RISC V Pipelined Processor

Final Report

Submitted by:
Breeha Qasim
Ashbah Faisal
Rameez Wasif

Research Assistant:
Maham Tabassum

Course Instructor:
Dr. Tariq Kamal

Computer Science
Habib University Spring
Semester ’24

Date: April 22, 2024

Contents

1 Introduction

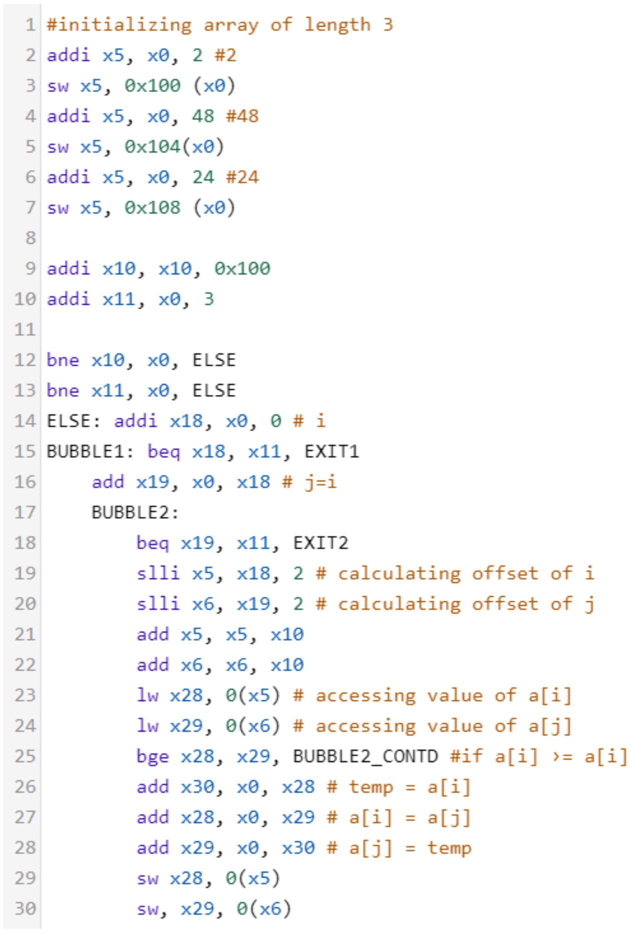
Our project involved developing a 5-stage pipelined RISC-V processor designed to execute a bubble sort algorithm. The project consisted of several key activities:

1. Translating bubble sort algorithm pseudocode into RISC-V assembly language and confirming its correctness using the Venus simulator.
2. Adapting a single-cycle processor from our previous lab (Lab 11) to execute the bubble sort algorithm which we wrote in Lab 4 Task 3 on Venus simulator.
3. Implementing pipelining in the processor and conducting a series of test cases to validate the functionality of the pipelined variant.
4. 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.
5. Evaluating and contrasting the execution time required for sorting an array using both the Single Cycle Processor and the newly developed pipelined RISC-V Processor.

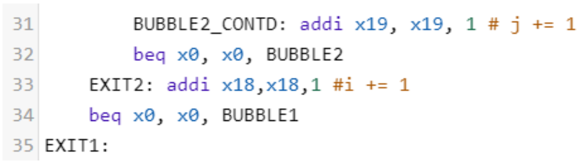
2 Task 1

2.1 Bubble Sort Assembly Code to Machine Code

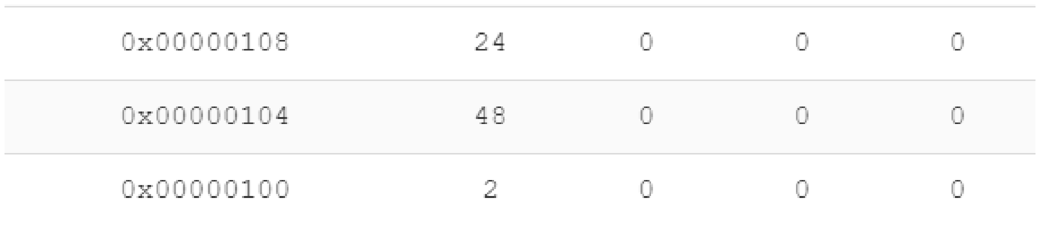
Initially, we executed the sorting algorithm using RISC V assembly language within the Venus simulator environment. This was not our first experience with implementing sorting, as we had previously implemented sorting logic in Lab 4 Task 03.



```
#initializing array of length 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
BUBBLE1: beq x18, x11, EXIT1
    add x19, x0, x18 # j=i
    BUBBLE2:
        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) # accessing value of a[i]
        lw x29, 0(x6) # accessing value of a[j]
        bge x28, x29, BUBBLE2_CONTD #if a[i] >= a[i]
        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)
```

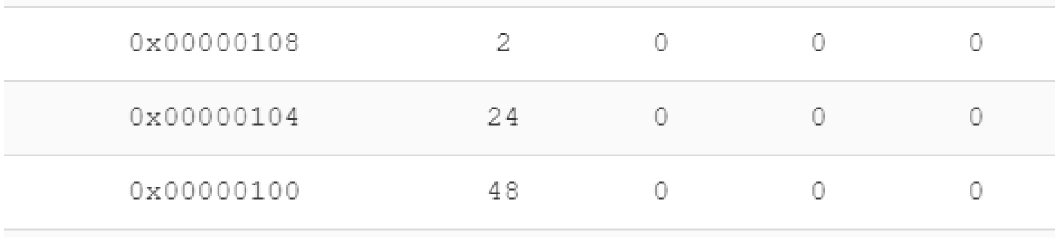


```
        BUBBLE2_CONTD: addi x19, x19, 1 # j += 1
        beq x0, x0, BUBBLE2
    EXIT2: addi x18,x18,1 #i += 1
    beq x0, x0, BUBBLE1
EXIT1:
```



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

Figure 1.1: Array before Sorting



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

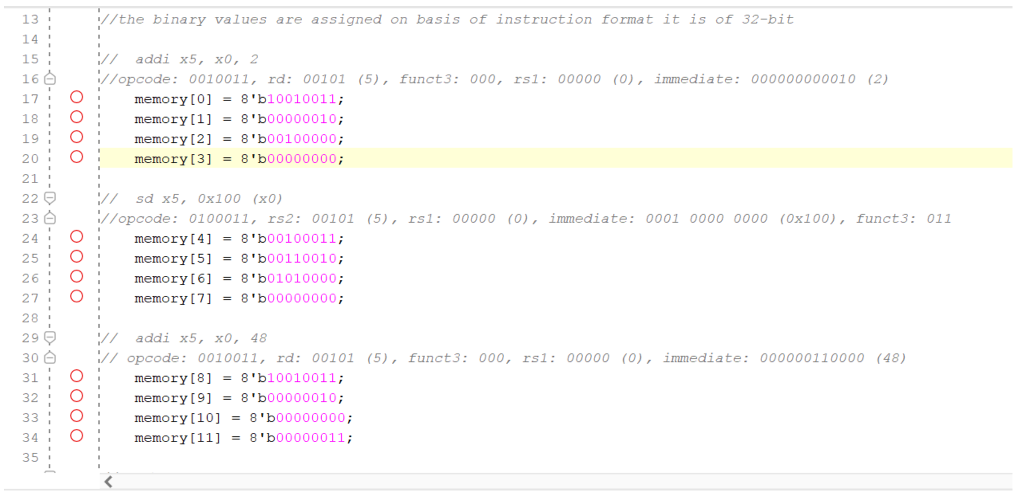
Figure 1.2: Array after Sorting

2.2 Bubble Sort Implementation Single Cycle

We made minor modifications to the Lab 11 module where we instantiated all modules together to make the processor. We modified Instruction Memory, Data Memory, Register File, Branch, ALU Control and ALU_64_Bit. Major Changes were seen in the Instruction Memory module. The rest modules have minor changes.

