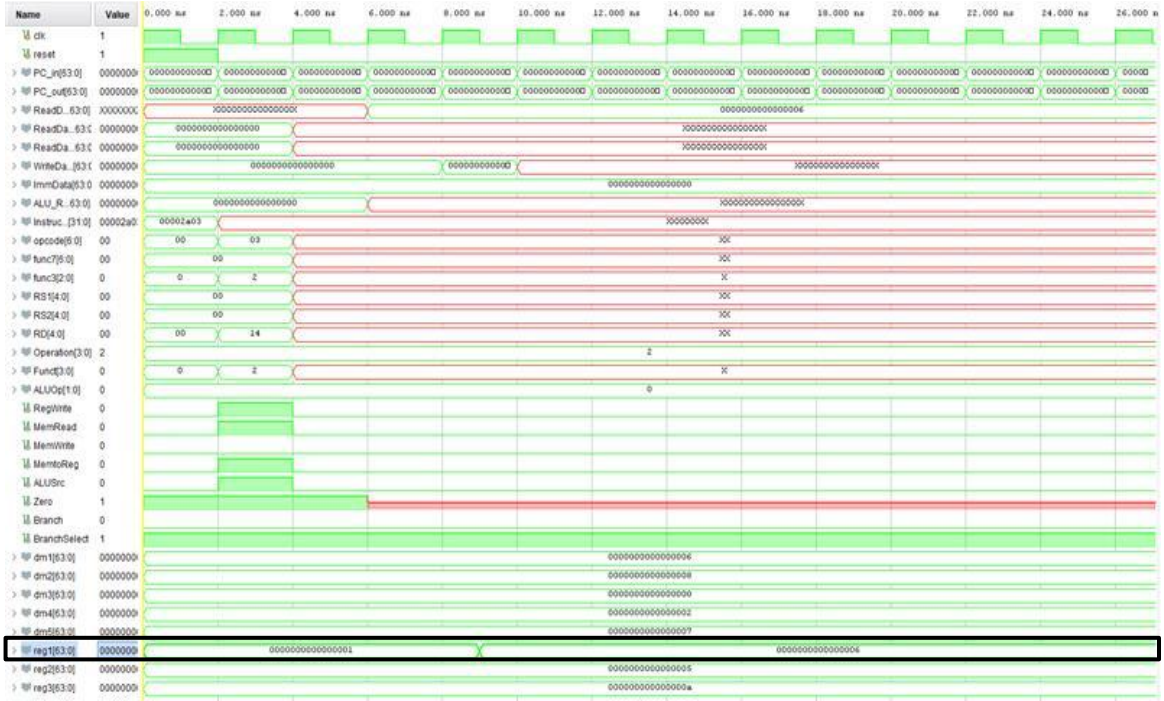
8. blt x11 x13 72 (taken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the value of reg2 is less than reg3, the branching condition is true, and we observe a change in PC values.



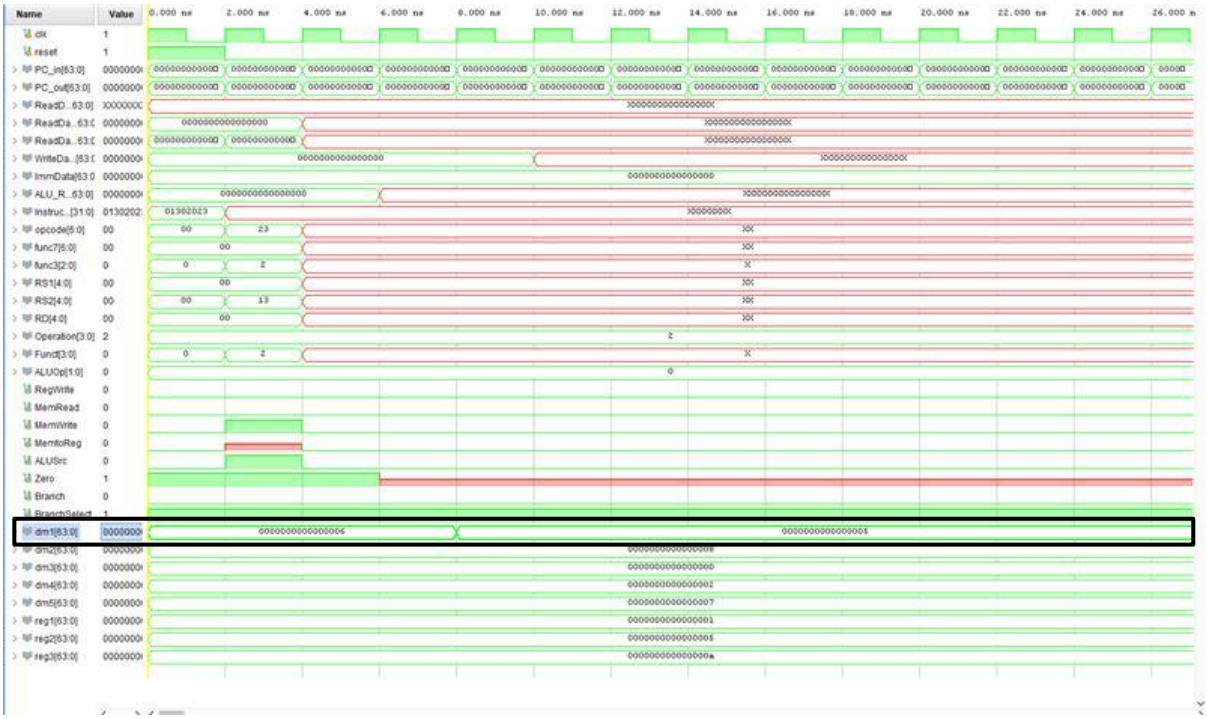
9. lw x20, 0(x0)

The value at 0x0 (dm1) is loaded in reg1(x20).



