RISC-V Pipelined Processor

CA Final Report

Submitted By:

Mahrukh Yousuf (08055)

Ayesha Eiman (08013)

Arsal Jangda (08514)

Research Assistant:

Saima Shaheen

Course Intructor:

Dr. Tariq Kamal

Content Page

Github Link: https://github.com/mahrukhyousuf/CA-Project

1. Introduction

The aim of our project was to construct a 5-stage pipelined RISC-V Processor capable of executing our chosen sorting algorithm : **Bubble-Sort Algorithm.**

The structure of our project is as follows:

- Converted the Bubble Sort Algorithm to RISC-V Assembly Language and verified it on Kwakil Venus.
- We constructed our Instruction Memory by converting the assembly code to machine code and modified our single-cycle processor we built in lab 11 to run our sorting algorithm.
- We then pipelined our single-cycle processor and tested it to make sure it was able to execute each instruction properly
- Implementing hazard detection mechanisms to identify various types of hazards such as data, control, and structural, and addressing these hazards with techniques like data forwarding, stalling, and pipeline flushing.
- Compared the performance between Single-Cycle Processor and our Pipelined Processor in terms of execution time

2. TASK 1

2.1 Bubble Sort Assembly Code to Machine Code

We converted our Bubble Sort Algorithm into RISCV Assembly Language on the Venus simulator environment. We used the same sorting algorithm we used in our previous lab 04. We implemented the sorting algorithm on an array initialized with length 3.

```
# Initializing Array - Size 3
addi x5, x0, 2 # -- 2
sw x5, 0x100 (x0)
addi x5, x0, 48 # -- 48
sw x5, 0x104(x0)
addi x5, x0, 24 # -- 24
sw x5, 0x108(x0)
```

```
addi x10, x10, 0x100
addi x11, x0, 3

bne x10, x0, Else
bne x11, x0, Else

Else: addi x18, x0, 0 # i

LOOP1: beq x18, x11, Exit1
    add x19, x0, x18      # j = i
    LOOP2:
        beq x19, x11, Exit2
        slli x5, x18, 2  # Calculating offset of i
        slli x6, x19, 2  # Calculating offset of j
        add x5, x5, x10
        add x6, x6, x10
        lw x28, 0(x5)    # Loading a[i]
        lw x29, 0(x6)    # Loading a[j]
        bge x28, x29, Loop2_Continued  # if a[i] >= a[j]
        # Swapping
        add x30, x0, x28   # temp = a[i]
        add x28, x0, x29   # a[i] = a[j]
        add x29, x0, x30   # a[j] = temp

        sw x28, 0(x5)
        sw x29, 0(x6)

        Loop2_Continued: addi x19, x10, 1   # Incrementing j
        beq x0, x0, LOOP2

    Exit2: addi x18, x18, 1   # i+=1
    beq x0,x0, LOOP1

```

Memory Values - Before & After Swapping

0x00000108	24	0	0	0
0x00000104	48	0	0	0
0x00000100	2	0	0	0

Figure 1.1 Array Size 3 - BEFORE SWAPPING

0x00000108	2	0	0	0
0x00000104	24	0	0	0
0x00000100	48	0	0	0

Figure 1.2 Array Size 3 - AFTER SWAPPING

2.2 Bubble Sort Implementation (Single-Cycle)

We modified our previous Lab 11 that integrated all the modules we had worked on throughout the semesters labs to make a processor. We modified Instruction Memory, Data Memory, Register File, Branch, ALU Control and ALU_64_Bit according to Task 1.