2.2.1 Instruction Memory

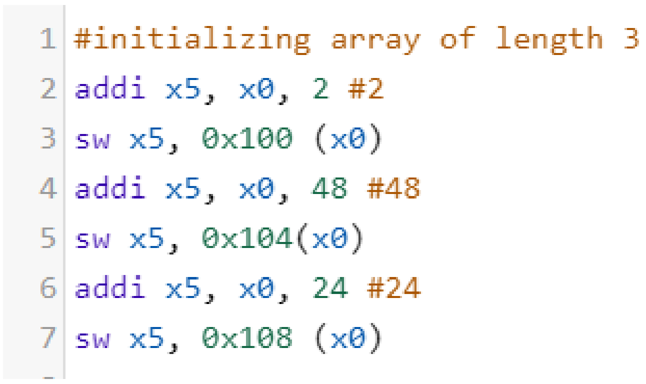
Here as you can see we converted each assembly instruction to its Format Type with the help of RISC V Green Card. Then we grouped the 32-bit instruction to 8 bits.



```
//the binary values are assigned on basis of instruction format it is of 32-bit

//  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;

//  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;

//  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;
```

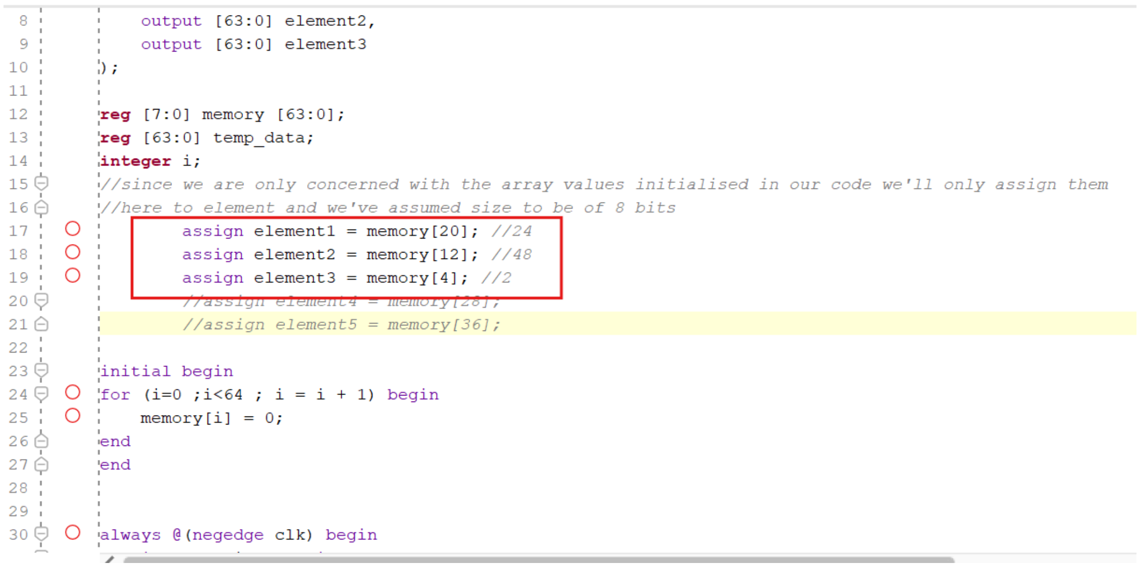
2.2.2 Data Memory

Here we've initialized the values in an inverted way in our Verilog Code.



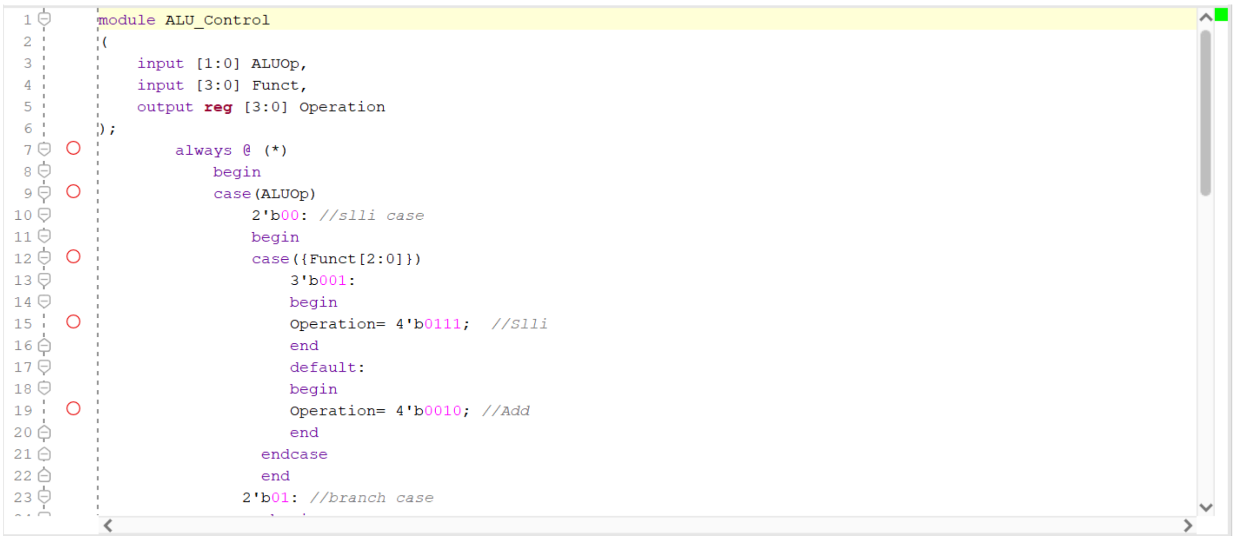
```
#initializing array of length 3
addi x5, x0, 2 #2
sw x5, 0x100 (x0)
addi x5, x0, 48 #48
sw x5, 0x104(x0)
addi x5, x0, 24 #24
sw x5, 0x108 (x0)
```