10. sw x19, 0(x0)

The value at x19(reg2) is stored in dm1(0x0).



2.3 Task 3

2.3.1 Handling Data Hazards

To manage data hazards in our circuit design, we implemented modules that can both detect hazards and control pipeline stalling. These detection units help determine when data forwarding is needed. Once a hazard is identified, the forwarding logic is activated to resolve it.

2.3.2 Changes

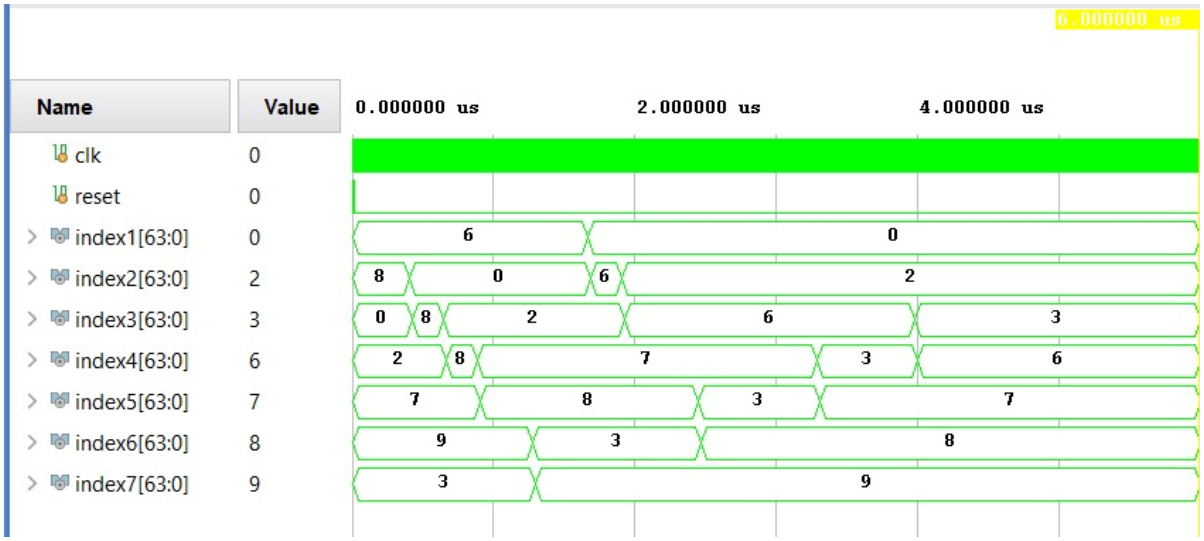
The following modules were added/modified:

- Hazard Detection Unit
- Forwarding Unit

We added a Hazard_Detection module to our processor to manage data hazards, particularly load-use hazards. This module takes IDEX_rd, IFID_rs1, IFID_rs2, and IDEX_MemRead as inputs to detect potential conflicts between consecutive instructions. If a hazard is detected—specifically when the instruction in the ID/EX stage is reading from memory and its destination register matches one of the source registers of the instruction in the IF/ID stage—the module asserts control signals to stall the pipeline.

When such a hazard is detected, the Hazard_Detection module sets IDEX_mux_out, IFID_Write, and PCWrite to 0, which prevents updates to the PC and IF/ID pipeline register and inserts a bubble in the pipeline. Otherwise, all signals are set to 1, allowing normal instruction flow. This addition improves the processor's ability to handle hazards efficiently, ensuring correct execution and enhancing overall performance.

2.3.3 Waveform



2.4 Task 4

2.4.1 Performance Comparison

Single Cycle Processor:

- Clock cycle time = 2 ns
- Execution time = 640 ns
- Clock cycles = 640/2 = 320cc

Pipelined Processor:

- Clock cycle time = 2/5 = 0.4ns
- Execution time = 911 ns
- Clock cycles = 911/0.4 = 2,277.5 cc

Speed up:

- Speedup = Single Cycle Execution Time / Pipelined Execution Time
- Speed up = 640/911 = 0.7

The pipelined processor for our project takes more time than the single cycle processor. It is 0.7 times slower.

3) Challenges

In task 1, our initial bubble sorting code was incorrect. A lot of time was spent on debugging the processor even though our instruction memory was incorrect. We figured out that our instruction memory was wrong and updated it. For task 2, we struggled in making the branch instructions work. Task 3 overall was very complex and writing the top level module took a lot of time and a lot of debugging.

4) Task Division

- Task 1 : Mikaal, Afeera, Emaan
- Task 2: Mikaal, Emaan, Afeera
- Task 3: Mikaal, Afeera
- Task 4: Mikaal, Afeera
- Report: Afeera, Emaan