DATA MEMORY

Here we initialized of array of size 3 . You can see addition we made in the cod snippet below:

```
reg [7:0] memory [63:0];  // Memory array - 8 b
reg [63:0] temp_data;
integer i;
//since we are only concerned with the array va
//here to element and we've assumed size to be
        assign element1 = memory[20]; //24
        assign element2 = memory[12]; //48
        assign element3 = memory[4]; //2
```

ALU CONTROL

Here you can see the minor changes and in the Branch case, we used funct3 values from RISCV green card.

beq - SB - 1100011 000

bne - SB - 1100011 001

bge - SB - 1100011 101

```
module ALU_Control
(
    input [1:0] ALUOp,  // ALU operation code input
    input [3:0] Funct,  // Function code input
    output reg [3:0] Operation  // selected ALU operation output
);
        always @ (*)
            begin
            case(ALUOp)
                // When ALUOp is 2'b00
                2'b00: //slli case
                begin
                case({Funct[2:0]}) // SLLI operation - When ALUOp is 2'b00 (slli operation) & func code is 3'b001
                    3'b001:
                    begin
                    Operation= 4'b0111;  //Slli
                    end
                    default:
                    begin
                    Operation= 4'b0010; //Add - Otherwise
                    end
                 endcase
                 end
               2'b01: //branch case
                  begin
                   case ({Funct[2:0]})
                   3'b000:
                       begin
                       Operation= 4'b0110; //BEQ
                       end
                   3'b001:
                       begin
                       Operation = 4'b0110; //BNE
```

```
                       Operation = 4'b0110; //BNE
                       end
                   3'b101:
                       begin
                       Operation=4'b0110; //BGE
                       end
                    endcase
                   end

              2'b10: // check of and or add sub case
              begin
              case(Funct)
                  4'b0000:
                      begin
                      Operation = 4'b0010; //add
                      end
                  4'b1000:
                      begin
                      Operation = 4'b0110; //sub
                      end
                  4'b0111:
                      begin
                      Operation = 4'b0000; //and
                      end
                  4'b0110:
                      begin
                      Operation = 4'b0001; //or
                      end
              endcase
              end
        endcase
```

Branch Module

```
// timescale 1ns / 1ps
module Branch(
    input Branch,  // Signal indicating whether branch instruction is currently being executed
    input ZERO, // A signal indicating whether the result of a comparison operation is zero.
    input Isgreater,
    input [3:0] funct, // Function bits extracted from the instruction, which determine the branch condition.
    output reg switch_branch // This signal determines whether to execute the branch based on the condition evaluated.
    );

    always @(*) begin
        if(Branch) begin
        // Checks different branch conditions based on function bits
        // Zero=-true indicates previous operation resulted in zero
            case({funct[2:0]})
                3'b000: switch_branch = ZERO ? 1:0; // ZERO is true so switch_branch set to 1 (branch should be taken)
                3'b001: switch_branch = ZERO ? 0:1;
                3'b101: switch_branch =Isgreater ? 1:0;
            endcase
          end
        else
        // If Branch is not asserted, indicating no branch instruction is being executed, switch_branch is set to 0, indicating that no branch should be taken.
          switch_branch=0;
     end
endmodule
```

Control Unit

We used the table provided to us in our book for assigning values to control signals.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Code snippet provided below:

```
// CU takes 7-bit opcode input
module Control_Unit
(
    input [6:0] Opcode, // Intructions opcode
    output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, // Output Signals - control signals for various components of the processor
    output reg [1:0] ALUOp
);
// CASE STATEMENT - decodes opcode and sets control signal according to 7-bit opcode
    always @ (*) // Block of code should run whenever signal/variable in this block changes.
        begin
            case (Opcode)
            7'b0110011: //R type
             begin
                Branch = 0;
                MemRead = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc = 0;
                RegWrite = 1;
                ALUOp = 2'b10; // ALU Operation for R-Type Instructions
             end
            7'b0000011: //ld
             begin
                Branch = 0;
                MemRead = 1;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc = 1;
                RegWrite = 1;
                ALUOp = 2'b00; // ALU Operation for ld Instructions
             end
```

```
             end
            7'b0010011: //addi
               begin
                Branch = 1'b0;
                MemRead = 1'b0;
                MemtoReg = 1'b0;
                MemWrite = 1'b0;
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 2'b00; // ALU Operation for I-Type Instructions addi
             end
             7'b0100011: //S type
              begin
                Branch = 0;
                MemRead = 0;
                MemtoReg = 1'bX;
                MemWrite = 1;
                ALUSrc = 1;
                RegWrite = 0;
                ALUOp = 2'b00; // ALU Operation for S-Type Instructions
               end
            7'b1100011: //SB
             begin
            Branch = 1; // Branch Instruction
            MemRead = 0;
            MemtoReg = 1'bX;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
            ALUOp = 2'b01; // ALU Operation for R-Type Instructions
             end
        endcase
```