Figure 2.2.2: Initialisation of Array

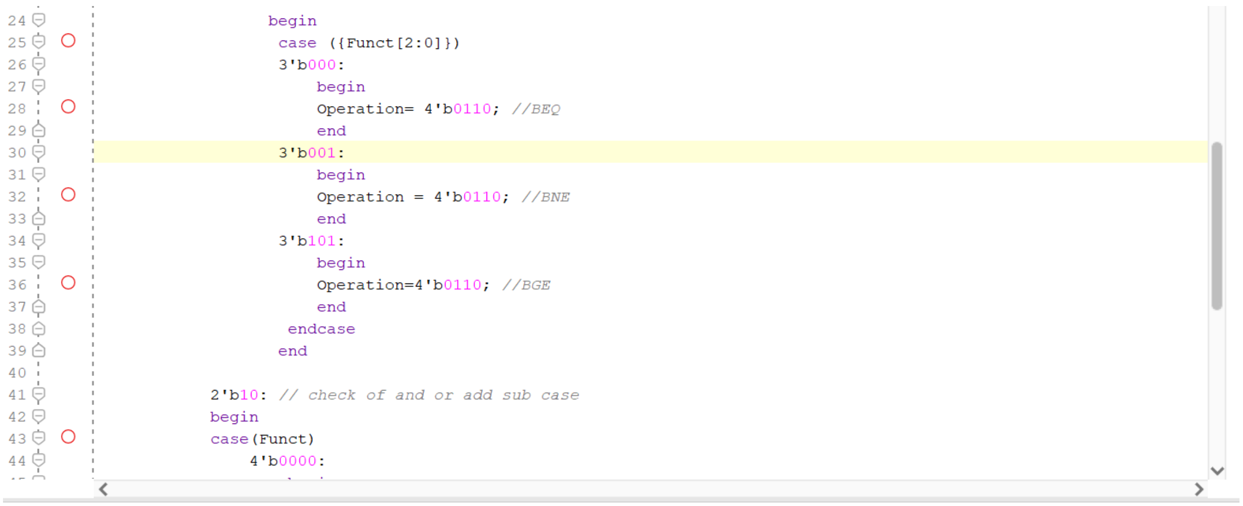


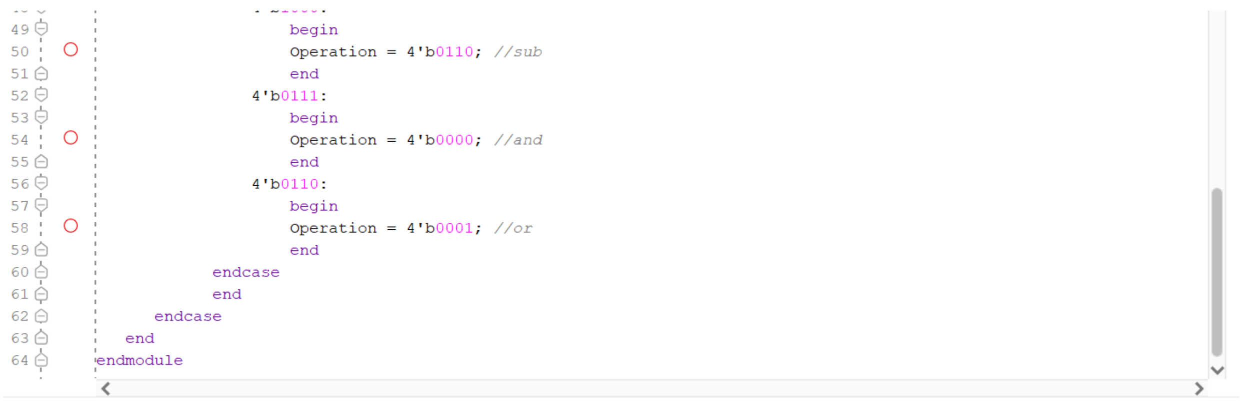
```
    output [63:0] element2,
    output [63:0] element3
);

reg [7:0] memory [63:0];
reg [63:0] temp_data;
integer i;
//since we are only concerned with the array values initialised in our code we'll only assign them
//here to element and we've assumed size to be of 8 bits
        assign element1 = memory[20]; //24
        assign element2 = memory[12]; //48
        assign element3 = memory[4]; //2
        //assign element4 = memory[28];
        //assign element5 = memory[36];

initial begin
for (i=0 ;i<64 ; i = i + 1) begin
    memory[i] = 0;
end
end


always @(negedge clk) begin
```

2.2.3 ALU Control

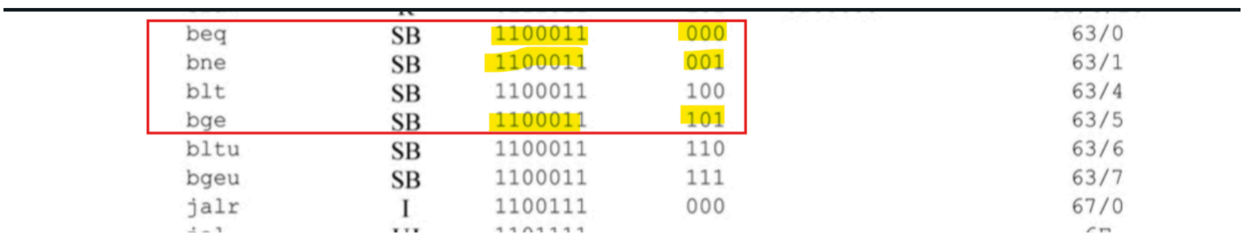


```
module ALU_Control
(
    input [1:0] ALUOp,
    input [3:0] Funct,
    output reg [3:0] Operation
);
        always @ (*)
            begin
            case(ALUOp)
                2'b00: //slli case
                begin
                case({Funct[2:0]})
                    3'b001:
                    begin
                    Operation= 4'b0111;  //Slli
                    end
                    default:
                    begin
                    Operation= 4'b0010; //Add
                    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
                       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:
```

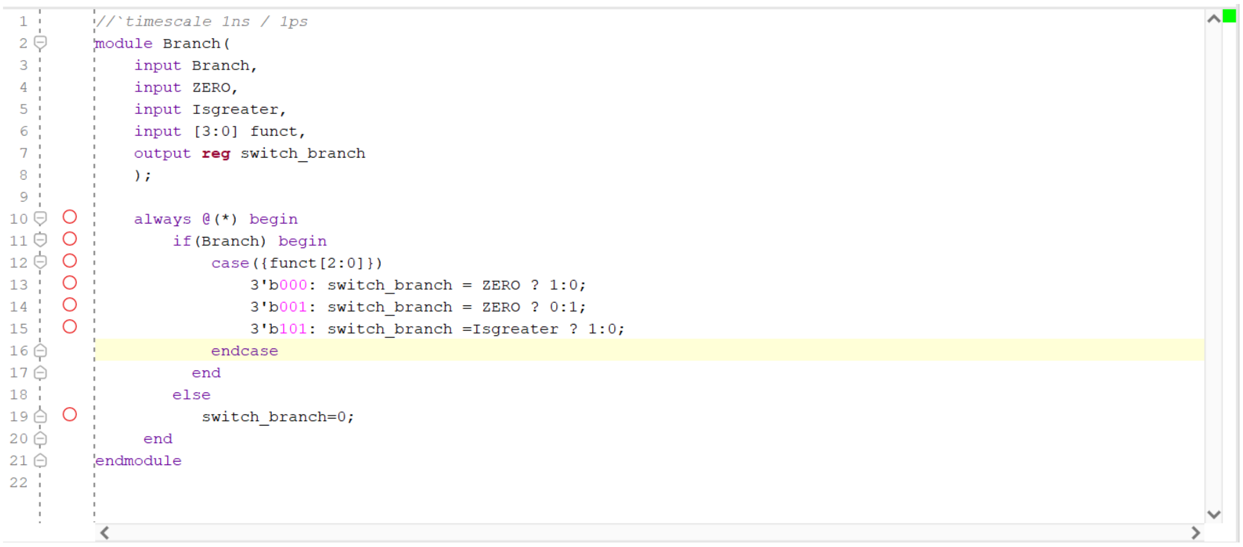

```
                    begin
                    Operation = 4'b0110; //sub
                    end
                4'b0111:
                    begin
                    Operation = 4'b0000; //and
                    end
                4'b0110:
                    begin
                    Operation = 4'b0001; //or
                    end
            endcase
            end
      endcase
   end
endmodule
```

In Branch case, we used funct3 values from RISCV Green Card

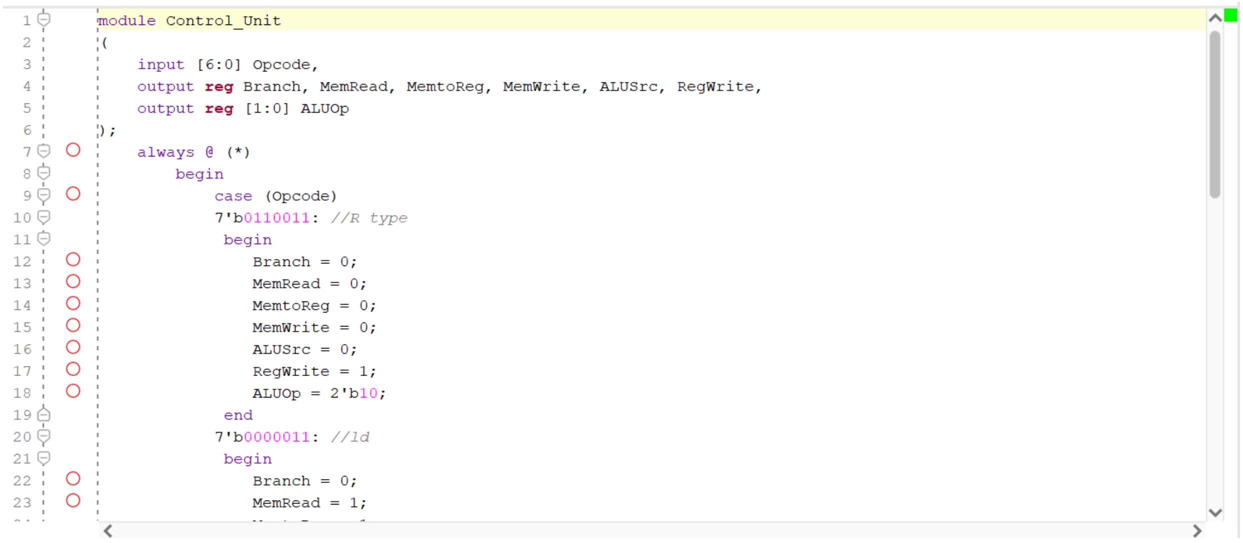


beq	SB	1100011	000	63/0
bne	SB	1100011	001	63/1
blt	SB	1100011	100	63/4
bge	SB	1100011	101	63/5
bltu	SB	1100011	110	63/6
bgeu	SB	1100011	111	63/7
jalr	I	1100111	000	67/0