5) Appendix

A: Bubble Sort Code

```
#assuming all regs are initialised at 0
addi x18 x0 0x100       #base addy
addi x19, x0, 7         # length of the array
outer_loop:
    beq x8, x19, exit   # exit the program
    addi x9, x0, 0      # j = 0
    addi x10, x0, 0     # reseting swapping flag
    addi x6, x19, -1    # Limit for the inner loop
    sub x6, x6, x8      # making the inner loop smaller
inner_loop:
    beq x9, x6, inner_loop_exit
    slli x7, x9, 2      # Multiply i by 4
    add x7, x7, x18     # Calc addr of a[j]
    lw x5, 0(x7)        # Load a[j] into x5
    addi x29, x9, 1     # j+1
    slli x28, x29, 2    # Multiply i+1 by 4
    add x28, x28, x18   # calc addr of a[j+1]
    lw x30, 0(x28)      # load a[j+1]
    bge x5 x30 swap     #branch if a[j] > a[j+1]
    #blt x30, x5, swap   # Branch if a[j+1] < a[j] then swap
    addi x9, x9, 1      # j += 1
    beq x0, x0, inner_loop # next iter inner_loop
swap:
    addi x31, x5, 0     # swapping starts
    addi x5, x30, 0
    sw x5, 0(x7)
    addi x30, x31, 0
    sw x30, 0(x28)
    addi x10, x0, 1     # swapped falg = true
    addi x9, x9, 1
    beq x0, x0, inner_loop # next iter inner_loop
inner_loop_exit:
    addi x8, x8, 1      # i += 1
    beq x10, x0, exit   # if no swapps exit program
    beq x0, x0, outer_loop # next iter outer_loop
exit:
```

B: Codes for Task 1

ALU Control Module:

```
module ALU_Control(ALUOp, Funct, Operation);

input [1:0] ALUOp;
input [3:0] Funct;
output reg [3:0] Operation;

always@ (*)
begin
    case (ALUOp)
        2'b00: // I Type
            begin
            case (Funct[2:0])
                4'b001 :
                    Operation = 4'b1000; // slli
                default:
                   Operation = 4'b0010; // sd n ld
            endcase
            end
        2'b01: // SB Type
            begin
                Operation = 4'b0110; //beq
            end
        2'b10: // R Type
            begin
                case (Funct)
                    4'b0000:
                        Operation = 4'b0010; // add
                    4'b1000:
                        Operation = 4'b0110; // sub
                    4'b0111:
                        Operation = 4'b0000; // and
                    4'b0110:
                        Operation = 4'b0001; // or
                endcase
            end
    endcase
end
endmodule
```

Control Unit :

```
`timescale 1ns / 1ps

module Control_Unit(
    input [6:0] opcode,
    output reg [1:0] ALUOp,
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg MemWrite,
    output reg ALUSrc,
    output reg RegWrite);

always@(*)begin
    case (opcode)
        7'b0110011: // R type
            begin
            ALUSrc = 0;
            MemtoReg = 0;
            RegWrite = 1;
            MemRead = 0;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 2'b10;
            end
        7'b0000011: // I type (ld)
            begin
            ALUSrc = 1;
            MemtoReg = 1;
            RegWrite = 1;
            MemRead = 1;
            MemWrite = 0;
            Branch = 0;
            ALUOp = 2'b00;
            end
        7'b0100011: // S type (sd)
            begin
            ALUSrc = 1;
            MemtoReg = 1'bx;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 1;
            Branch = 0;
            ALUOp = 2'b00;
            end
        7'b1100011: // SB type (beq)
            begin
            ALUSrc = 0;
            MemtoReg = 1'bx;
            RegWrite = 0;
            MemRead = 0;
            MemWrite = 0;
            Branch = 1;
            ALUOp = 2'b01;
            end
        7'b0010011: // I type (addi)
            begin
            ALUSrc = 1;
            MemtoReg = 0;
            RegWrite = 1;
            MemRead = 0;
```

```
                MemWrite = 0;
                Branch = 0;
                ALUOp = 2'b00;
                end
            default: // initializing
                begin
                ALUSrc = 0;
                MemtoReg = 0;
                RegWrite = 0;
                MemRead = 0;
                MemWrite = 0;
                Branch = 0;
                ALUOp = 2'b00;
                end
        endcase
end
endmodule
```

Instruction Memory:

```
`timescale 1ns / 1ps

module Instruction_Memory(
    input [63:0] Inst_Address,
    output reg [31:0] Instruction);

reg [7:0] inst_mem [120:0];

initial
    begin
      {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00500993; // 1
      {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h07340663; // 2
      {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00000493; // 3
      {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00000513; // 4
      {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'hfff98313; // 5
      {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h40830333; // 6
      {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h04648663; // 7
      {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h00349393; // 8
      {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h012383b3; // 9
      {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h0003b283; // 10
      {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00148e93; // 11
      {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h003e9e13; // 12
      {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h012e0e33; // 13
      {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h000e3f03; // 14
      {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h005f4663; // 15
      {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h00148493; // 16
      {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'hfc000ce3; // 17
      {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h00028f93; // 18
      {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h000f0293; // 19
      {inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'h0053b023; // 20
```

```
        {inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h000f8f13; // 21
        {inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h01ee3023; // 22
        {inst_mem[91], inst_mem[90], inst_mem[89], inst_mem[88]} = 32'h00100513; // 23
        {inst_mem[95], inst_mem[94], inst_mem[93], inst_mem[92]} = 32'h00148493; // 24
        {inst_mem[99], inst_mem[98], inst_mem[97], inst_mem[96]} = 32'hfa000ce3; // 25
        {inst_mem[103], inst_mem[102], inst_mem[101], inst_mem[100]} = 32'h00140413; // 26
        {inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]} = 32'h00050463; // 27
        {inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]} = 32'hf8000ce3; // 28
    end

    always @(Inst_Address)begin
        Instruction[31:24] <= inst_mem[Inst_Address + 3];
        Instruction[23:16] <= inst_mem[Inst_Address + 2];
        Instruction[15:8] <= inst_mem[Inst_Address + 1];
        Instruction[7:0] <= inst_mem[Inst_Address];
    end
    endmodule
```