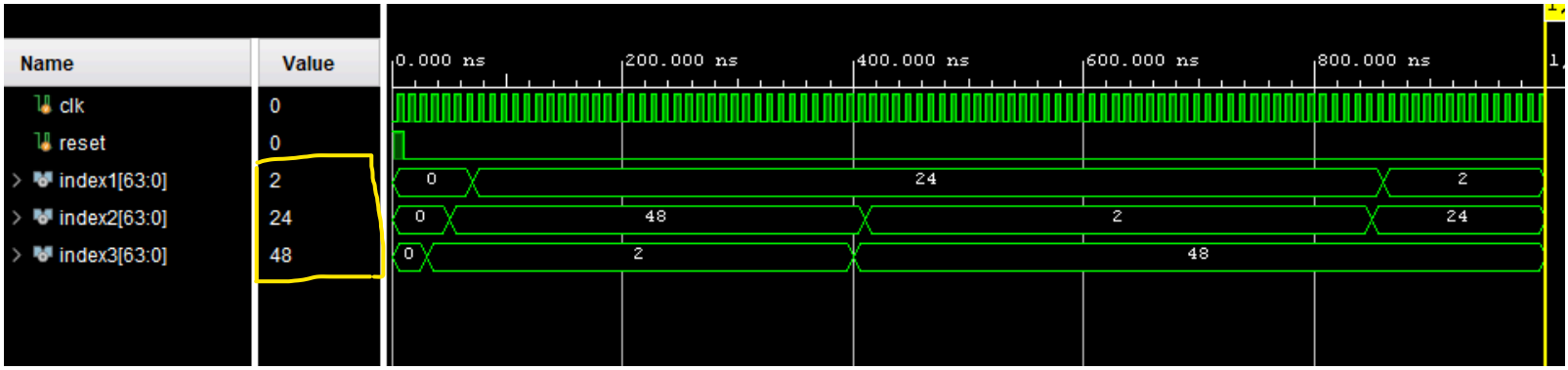
Instruction Memory

We converted each RISCV assembly instruction to its format type using the RISCV Green card. Then we grouped the 32-bit instructions to 8 bits.

Below is snippet of a few instructions we converted:

```
//  INSTRUCTION 1: addi x5, x0, 2
//opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000000010 (2)
    memory[0] = 8'b10010011;
    memory[1] = 8'b00000010;
    memory[2] = 8'b00100000;
    memory[3] = 8'b00000000;

//  INSTRUCTION 2: sd x5, 0x100 (x0)
//opcode: 0100011, rs2: 00101 (5), rs1: 00000 (0), immediate: 0001 0000 0000 (0x100), funct3: 011
    memory[4] = 8'b00100011;
    memory[5] = 8'b00110010;
    memory[6] = 8'b01010000;
    memory[7] = 8'b00000000;

//  INSTRUCTION 3 :addi x5, x0, 48
// opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000110000 (48)
    memory[8] = 8'b10010011;
    memory[9] = 8'b00000010;
    memory[10] = 8'b00000000;
    memory[11] = 8'b00000011;

//  INSTRUCTION 4: sd x5, 0x104(x0)
// opcode: 0100011, rs2: 00101 (5), rs1: 00000 (0), immediate: 0001 0000 0100 (0x104), funct3: 011
    memory[12] = 8'b00100011;
    memory[13] = 8'b00110110;
    memory[14] = 8'b01010000;
    memory[15] = 8'b00000000;
```

2.2 Result - Wave Simulation



After running simulation, our code ran successfully as you can see our values were sorted.

Hence completing task 1.