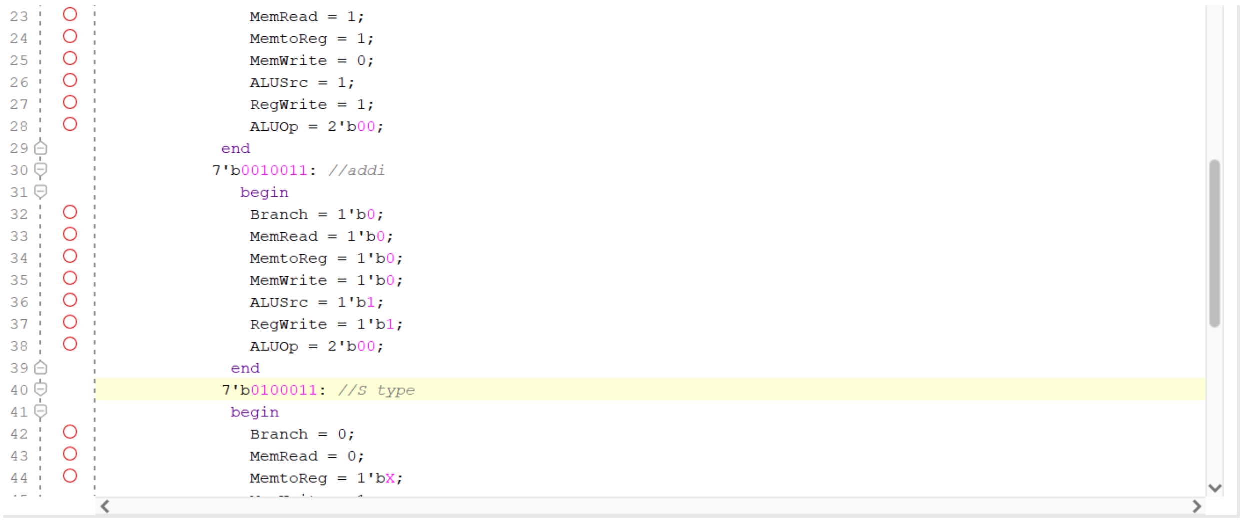
2.2.4 Branch Module


```
//`timescale 1ns / 1ps
module Branch(
    input Branch,
    input ZERO,
    input Isgreater,
    input [3:0] funct,
    output reg switch_branch
    );

    always @(*) begin
        if(Branch) begin
            case({funct[2:0]})
                3'b000: switch_branch = ZERO ? 1:0;
                3'b001: switch_branch = ZERO ? 0:1;
                3'b101: switch_branch =Isgreater ? 1:0;
            endcase
          end
        else
           switch_branch=0;
     end
endmodule
```

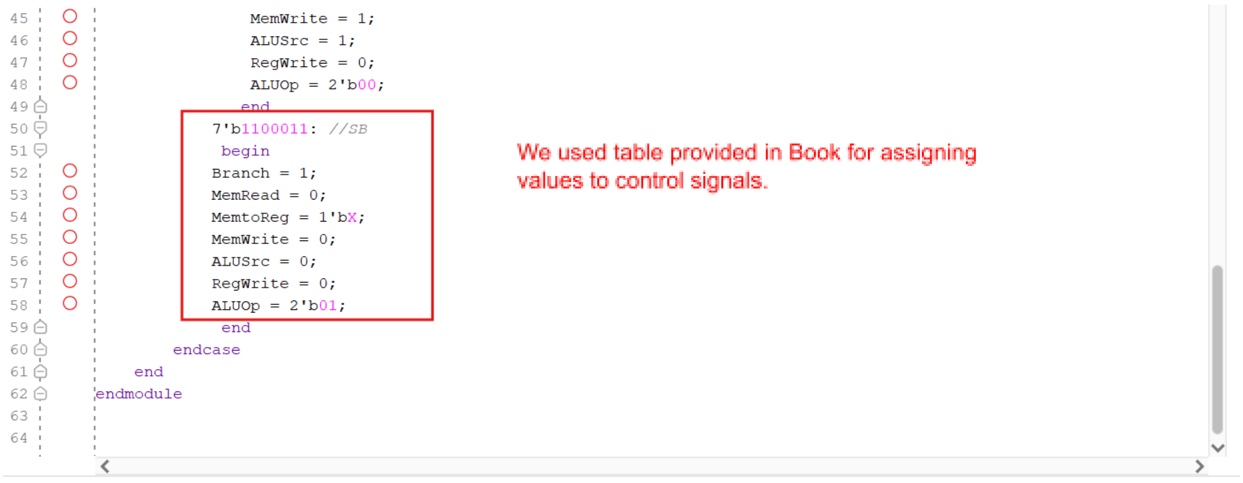
2.2.5 Control Unit



```
module Control_Unit
(
    input [6:0] Opcode,
    output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite,
    output reg [1:0] ALUOp
);
    always @ (*)
        begin
            case (Opcode)
            7'b0110011: //R type
             begin
                Branch = 0;
                MemRead = 0;
                MemtoReg = 0;
                MemWrite = 0;
                ALUSrc = 0;
                RegWrite = 1;
                ALUOp = 2'b10;
             end
            7'b0000011: //ld
             begin
                Branch = 0;
                MemRead = 1;
```

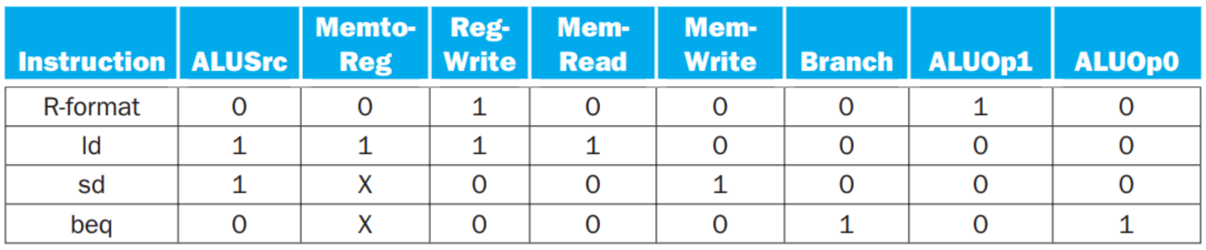


```
                MemRead = 1;
                MemtoReg = 1;
                MemWrite = 0;
                ALUSrc = 1;
                RegWrite = 1;
                ALUOp = 2'b00;
             end
            7'b0010011: //addi
               begin
                Branch = 1'b0;
                MemRead = 1'b0;
                MemtoReg = 1'b0;
                MemWrite = 1'b0;
                ALUSrc = 1'b1;
                RegWrite = 1'b1;
                ALUOp = 2'b00;
              end
             7'b0100011: //S type
              begin
                Branch = 0;
                MemRead = 0;
                MemtoReg = 1'bX;
```



```
                MemWrite = 1;
                ALUSrc = 1;
                RegWrite = 0;
                ALUOp = 2'b00;
               end
            7'b1100011: //SB
             begin
            Branch = 1;
            MemRead = 0;
            MemtoReg = 1'bX;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
            ALUOp = 2'b01;
             end
        endcase
    end
endmodule
```

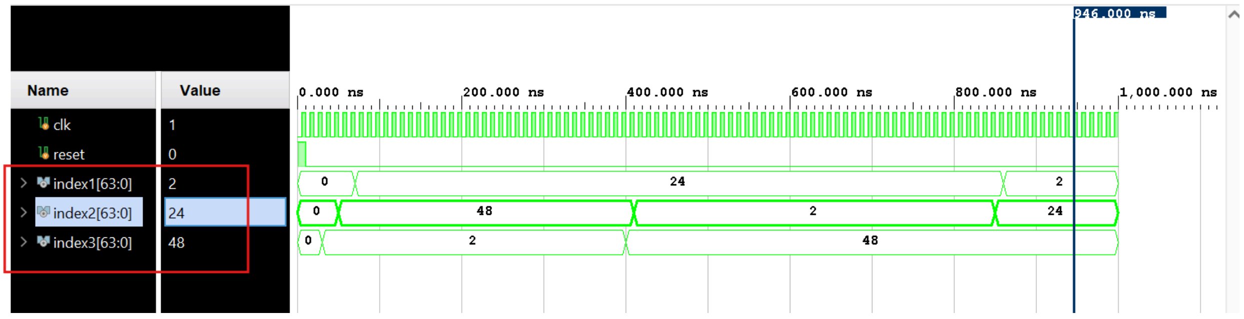
We used table provided in Book for assigning values to control signals.