RISC V Single Cycle Processor:

```
`timescale 1ns / 1ps

module RISCV_SCProcessor(
    input clk,
    input reset,
    output [63:0] PC_in, PC_out, ReadData, ReadData1, ReadData2, WriteData, ImmData, Result,
    output [63:0] shifted_data, Data_Out, Out1, Out2,
    output [31:0] Instruction,
    output [6:0] opcode, func7,
    output [2:0] func3,
    output [4:0] RS1, RS2, RD,
    output [3:0] Operation, Funct,
    output [1:0] ALUOp,
    output RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, Zero, Branch, PCSrc, BranchSelect ,
    output [63:0] dm1, dm2, dm3, dm4, dm5
    );


    // Fetching
    Adder FOURADDER(PC_out, 64'd4, Out1);
    Mux_2x1 BRANCH(Out1, Out2, (Branch & BranchSelect), PC_in);
    Program_Counter PC(clk, reset, PC_in, PC_out);
    Instruction_Memory IM(PC_out, Instruction);

    // Decode
    Instruction_Parser IP(Instruction, opcode, RD, func3, RS1, RS2, func7);
    Imm_Gen IG(Instruction, ImmData);
    Control_Unit CU(opcode, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);
    Register_File RF(WriteData, RS1, RS2, RD, RegWrite, clk, reset, ReadData1, ReadData2);
    Branch Unit BU(func3, ReadData1, ReadData2, BranchSelect);

    // Execute
    assign shifted_data = ImmData << 1;
    Adder BRANCHADDER(PC_out, shifted_data, Out2);
    Mux_2x1 ALUSRC(ReadData2, ImmData, ALUSrc, Data_Out);
    assign Funct = {Instruction[30],Instruction[14:12]};
    ALU_Control ALUC(ALUOp, Funct, Operation);
    ALU64 ALU(ReadData1, Data_Out, Operation, Result, Zero);

    // Memory
    Data_Memory DM(Result, ReadData2, clk, MemWrite, MemRead, ReadData, dm1, dm2, dm3, dm4, dm5);

    // Write Back
    Mux_2x1 WB(Result, ReadData, MemtoReg, WriteData);

endmodule
```

C: Codes for Task 2

Pipelined RISC-V Processor:

```
`timescale 1ns / 1ps
module RISC_V_Pipelined(
    input clk,
    input reset,
    output wire [63:0] PC_in, PC_out, ReadData, ReadData1, ReadData2, WriteData, ImmData, ALU_Result,
    output wire [31:0] Instruction,
    output wire [6:0] opcode, func7,
    output wire [2:0] func3,
    output wire [4:0] RS1, RS2, RD,
    output wire [3:0] Operation, Funct,
    output wire [1:0] ALUOp,
    output wire RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, Zero, Branch, BranchSelect, // PCSrc
    output wire [63:0] dm1, dm2, dm3, dm4, dm5,
    output wire [63:0] reg1, reg2, reg3);

    // Single Cycle wires
    wire [63:0] shifted_data, Data_Out, Out1, Adder_out;
    //wire [63:0] dm1, dm2, dm3, dm4, dm5;

    // IF_ID wires
    wire [63:0] IF_ID_PC_out;
    wire [31:0] IF_ID_Instruction;

    // ID_EX wires
    wire ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUSrc;
    wire [1:0] ID_EX_ALUOp;
    wire [63:0] ID_EX_PC_out, ID_EX_ReadData1, ID_EX_ReadData2, ID_EX_ImmData;
    wire [3:0] ID_EX_Funct;
    wire [4:0] ID_EX_RS1, ID_EX_RS2, ID_EX_RD ;

    // EX_MEM wires
    wire  EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemtoReg, EX_MEM_MemWrite, EX_MEM_Branch, EX_MEM_Zero;
    wire [4:0] EX_MEM_RD;
    wire [63:0] EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_ReadData2;

    // MEM_WB wires
    wire MEM_WB_RegWrite, MEM_WB_MemtoReg;
    wire [4:0] MEM_WB_RD;
    wire [63:0] MEM_WB_ReadData, MEM_WB_ALU_Result;

    // IF Stage
    Adder FOURADDER(64'd4, PC_out, Out1);
     Mux_2x1 BRANCH(Out1, EX_MEM_Adder_out, (EX_MEM_Branch & EX_MEM_Zero), PC_in);
    Program_Counter PC(clk, reset, PC_in, PC_out);
    Instruction_Memory_Pipelined IMP(PC_out, Instruction);

    IF_ID Pipeline1(clk, reset, PC_out, Instruction, IF_ID_PC_out, IF_ID_Instruction);

    // ID Stage
    assign Funct = {IF_ID_Instruction[30],IF_ID_Instruction[14:12]};
    Instruction_Parser IP(IF_ID_Instruction, opcode, RD, func3, RS1, RS2, func7);
    Imm_Gen IG(IF_ID_Instruction, ImmData);
    Control_Unit CU(opcode, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);
    register_file_pipelined RF(WriteData, RS1, RS2, MEM_WB_RD, MEM_WB_RegWrite, clk, reset, ReadData1, ReadData2,reg1,reg2,reg3);
    Branch_Unit BU(func3, ReadData1, ReadData2, BranchSelect);

    ID_EX Pipeline2(clk, reset,
            RegWrite, MemRead, MemtoReg, MemWrite, Branch, ALUOp, ALUSrc, // control signals
            IF_ID_PC_out, ReadData1, ReadData2, ImmData, RS1, RS2, RD, Funct, // inputs
            ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUOp, ID_EX_ALUSrc, // output control signals
```

```
            ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUOp, ID_EX_ALUSrc, // output control signals
            ID_EX_PC_out, ID_EX_ReadData1, ID_EX_ReadData2, ID_EX_ImmData, ID_EX_RS1, ID_EX_RS2, ID_EX_RD, ID_EX_Funct); // ouputs

    // EXE Stage
    assign shifted_data = ID_EX_ImmData << 1;
    Adder BRANCHADDER(ID_EX_PC_out, shifted_data, Adder_out);
    Mux_2x1 MuxALUSrc(ID_EX_ReadData2, ID_EX_ImmData, ID_EX_ALUSrc, Data_Out);
    ALU_Control ALUC(ID_EX_ALUOp, ID_EX_Funct, Operation);
    ALU64 ALU(ID_EX_ReadData1, Data_Out, Operation, ALU_Result, Zero);

    EX_MEM Pipeline3(clk, reset,
            ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, // input control signals
            Adder_out, ALU_Result, BranchSelect, ID_EX_ReadData2, ID_EX_RD,  // inputs
            EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemtoReg, EX_MEM_MemWrite, EX_MEM_Branch, // output control signals
            EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_Zero, EX_MEM_ReadData2, EX_MEM_RD // outputs
            );

    // MEM Stage
    Data_Memory DM(EX_MEM_ALU_Result, EX_MEM_ReadData2, clk, EX_MEM_MemWrite, EX_MEM_MemRead, ReadData,
    dm1, dm2, dm3, dm4, dm5);

    MEM_WB Pipeline4(clk, reset,
            EX_MEM_RegWrite, EX_MEM_MemtoReg, // input control signals
            ReadData, EX_MEM_ALU_Result, EX_MEM_RD, // inputs
            MEM_WB_RegWrite, MEM_WB_MemtoReg, // output control signals
            MEM_WB_ReadData, MEM_WB_ALU_Result, MEM_WB_RD  // outputs
            );

    // WB
    Mux 2x1 WB(MEM WB ALU Result, MEM WB ReadData, MEM WB MemtoReg, WriteData);
```