3. TASK 2

3.1 Pipelined RISCV Processor Implementation

Once our RISCV Single-Cycle Processor was functional, we modified our processor to a pipelined processor by introducing the Pipeline registers as modules:

1. **IF/ID**
2. **ID/EX**
3. **EX/MEM**
4. **MEM/WB**

These Pipeline Registers ensure each stage of the pipeline operates in the same frequency and prevents interference between consecutive stages and help in detecting and mitigation of hazards which we will address in Task 03

1. IF/ID (Instruction Fetch/Decode) Module:

Serves as Pipeline Register to store : the Instruction and Program Counter (PC) fetched in Instruction Fetch (IF) stage.

```
module IF_ID
(
    input clk,
    input [63:0] pc_wire,
    input [31:0] inst,
    output reg [63:0] pc_store,
    output reg [31:0] inst_store
);
    // Store the instruction and program counter (PC) fetched in the Instruction Fetch (IF) stage.
    always @ (negedge clk) begin
        pc_store = pc_wire;
        inst_store = inst;
        end

endmodule
```

2. ID/EX (Instruction Decode/Execute) Module:

Serves as a register file to store the input signals used by the subsequent Pipeline Stage

The register versions of all input signals are the stored versions of these inputs - stored in the corresponding register.

The stored values are available in the next pipeline stage: EX/MEM Stage

```
module ID_EX(
    input clk,
    input [63:0] pc_wire,  // PC value - address of next instruction
    input [63:0] readdata1,  // Data Read from Register File - first source register (rs1)
    input [63:0] readdata2, // Data Read from Register File - sencond source register (rs2)
    input [63:0] immgen_val, //  imm-data - Immediate value generated by the immediate generati
    input [3:0] funct_in, // Function code extracted from the instruction
    input [4:0] rd_in, // Destination Register Address extracted from instruction
    // Control Signals
    input MemtoReg,
    input RegWrite,
    input Branch,
    input MemWrite,
    input MemRead,
    input ALUsrc,
    input [1:0] ALU_op,

    // All the stored versions of the input to the pipeline registers
    output reg [63:0] pc_wire_store,
    output reg [63:0] readdata1_store,
    output reg [63:0] readdata2_store,
    output reg [63:0] immgen_val_store,
    output reg [3:0] funct_in_store,
    output reg [4:0] rd_in_store,
    output reg MemtoReg_store,
    output reg RegWrite_store,
    output reg Branch_store,
    output reg MemWrite_store,
    output reg MemRead_store,
    output reg ALUsrc_store,
    output reg [1:0] ALU_op_store
    );
```

```
// Assigning input to output - storing values
always @(negedge clk)
begin
pc_wire_store = pc_wire;
readdata1_store = readdata1;
readdata2_store = readdata2;
immgen_val_store = immgen_val;
funct_in_store = funct_in;
rd_in_store = rd_in;
RegWrite_store = RegWrite;
MemtoReg_store = MemtoReg;
Branch_store = Branch;
MemWrite_store = MemWrite;
MemRead_store = MemRead;
ALUsrc_store = ALUsrc;
ALU_op_store = ALU_op;
end
```

3. EX/MEM (Execute/Memory) Module:

This module implements a pipeline register to hold the results and control signals generated in the Execute stage, ensuring they remain stable and available throughout the execution of the current cycle.

```
module EX_MEM
(
    input clk,
    input RegWrite, MemtoReg,
    input Branch, Zero, MemWrite, MemRead, Is_Greater,
    input [63:0] sum, ALU_result, Readdata2,
    input [3:0] funct_in,
    input [4:0] rd,

    output reg RegWrite_store, MemtoReg_store,
    output reg Branch_store, Zero_store, MemWrite_store, MemRead_store, Is_Greater_store,
    output reg [63:0] sum_store, ALU_result_store, WriteData,
    output reg [3:0] funct_in_store,
    output reg [4:0] rd_store
);

always @(negedge clk) begin
    RegWrite_store = RegWrite;
    MemtoReg_store = MemtoReg;
    Branch_store = Branch;
    Zero_store = Zero;
    Is_Greater_store = Is_Greater;
    MemWrite_store = MemWrite;
    MemRead_store = MemRead;
    sum_store = sum;
    ALU_result_store = ALU_result;
    WriteData = Readdata2;
    funct_in_store = funct_in;
    rd_store = rd;
end

endmodule
```