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

Figure 2.2.5: The setting of the control lines

2.3 Simulation Output



Name	Value
clk	1
reset	0
index1[63:0]	2
index2[63:0]	24
index3[63:0]	48

Figure 2.3: Sorted Array

3 Task 2

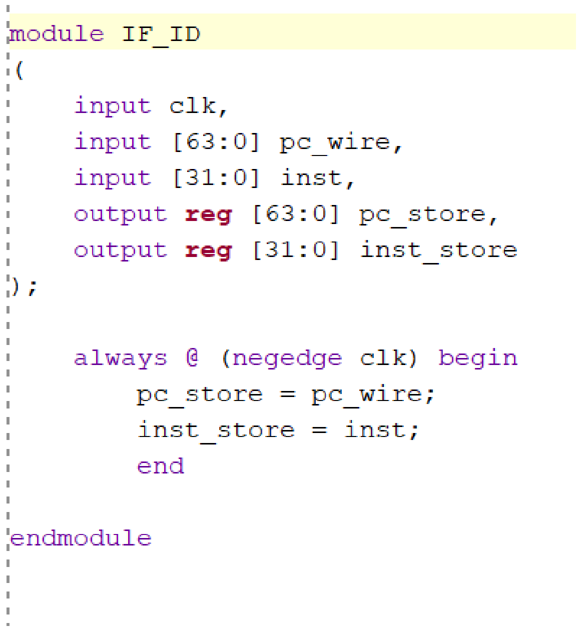
3.1 Pipelined RISC V Processor

We first got the algorithm to work on a single-cycle processor. Then we updated the processor to a pipelined version. We added pipeline registers named

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

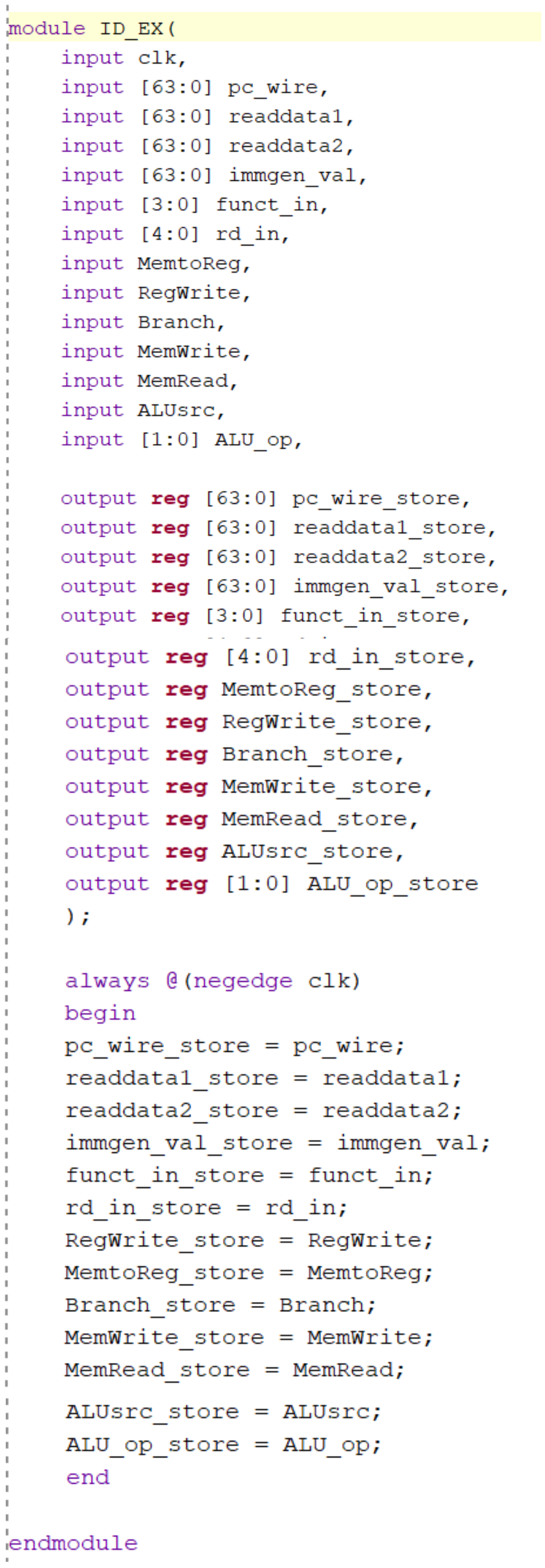
based on what we learned from our textbook. These registers hold data from one stage of the pipeline to the next. We checked that each stage of the pipeline was working right by testing instructions one by one.

3.3.1 IF/ID Module



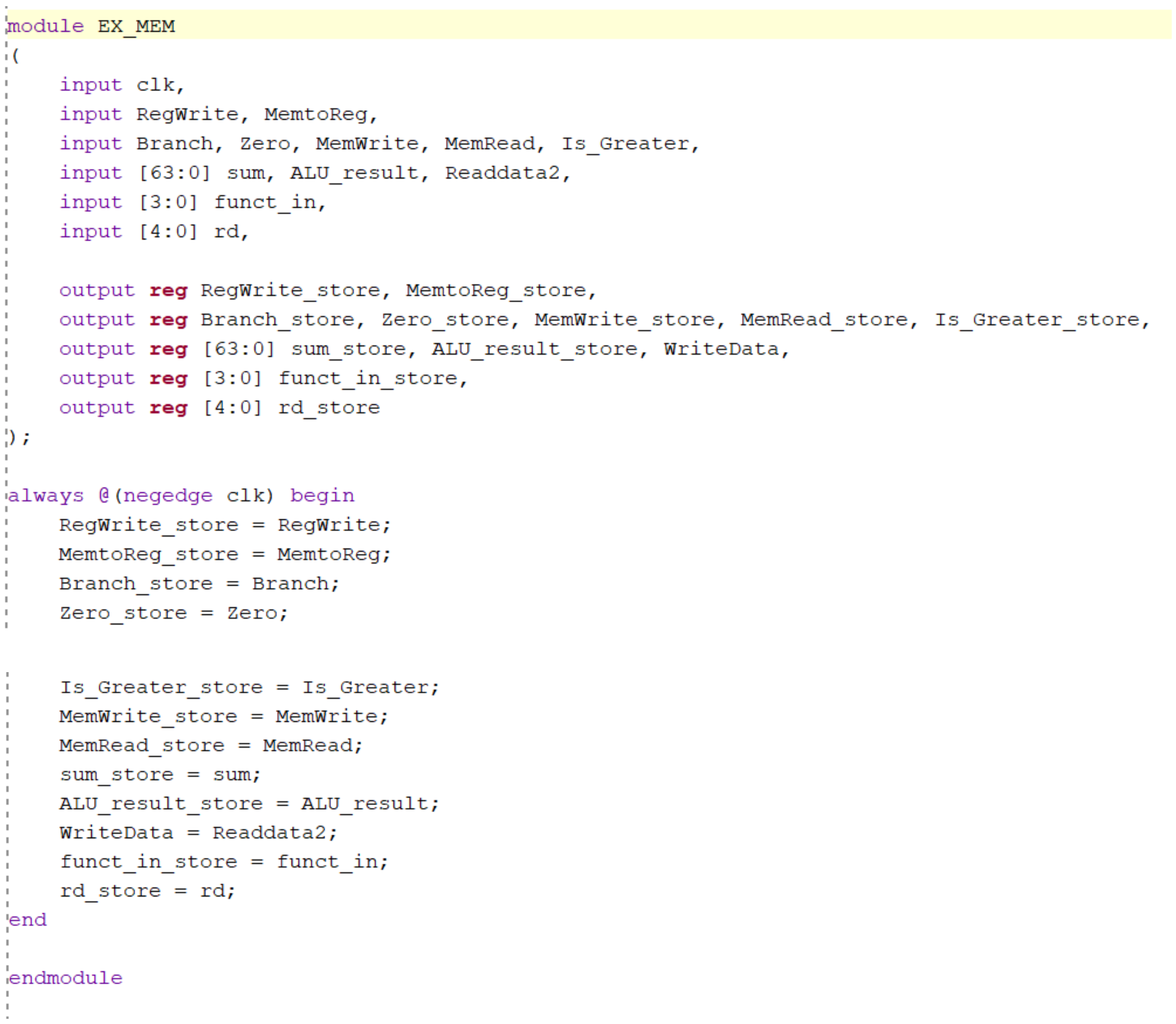
```
module IF_ID
(
    input clk,
    input [63:0] pc_wire,
    input [31:0] inst,
    output reg [63:0] pc_store,
    output reg [31:0] inst_store
);

    always @ (negedge clk) begin
        pc_store = pc_wire;
        inst_store = inst;
        end

endmodule
```