Pipelined Register File:

```
`timescale 1ns / 1ps
module register_file_pipelined(
    input clk, reset, RegWrite,
    input [63:0] WriteData,
    input [4:0] RS1, RS2, RD,
    output reg [63:0] ReadData1, ReadData2,
    output [63:0] reg1, reg2, reg3);


reg [63:0] Registers [31:0];
integer k;
initial begin
    for (k = 0 ; k < 31 ; k = k + 1)
        Registers[k] = 0;
        Registers[10] = 64'd1;
        Registers[11] = 64'd5;
        Registers[13] = 64'd10;
end
assign reg1 = Registers[10];
assign reg2 = Registers[11];
assign reg3 = Registers[13];

always @(negedge clk) begin
    if (RegWrite) begin //writing data to memory
        Registers[RD] <= WriteData;
    end
end


always @(*) begin
 always @(*) begin
     if (reset)begin //resseting
         ReadData1 = 0;
         ReadData2 = 0;
     end
     else begin //reading datafrom mem into rs1 read data 1 and 2
         ReadData1 <= Registers[RS1];
         ReadData2 <= Registers[RS2];
     end
 end
 endmodule
```

IF/ID:

```
`timescale 1ns / 1ps

module IF_ID(
    input clk, reset,
    input [31:0] Instruction,
    input [63:0] PC_out,
    output reg [31:0] IF_ID_Instruction,
    output reg [63:0] IF_ID_PC_out);

always @ (posedge clk)
begin
    case (reset)
        1'b1:
            begin //resseting
            IF_ID_Instruction <= 0;
            IF_ID_PC_out <= 0;
            end
        1'b0:
            begin //taking the instruction and giving it to IF_ID
            IF_ID_Instruction <= Instruction;
            IF_ID_PC_out <= PC_out;
            end
    endcase
end
endmodule
```

ID/EX:

```
`timescale 1ns / 1ps
module ID_EX (
    input clk, reset, RegWrite, MemRead, MemToReg, MemWrite, Branch, ALUSrc,
    input [1:0] ALUOp,
    input [63:0] IF_ID_PC_out, ReadData1, ReadData2, ImmData,
    input [3:0] Funct,
    input [4:0] RS1, RS2, RD,
    output reg ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemToReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUSrc,
    output reg [1:0] ID_EX_ALUOp,
    output reg [63:0] ID_EX_PC_out, ID_EX_ReadData1, ID_EX_ReadData2, ID_EX_ImmData,
    output reg [3:0] ID_EX_Funct,
    output reg [4:0] ID_EX_RS1, ID_EX_RS2, ID_EX_RD );


always @(posedge clk) begin
    case (reset)
        1'b1:
        begin //resseting
            ID_EX_RegWrite <= 0;
            ID_EX_MemRead <= 0;
            ID_EX_MemToReg <= 0;
            ID_EX_MemWrite <= 0;
            ID_EX_Branch <= 0;
            ID_EX_ALUSrc <= 0;
            ID_EX_ALUOp <= 0;
            ID_EX_PC_out <= 0;
            ID_EX_ReadData1 <= 0;
            ID_EX_ReadData2 <= 0;
            ID_EX_ImmData <= 0;
            ID_EX_Funct <= 0;
```

```
            ID_EX_RS1 <= 0;
            ID_EX_RS2 <= 0;
            ID_EX_RD <= 0;
        end

        1'b0:
        begin //decoding the instruction and giving it to ID_EX
            ID_EX_RegWrite <= RegWrite;
            ID_EX_MemRead <= MemRead;
            ID_EX_MemToReg <= MemToReg;
            ID_EX_MemWrite <= MemWrite;
            ID_EX_Branch <= Branch;
            ID_EX_ALUSrc <= ALUSrc;
            ID_EX_ALUOp <= ALUOp;
            ID_EX_PC_out <= IF_ID_PC_out;
            ID_EX_ReadData1 <= ReadData1;
            ID_EX_ReadData2 <= ReadData2;
            ID_EX_ImmData <= ImmData;
            ID_EX_Funct <= Funct;
            ID_EX_RS1 <= RS1;
            ID_EX_RS2 <= RS2;
            ID_EX_RD <= RD;
        end
    endcase

end
endmodule
```