4. MEM/WB (Memory/ Write Back Stage):

By storing the results and control signals, the module facilitates the subsequent pipeline stage (Write-Back stage) to access and process them without interference, ensuring smooth data flow through the pipeline.

```
module MEM_WB
(
    input clk,
    input RegWrite, MemtoReg,
    input [63:0] ReadData, ALU_result,
    input [4:0] rd,

    output reg RegWrite_store, MemtoReg_store,
    output reg [63:0] ReadData_store, ALU_result_store,
    output reg [4:0] rd_store

);

always @(negedge clk) begin

    RegWrite_store = RegWrite;
    MemtoReg_store = MemtoReg;
    ReadData_store = ReadData;
    ALU_result_store = ALU_result;
    rd_store = rd;
end
```

3.2 Test Case

To test whether our Pipelined Implementation is working, we will test this on our first instruction defined in our Instruction Memory

Test Case1:

```
//  INSTRUCTION 1: addi x5, x0, 2
//opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000000010 (2)
    memory[0] = 8'b10010011;
    memory[1] = 8'b00000010;
    memory[2] = 8'b00100000;
    memory[3] = 8'b00000000;
```

According to this logic, **rd_out** should be 5 and **rs1** is x0 and **rs2** is 2 according to the instruction (addi x5, x0, 2). These are the values that should be input to Pipeline Register ID/EX, to confirm its functionality - the same values should be outputted.

Test Case 2:

```
////  sd x5, 0x100 (x0)
////opcode: 0100011, rs2: 00101 (5), rs1: 00000 (0), immediate: 0001 0000 0000 (0x100), funct3: 011
    memory[4] = 8'b00100011;
    memory[5] = 8'b00110010;
    memory[6] = 8'b01010000;
    memory[7] = 8'b00000000;
```

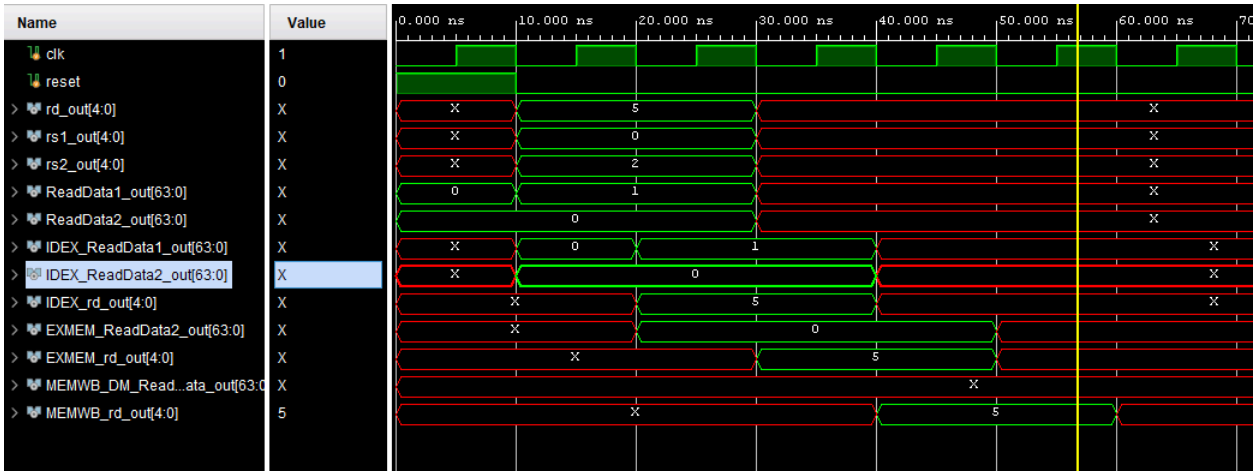
Test Case 3:

```
////  addi x5, x0, 48
//// opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000110000 (48)
    memory[8] = 8'b10010011;
    memory[9] = 8'b00000010;
    memory[10] = 8'b00000000;
    memory[11] = 8'b00000011;
```