3.3.2 ID/EX Module



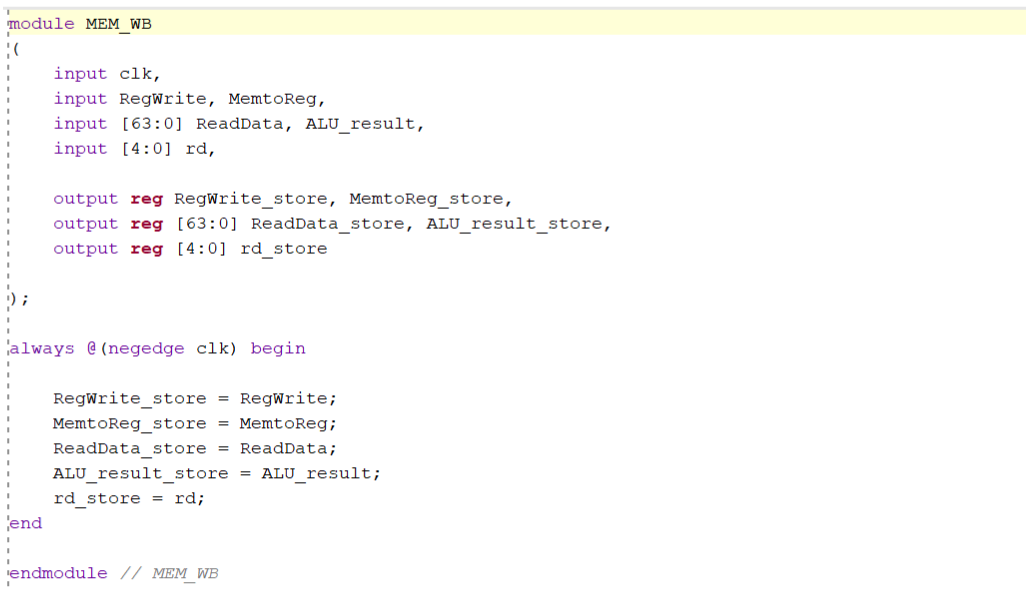
```
module ID_EX(
    input clk,
    input [63:0] pc_wire,
    input [63:0] readdata1,
    input [63:0] readdata2,
    input [63:0] immgen_val,
    input [3:0] funct_in,
    input [4:0] rd_in,
    input MemtoReg,
    input RegWrite,
    input Branch,
    input MemWrite,
    input MemRead,
    input ALUsrc,
    input [1:0] ALU_op,

    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
    );

    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

endmodule
```

3.3.3 EX/MEM Module



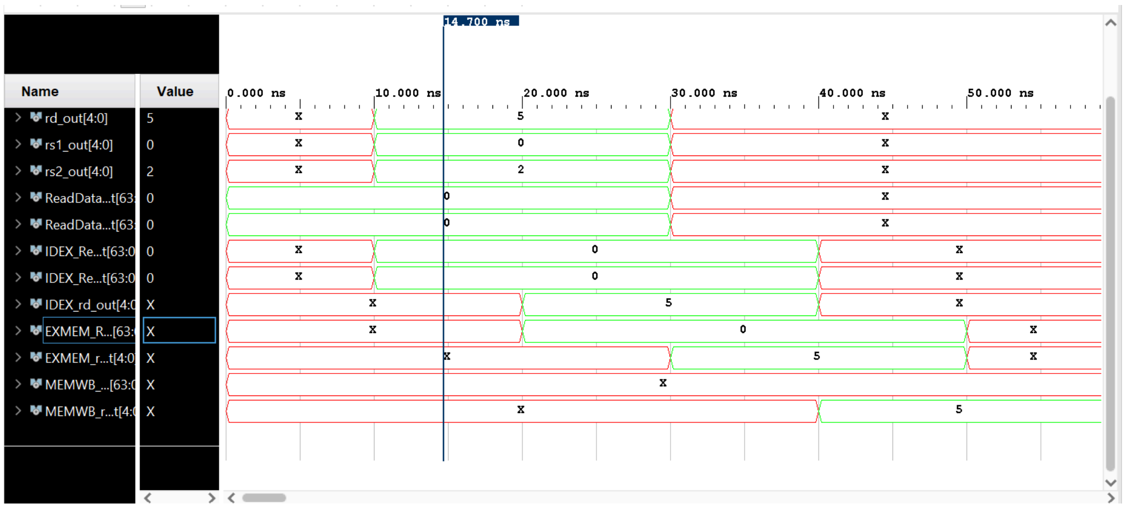
```
module EX_MEM
(
    input clk,
    input RegWrite, MemtoReg,
    input Branch, Zero, MemWrite, MemRead, Is_Greater,
    input [63:0] sum, ALU_result, Readdata2,
    input [3:0] funct_in,
    input [4:0] rd,

    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
```

3.3.4 MEM/WB Module



```
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

endmodule // MEM_WB
```

3.2 Test Case for addi x5, x0, 2



```
Name                 Value
rd_out[4:0]          5
rs1_out[4:0]         0
rs2_out[4:0]         2
ReadData...t[63:     0
ReadData...t[63:     0
IDEX_Re...t[63:0     0
IDEX_Re...t[63:0     0
IDEX_rd_out[4:0      X
EXMEM_R...[63:       X
EXMEM_r...t[4:0      X
MEMWB_...[63:0       X
MEMWB_r...t[4:0      X
```