EX/MEM:

```
`timescale 1ns / 1ps
module EX_MEM(
    input clk, reset, ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemToReg, ID_EX_MemWrite, ID_EX_Branch, Zero,
    input [4:0] ID_EX_RD,
    input [63:0] Adder_out, ALU_Result, ID_EX_ReadData2,
    output reg  EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemToReg, EX_MEM_MemWrite, EX_MEM_Branch, EX_MEM_Zero,
    output reg [4:0] EX_MEM_RD,
    output reg [63:0] EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_ReadData2);

always @(posedge clk) begin
    case (reset)
        1'b1:
        begin //resseting
            EX_MEM_RegWrite <= 0;
            EX_MEM_MemRead <= 0;
            EX_MEM_MemToReg <= 0;
            EX_MEM_MemWrite <= 0;
            EX_MEM_Branch <= 0;
            EX_MEM_Zero <= 0;
            EX_MEM_RD <= 0;
            EX_MEM_Adder_out <= 0;
            EX_MEM_ALU_Result <= 0;
            EX_MEM_ReadData2 <= 0;
        end
        1'b0:
        begin //takes data from ID_EX stage and gives it to EX_MEM stage
            EX_MEM_RegWrite <= ID_EX_RegWrite;
            EX_MEM_MemRead <= ID_EX_MemRead;
            EX_MEM_MemToReg <= ID_EX_MemToReg;
            EX_MEM_MemWrite <= ID_EX_MemWrite;
```

```
            EX_MEM_Branch <= ID_EX_Branch;
            EX_MEM_Zero <= Zero;
            EX_MEM_RD <= ID_EX_RD;
            EX_MEM_Adder_out <= Adder_out;
            EX_MEM_ALU_Result <= ALU_Result;
            EX_MEM_ReadData2 <= ID_EX_ReadData2;
        end
    endcase

end
endmodule
```

MEM/WB:

```
`timescale 1ns / 1ps
module MEM_WB(
    input clk, reset, EX_MEM_RegWrite, EX_MEM_MemToReg,
    input [4:0] EX_MEM_RD,
    input [63:0] ReadData, EX_MEM_ALU_Result,
    output reg MEM_WB_RegWrite, MEM_WB_MemToReg,
    output reg [4:0] MEM_WB_RD,
    output reg [63:0] MEM_WB_ReadData, MEM_WB_ALU_Result);

always @(posedge clk) begin
    case(reset)
          1'b1:
            begin //resseting
                MEM_WB_RegWrite <= 0;
                MEM_WB_MemToReg <= 0;
                MEM_WB_ReadData <= 0;
                MEM_WB_ALU_Result <= 0;
                MEM_WB_RD <= 0;
            end

          1'b0:
            begin //takes data changed from EX_MEM to MEM_WB
                MEM_WB_RegWrite <= EX_MEM_RegWrite;
                MEM_WB_MemToReg <= EX_MEM_MemToReg;
                MEM_WB_ReadData <= ReadData;
                MEM_WB_ALU_Result <= EX_MEM_ALU_Result;
                MEM_WB_RD <=  EX_MEM_RD;
            end
   endcase