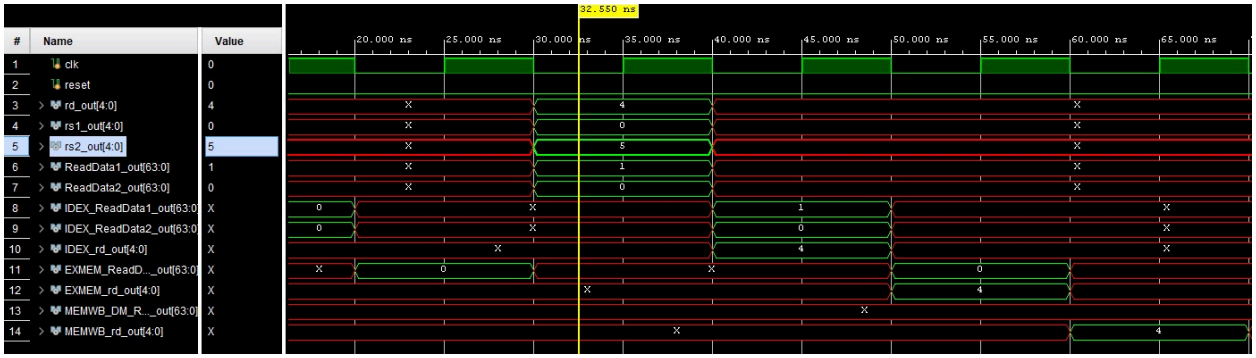
We tested this logic which can be seen in our wave simulation.

3.3 Result - Wave Simulation

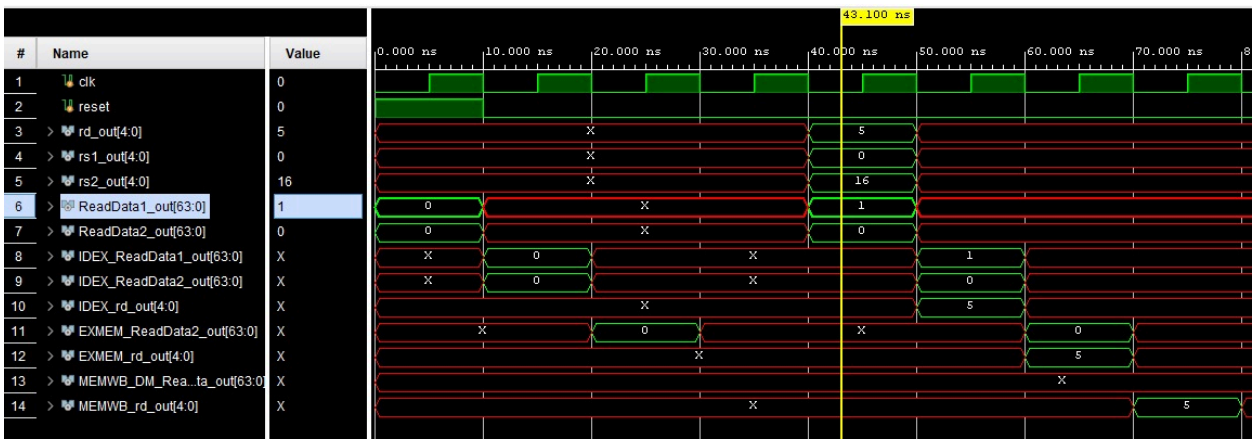
Test Case 1:



Test Case 2:



Test Case 3:



Above is the simulation output on all instructions and we can see the values have successfully been passed from one pipeline register to the next.

Hence our task 2 has been completed.

4. TASK 3

4.1 Implementing Hazard Detection Circuitry

To implement hazard detection circuitry in a pipelined processor, you typically look for hazards that might occur due to data dependencies between instructions in different stages of the pipeline. The most common hazards are data hazards, which occur when an instruction depends on the result of a previous instruction that has not yet completed.

We address this by incorporating a **hazard detection mechanism.**

To handle potential issues such as data hazards, structural hazard and control hazards, we implemnted a **Hazard Detection** module that determines when to stall the pipeline and **Forwarding Unit** module to either stall or clear the pipeline.