Figure 3.2.1: Snippet of Simulation Output for Test Case

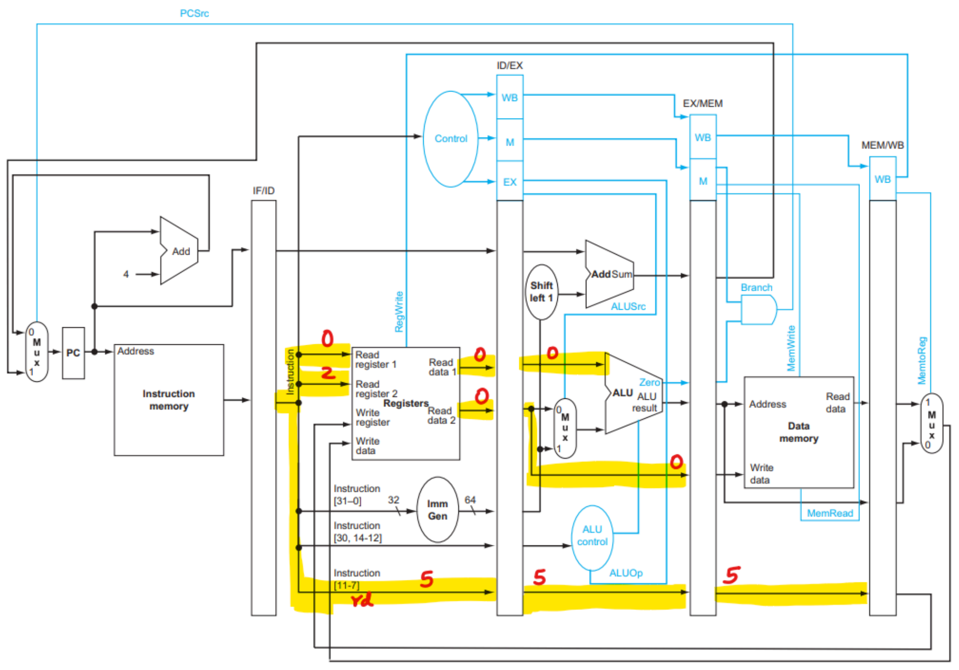


Figure 3.2.2: Tracing of instruction on Pipelined Data Path

3.3 Simulation Output

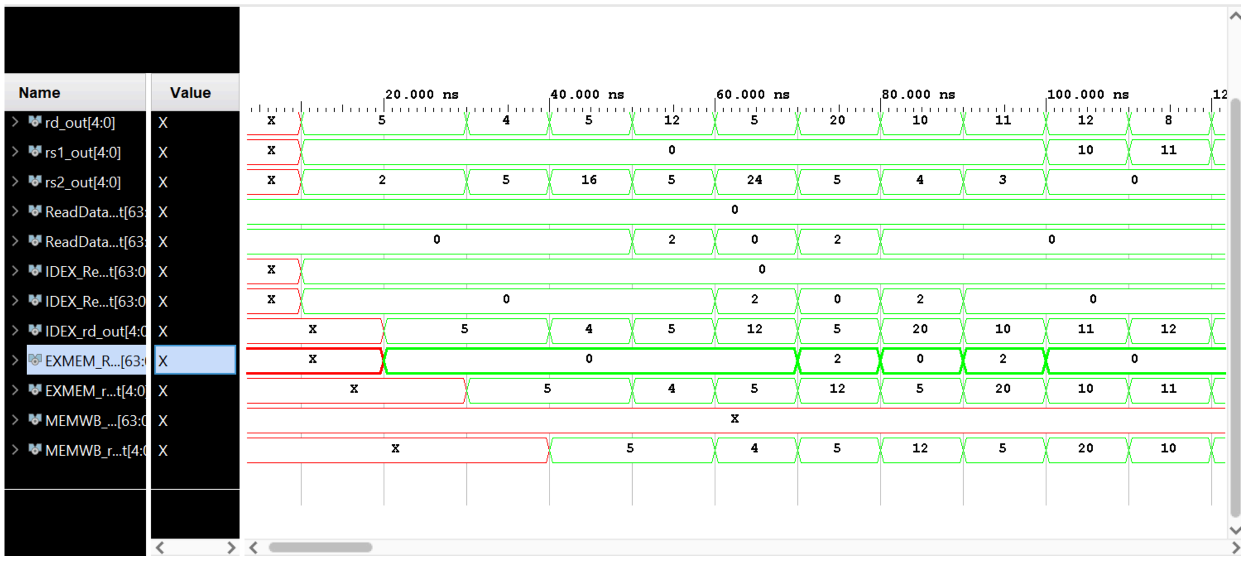


Figure 3.3: Snippet of Simulation Output for all Instructions

4 Task 3

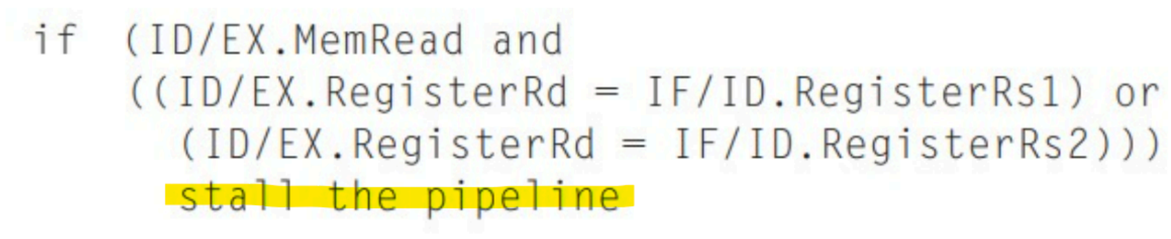
4.1 Implementing Hazard Detection Circuitry

The code addresses potential issues like data, structural, and control hazards by incorporating hazard detection mechanisms that stall the pipeline. These complications generally stem from code dependencies or situations where data must be forwarded to subsequent stages. To handle these scenarios, we have made a hazard detection component that determines when to pause the pipeline's operations or instruct the forwarding unit to either stall or clear the pipeline.

4.1.1 Hazard Detection Module

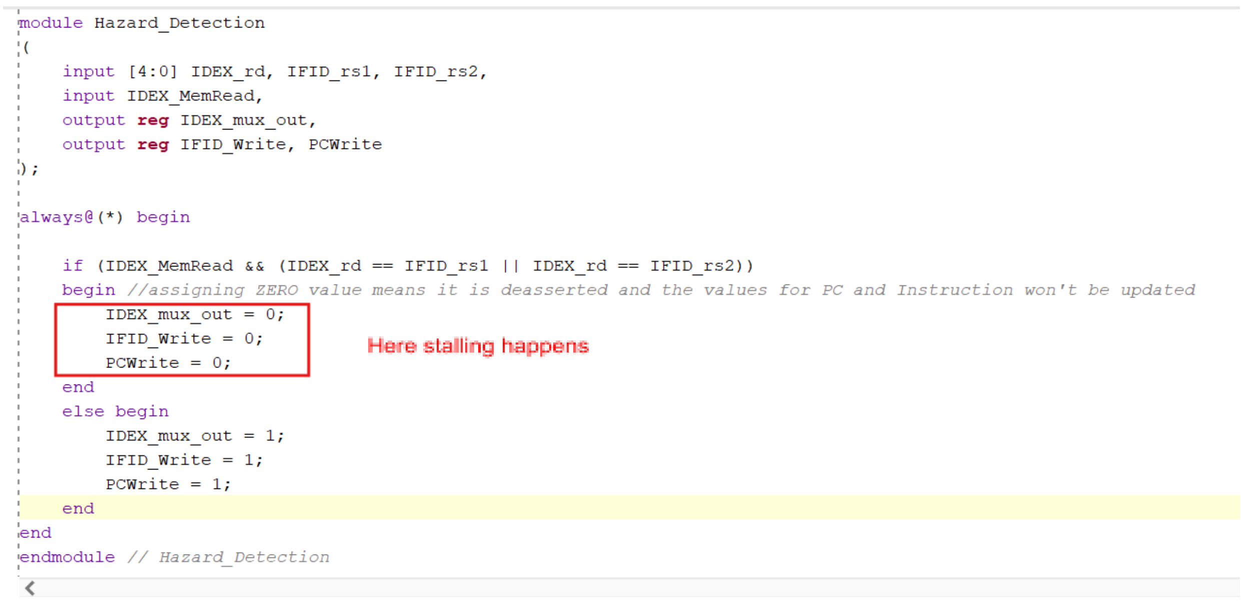
The **`Hazard_Detection`** module in Verilog pauses the processor's actions if the current instruction is waiting for data, preventing updates to the program counter and the instruction register. If no such waiting is needed, the processor proceeds normally.

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



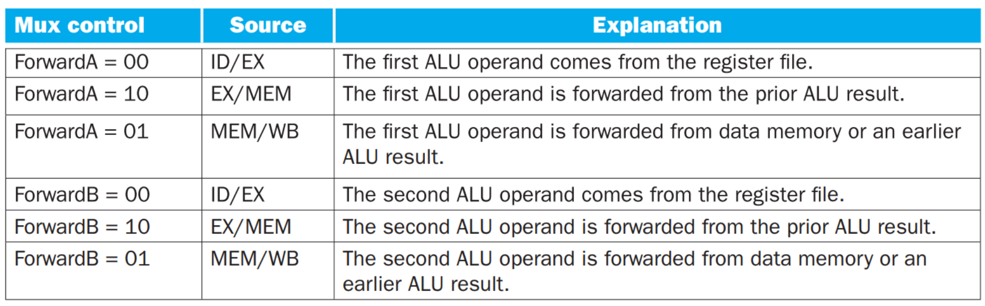
```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
     stall the pipeline
```

Figure 4.1.1: Snippet of Hazarding conditions (Patterson and Hennessy 300)