```

D: Codes for Task 3

RISC V Processor

```
23      module RISCV_PP_forwarding(
24          input clk,reset,
25          output[63:0] PC_to_INSTMEM,
26          output [31:0] INSTMEM_to_IF_ID,
27          output [63:0] element1,element2,element3,element4,element5,element6,element7);
28
29      wire [1:0] forwardA, forwardB;
30      //wire [63:0] PC_to_INSTMEM;
31      //wire [31:0] INSTMEM_to_IF_ID;
32      wire [6:0] CONTROL_IN;
33      wire[4:0] rd;
34      wire [2:0] funct3;
35      wire [6:0] funct7;
36      wire [4:0] rs1, rs2;
37      wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
38      wire Is_Greater;
39      wire [1:0] ALUOp;
40      wire [63:0] mux_to_reg;
41      wire [63:0] mux_to_pc_in;
42      wire [3:0] ALU_C_Operation;
43      wire [63:0] ReadData1, ReadData2;
44      wire [63:0] imm_data;
45
46
47
48      wire [63:0] fixed_4 = 64'd4;
49      wire [63:0] PC_plus_4_to_mux;
50
51      wire [63:0] alu_mux;
52
53      wire [63:0] alu_result;
54      wire zero;
55
56      wire [63:0] imm_to_adder;
57      wire [63:0] imm_adder_to_mux;
58
59      wire [63:0] DM_Read_Data;
60
61      wire pc_mux_sel_wire;
62      wire PCWrite;
```

```
    //IF_ID WIRES
    wire [63:0] IF_ID_PC_addr;
    wire [31:0] IF_ID_IM_to_parse;
    wire IF_ID_Write;

    //ID_EX WIRES

    wire ID_EX_Branch, ID_EX_MemRead, ID_EX_MemtoReg;
    wire ID_EX_MemWrite, ID_EX_ALUSrc, ID_EX_RegWrite;

    wire [63:0] ID_EX_PC_addr, ID_EX_ReadData1, ID_EX_ReadData2,
                ID_EX_imm_data;
    wire [3:0] ID_EX_funct_in;
    wire [4:0] ID_EX_rd, ID_EX_rs1, ID_EX_rs2;
    wire [1:0] ID_EX_ALUOp;

    assign imm_to_adder = ID_EX_imm_data<< 1;


    //EX_MEM WIRES
    wire EX_MEM_Branch, EX_MEM_MemRead, EX_MEM_MemtoReg;
    wire EX_MEM_MemWrite, EX_MEM_RegWrite;
    wire EX_MEM_zero, EX_MEM_Is_Greater;
    wire [63:0] EX_MEM_PC_plus_imm, EX_MEM_alu_result, EX_MEM_ReadData2;
    wire [3:0] EX_MEM_funct_in;
    wire [4:0] EX_MEM_rd;
    wire NOP_Check;

    //MEM_WB WIRES
    wire MEM_WB_MemtoReg, MEM_WB_RegWrite;
    wire [63:0] MEM_WB_DM_Read_Data, MEM_WB_alu_result;
    wire [4:0] MEM_WB_rd;


    Mux_2x1 pcsrcmux(EX_MEM_PC_plus_imm, PC_plus_4_to_mux, pc_mux_sel_wire, mux_to_pc_in);

    PC_for_forwarding PC (clk, reset,PCWrite, mux_to_pc_in,PC_to_INSTMEM);

    Adder pcadder(PC_to_INSTMEM, fixed_4, PC_plus_4_to_mux);

Instruction_mem_forwarding insmem(PC_to_INSTMEM, INSTMEM_to_IF_ID);


//IF_ID STAGE
IF_ID_forwarding IFIDreg(.clk(clk), .Flush(NOP_Check), .IFID_Write(IF_ID_Write), .PC_addr(PC_to_INSTMEM),
              .Instruc(INSTMEM_to_IF_ID), .PC_store(IF_ID_PC_addr), .
              Instr_store(IF_ID_IM_to_parse));

wire control_mux_sel;

Hazard_Detection hazarddetectionunit(ID_EX_rd, rs1, rs2, ID_EX_MemRead, control_mux_sel,
                                     IF_ID_Write, PCWrite);



Instruction_Parser insparser(IF_ID_IM_to_parse, CONTROL_IN, rd, funct3, rs1, rs2, funct7);

wire [3:0] funct_in;
//makes a ALU OP
assign funct_in = {IF_ID_IM_to_parse[30],IF_ID_IM_to_parse[14:12]};

Control_Unit cunit(CONTROL_IN, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);


Register_File registerfiles(mux_to_reg, rs1, rs2, MEM_WB_rd, MEM_WB_RegWrite, clk, reset,
                            ReadData1, ReadData2);


Imm_Gen immgen(IF_ID_IM_to_parse, imm_data);

wire MemtoReg_ID_EXin, RegWrite_ID_EXin, Branch_ID_EXin, MemWrite_ID_EXin, MemRead_ID_EXin, ALUSrc_ID_EXin;

//if a hazard is detected then stop taking in instructions
assign MemtoReg_ID_EXin = control_mux_sel ? MemtoReg : 0;
assign RegWrite_ID_EXin = control_mux_sel ? RegWrite : 0;
assign Branch_ID_EXin = control_mux_sel ? Branch : 0;
assign MemWrite_ID_EXin = control_mux_sel ? MemWrite : 0;
assign MemRead_ID_EXin = control_mux_sel ? MemRead : 0;
assign ALUSrc_ID_EXin = control_mux_sel ? ALUSrc : 0;
wire [1:0] ALUop_ID_EXin;
```

```
    wire [1:0] ALUop_ID_EXin;
    assign ALUop_ID_EXin = control_mux_sel ? ALUOp : 2'b00;


    // ID/EX STAGE
    ID_EX_new ID_EX1(.clk(clk), .Flush(NOP_Check), .PC_addr(IF_ID_PC_addr), .read_data1(ReadData1),
                .read_data2(ReadData2), .imm_val(imm_data), .funct_in(funct_in), .rd_in(rd),
                .rs1_in(rs1), .rs2_in(rs2), .RegWrite(RegWrite_ID_EXin),
                .MemtoReg(MemtoReg_ID_EXin), .Branch(Branch_ID_EXin),
                .MemWrite(MemWrite_ID_EXin), .MemRead(MemRead_ID_EXin), .ALUSrc(ALUSrc_ID_EXin),
                .ALU_op(ALUop_ID_EXin), .PC_addr_store(ID_EX_PC_addr),
                .read_data1_store(ID_EX_ReadData1), .read_data2_store(ID_EX_ReadData2),
                .imm_val_store(ID_EX_imm_data), .funct_in_store(ID_EX_funct_in),
                .rd_in_store(ID_EX_rd), .rs1_in_store(ID_EX_rs1), .rs2_in_store(ID_EX_rs2),
                .RegWrite_store(ID_EX_RegWrite), .MemtoReg_store(ID_EX_MemtoReg),
                .Branch_store(ID_EX_Branch), .MemWrite_store(ID_EX_MemWrite),
                .MemRead_store(ID_EX_MemRead), .ALUSrc_store(ID_EX_ALUSrc),
                .ALU_op_store(ID_EX_ALUOp));

    ALU_Control ALU_Control1(ID_EX_ALUOp, ID_EX_funct_in, ALU_C_Operation);

    wire [63:0] triplemux_to_a, triplemux_to_b;

    Mux_2x1 ALU_mux(ID_EX_imm_data, triplemux_to_b, ID_EX_ALUSrc, alu_mux);



    Mux_3x1 mux_for_a(ID_EX_ReadData1, mux_to_reg, EX_MEM_alu_result, forwardA, triplemux_to_a);

    Mux_3x1 mux_for_b(ID_EX_ReadData2, mux_to_reg, EX_MEM_alu_result, forwardB, triplemux_to_b);

    ALU64 ALU_64(triplemux_to_a, alu_mux, ALU_C_Operation, alu_result, zero, Is_Greater);



    Forwarding_Unit Fwd_unit(EX_MEM_rd, MEM_WB_rd, ID_EX_rs1, ID_EX_rs2, EX_MEM_RegWrite,
                             EX_MEM_MemtoReg, MEM_WB_RegWrite, forwardA, forwardB);


    wire [63:0] pc_add_imm_to_EX_MEM;

    Adder PC_plus_imm(ID_EX_PC_addr, imm_to_adder, pc_add_imm_to_EX_MEM);

    // EX/MEM STAGE

    EX_MEM_new EX_MEM1(.clk(clk),.Flush(NOP_Check),.RegWrite(ID_EX_RegWrite),.MemtoReg(ID_EX_MemtoReg),
                .Branch(ID_EX_Branch),.Zero(zero),.Is_Greater(Is_Greater),
                .MemWrite(ID_EX_MemWrite),.MemRead(ID_EX_MemRead),.PCplusimm(pc_add_imm_to_EX_MEM),
                .ALU_result(alu_result),.WriteData(triplemux_to_b),.funct_in(ID_EX_funct_in),
                .rd(ID_EX_rd),.RegWrite_store(EX_MEM_RegWrite),.MemtoReg_store(EX_MEM_MemtoReg),
                .Branch_store(EX_MEM_Branch),.Zero_store(EX_MEM_zero),
                .Is_Greater_store(EX_MEM_Is_Greater),.MemWrite_store(EX_MEM_MemWrite),
                .MemRead_store(EX_MEM_MemRead),.PCplusimm_store(EX_MEM_PC_plus_imm),
                .ALU_result_store(EX_MEM_alu_result),.WriteData_store(EX_MEM_ReadData2),
                .funct_in_store(EX_MEM_funct_in),.rd_store(EX_MEM_rd));


    Branch_Control Branch_Control(.Branch(EX_MEM_Branch), .Flush(NOP_Check), .Zero(EX_MEM_zero), .Is_Greater(EX_MEM_Is_Greater),
                                  .funct(EX_MEM_funct_in),.switch_branch(pc_mux_sel_wire));


    Data_Memory dm(EX_MEM_alu_result, EX_MEM_ReadData2, clk,EX_MEM_MemWrite,EX_MEM_MemRead,
        DM_Read_Data, element1,element2,element3,element4,element5,element6,element7);


    // MEM/WB STAGE

    MEM_WB_new MEM_WB1(.clk(clk), .RegWrite(EX_MEM_RegWrite), .MemtoReg(EX_MEM_MemtoReg),
                .ReadData(DM_Read_Data), .ALU_result(EX_MEM_alu_result), .rd(EX_MEM_rd),
                .RegWrite_store(MEM_WB_RegWrite), .MemtoReg_store(MEM_WB_MemtoReg),
                .ReadData_store(MEM_WB_DM_Read_Data), .ALU_result_store(MEM_WB_alu_result),
                .rd_store(MEM_WB_rd));


    Mux_2x1 mux2(MEM_WB_DM_Read_Data, MEM_WB_alu_result, MEM_WB_MemtoReg, mux_to_reg);


    endmodule
```