Hazard Detection Module

We incorporated an additional module Hazard_Detection to our pipelined processor that stalls the processors if an instruction is dependent on the outcome of another instruction, preventing updates to the PC and Instruction Register. If we do not need to stall, the processor proceeds per usual.

A hazard detection unit operates during the ID stage so that it can insert the stall between the load and the instruction dependent on it

Hazarding Condition:

- We stall the pipeline when

```
If (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
```

In our Hazard Detection module, the following three lines perform stalling.

```
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
        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
```

Stalling here

Forwarding Unit Module

According to:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

1. **EX Hazard:**

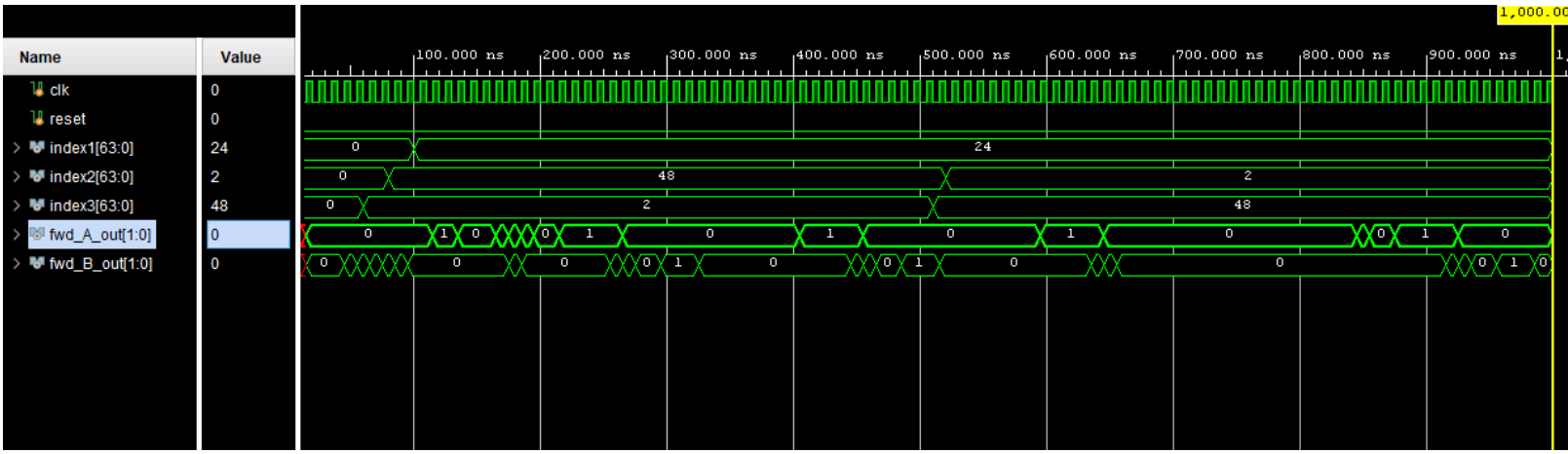
```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd !=0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd !=0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
ForwardB = 10
```

2. **MEM Hazard**

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd !=0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd !=0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
ForwardB = 01
```

```
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

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

4.2 Result - Wave Simulation



5. Performance Comparison

The pipelined RISC-V processor requires more than 1000 nanoseconds to finish executing the bubble sort algorithm, in contrast to the single-cycle processor, which completes the same task in 1000 nanoseconds. Consequently, the pipelined variant demonstrates reduced performance compared to the single-cycle model. This lower efficiency in the pipelined processor is attributed to unavoidable pipeline stalls that occur even when hazard detection and data forwarding mechanisms are in place. On the other hand, the absence of such stalls in the single-cycle processor accounts for its faster sorting operation.

6. Conclusion & Challenges

We faced alot of minor issues due to which we would not be able to proceed, we had recheck and correct our modules several times to be functional for this project. Implementing the branch equals instructions proved to be a complex task. However, it is these challenges that further strengthened our concepts and understanding of the processor and its functionality.

7. References

Course Book. Computer Organization and Design: The Hardware/Software Interface RISC-V Edition by David A. Patterson, John L. Hennessy