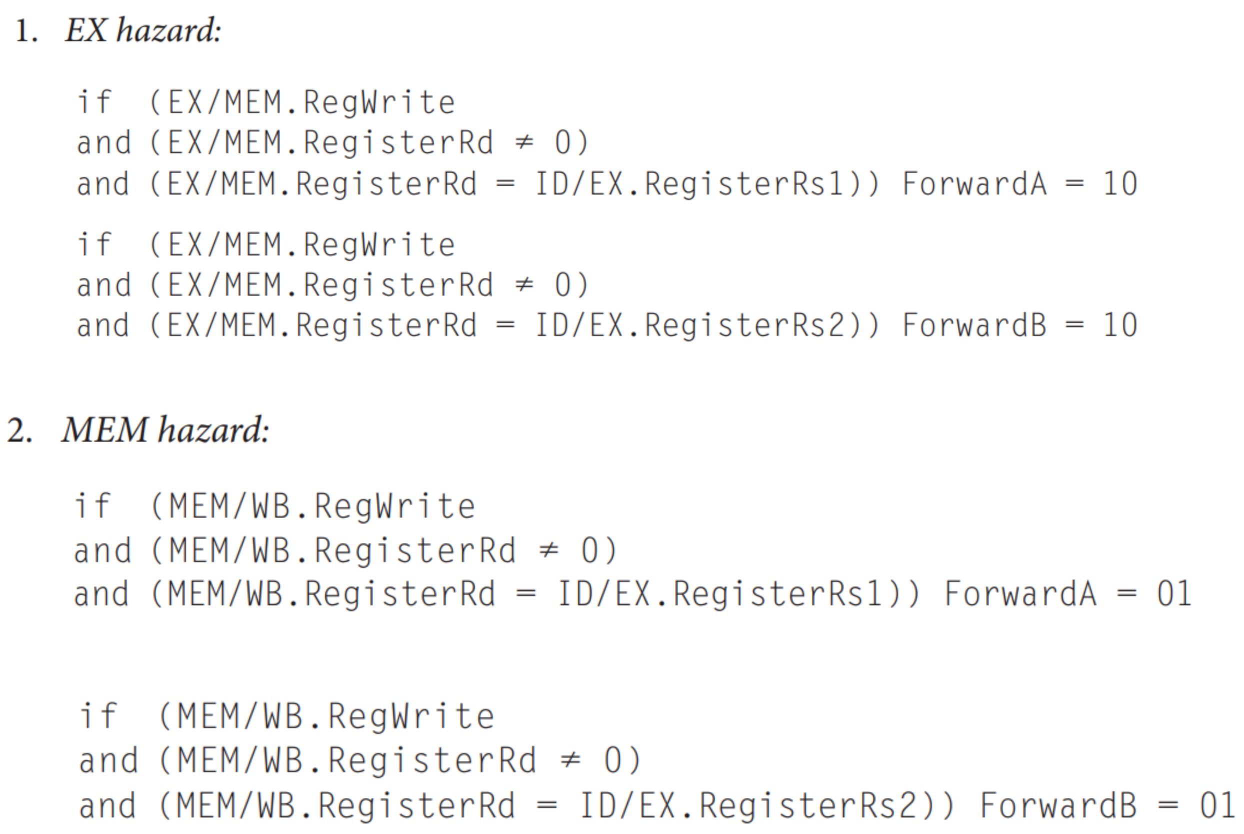
```
module Hazard_Detection
(
    input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
    input IDEX_MemRead,
    output reg IDEX_mux_out,
    output reg IFID_Write, PCWrite
);

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
```

4.1.2 Forwarding Unit Module



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.

Figure 4.1.2: Snippet of Control Values (Patterson and Hennessy 300)



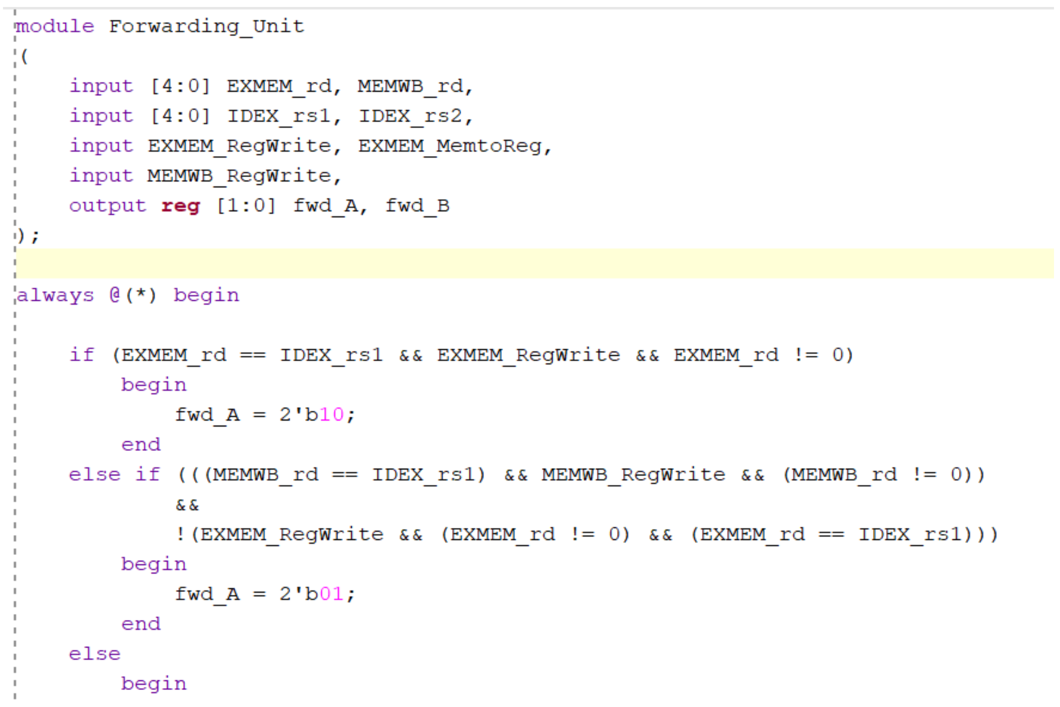
1. *EX hazard:*

```
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:*

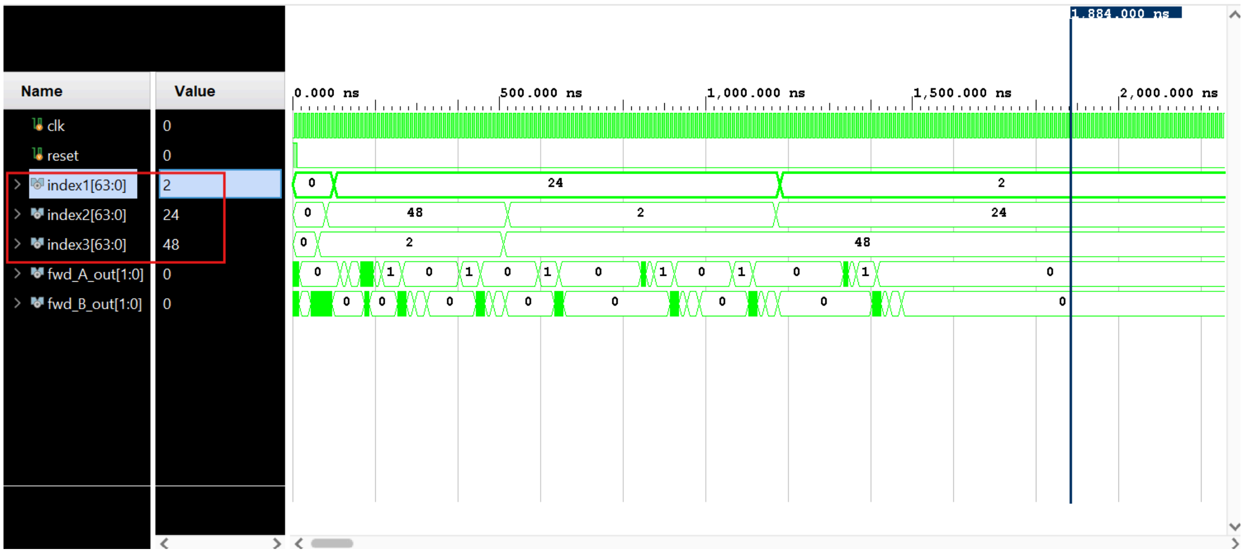
```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01


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

Figure 4.1.2: Snippet of Forwarding Conditions (Patterson and Hennessy 300)



```
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_rd == IDEX_rs1) && MEMWB_RegWrite && (MEMWB_rd != 0))
            &&
            !(EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_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_rd == IDEX_rs2) && (MEMWB_RegWrite == 1) && (MEMWB_rd != 0) )
        &&
            !( EXMEM_RegWrite && ( EXMEM_rd != 0 ) && ( EXMEM_rd == IDEX_rs2 ) )
        )
    begin
        fwd_B = 2'b01;
    end

else
    begin
        fwd_B = 2'b00;
    end
```

4.2 Simulation Output



Name	Value
clk	0
reset	0
index1[63:0]	2
index2[63:0]	24
index3[63:0]	48
fwd_A_out[1:0]	0
fwd_B_out[1:0]	0

Figure 4.2: Sorted Array

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 and Challenges

During the course of this project, we encountered several obstacles. A notable issue was the Venus Simulator's lack of support for double word instructions. This required us to dedicate time to rework the code to accommodate the execution of load double and store double instructions. Additionally, implementing the branch equals instructions proved to be a complex task. We invested considerable effort in research, ultimately discovering a solution that met our needs and functioned as intended.

7 Task Division

We worked on this entire project together since each Task was interlinked to each other. So we all were involved in the working of each Task.

8 References

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

9 Appendix

Attached below is the link Github

https://github.com/breehaqasim/CA-Project-2024.git