Hazard Detection Unit:

```
module Hazard_Detection
(
    input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
    input IDEX_MemRead,
    output reg IDEX_mux_out,
    output reg IFID_Write, PCWrite
);

always@(*) begin

    if (IDEX_MemRead && (IDEX_rd == IFID_rs1 || IDEX_rd == IFID_rs2))
    begin //assigning ZERO value means it is deasserted and the values for PC and Instruction won't be updated
        IDEX_mux_out = 0;
        IFID_Write = 0;
        PCWrite = 0;
    end
    else begin
        IDEX_mux_out = 1;
        IFID_Write = 1;
        PCWrite = 1;
    end
end
endmodule // Hazard_Detection
```

Forwarding Unit

```
module Forwarding_Unit
(
    input [4:0] EXMEM_rd, MEMWB_rd,
    input [4:0] IDEX_rs1, IDEX_rs2,
    input EXMEM_RegWrite, EXMEM_MemtoReg,
    input MEMWB_RegWrite,
    output reg [1:0] fwd_A, fwd_B
);

always @(*) begin

    if (EXMEM_rd == IDEX_rs1 && EXMEM_RegWrite && EXMEM_rd != 0)
        begin
            fwd_A = 2'b10;
        end
    else if (MEMWB_RegWrite && MEMWB_rd!=0 && MEMWB_rd==IDEX_rs1 )
        begin
            fwd_A = 2'b01;
        end
    else
        begin
            fwd_A = 2'b00;
        end


    if ((EXMEM_rd == IDEX_rs2) && (EXMEM_RegWrite) && (EXMEM_rd != 0))
        begin
            fwd_B = 2'b10;
        end
```

input [4:0]

```
    else if (MEMWB_RegWrite && MEMWB_rd!=0 && MEMWB_rd==IDEX_rs2)
        begin
            fwd_B = 2'b01;
        end

    else
        begin
            fwd_B = 2'b00;
        end
end

endmodule // Forwarding_Unit
```