



# **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

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Task 1</b>	<b>2</b>
2.1	Bubble Sort Assembly Code to Machine Code .....	2
2.2	Bubble Sort Implementation Single Cycle .....	3
2.3	Simulation Output .....	3
<b>3</b>	<b>Task 2</b>	<b>4</b>
3.1	Pipelined RISC V Processor .....	4
3.2	Test Case for addi x5, x0, 2 .....	5
3.3	Simulation Output .....	5
<b>4</b>	<b>Task 3</b>	<b>5</b>
4.1	Implementing Hazard Detection Circuitry .....	5
4.2	Simulation Output .....	6
<b>5</b>	<b>Performance Comparison</b>	<b>6</b>
<b>6</b>	<b>Conclusion and Challenges</b>	<b>6</b>
<b>7</b>	<b>Task Division</b>	
<b>8</b>	<b>References</b>	<b>7</b>
<b>9</b>	<b>Appendix</b>	<b>7</b>

# 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.

```
1 #initializing array of length 3
2 addi x5, x0, 2 #2
3 sw x5, 0x100(x0)
4 addi x5, x0, 48 #48
5 sw x5, 0x104(x0)
6 addi x5, x0, 24 #24
7 sw x5, 0x108(x0)
8
9 addi x10, x10, 0x100
10 addi x11, x0, 3
11
12 bne x10, x0, ELSE
13 bne x11, x0, ELSE
14 ELSE: addi x18, x0, 0 # i
15 BUBBLE1: beq x18, x11, EXIT1
16     add x19, x0, x18 # j=i
17     BUBBLE2:
18         beq x19, x11, EXIT2
19         slli x5, x18, 2 # calculating offset of i
20         slli x6, x19, 2 # calculating offset of j
21         add x5, x5, x10
22         add x6, x6, x10
23         lw x28, 0(x5) # accessing value of a[i]
24         lw x29, 0(x6) # accessing value of a[j]
25         bge x28, x29, BUBBLE2_CONTD #if a[i] >= a[j]
26         add x30, x0, x28 # temp = a[i]
27         add x28, x0, x29 # a[i] = a[j]
28         add x29, x0, x30 # a[j] = temp
29         sw x28, 0(x5)
30         sw, x29, 0(x6)
31         BUBBLE2_CONTD: addi x19, x19, 1 # j += 1
32         beq x0, x0, BUBBLE2
33     EXIT2: addi x18, x18, 1 # i += 1
34         beq x0, x0, BUBBLE1
35 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.

```

13 | //the binary values are assigned on basis of instruction format it is of 32-bit
14 |
15 | // addi x5, x0, 2
16 | //opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000000010 (2)
17 | memory[0] = 8'b10010011;
18 | memory[1] = 8'b00000010;
19 | memory[2] = 8'b00100000;
20 | memory[3] = 8'b00000000;
21 |
22 | // sd x5, 0x100 (x0)
23 | //opcode: 0100011, rs2: 00101 (5), rs1: 00000 (0), immediate: 0001 0000 0000 (0x100), funct3: 011
24 | memory[4] = 8'b00100011;
25 | memory[5] = 8'b00110010;
26 | memory[6] = 8'b01010000;
27 | memory[7] = 8'b00000000;
28 |
29 | // addi x5, x0, 48
30 | // opcode: 0010011, rd: 00101 (5), funct3: 000, rs1: 00000 (0), immediate: 000000110000 (48)
31 | memory[8] = 8'b10010011;
32 | memory[9] = 8'b00000010;
33 | memory[10] = 8'b00000000;
34 | memory[11] = 8'b00000011;
35 |

```

## 2.2.2 Data Memory

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

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

Figure 2.2.2: Initialisation of Array

```
8      output [63:0] element2,
9      output [63:0] element3
10     );
11
12     reg [7:0] memory [63:0];
13     reg [63:0] temp_data;
14     integer i;
15     //since we are only concerned with the array values initialised in our code we'll only assign them
16     //here to element and we've assumed size to be of 8 bits
17     assign element1 = memory[20]; //24
18     assign element2 = memory[12]; //48
19     assign element3 = memory[4]; //2
20     //assign element4 = memory[28];
21     //assign element5 = memory[36];
22
23     initial begin
24     for (i=0 ;i<64 ; i = i + 1) begin
25     memory[i] = 0;
26     end
27     end
28
29
30     always @(negedge clk) begin
```

## 2.2.3 ALU Control

```
1 module ALU_Control
2 {
3     input [1:0] ALUOp,
4     input [3:0] Funct,
5     output reg [3:0] Operation
6 };
7
8     always @ (*)
9     begin
10         case(ALUOp)
11         2'b00: //slli case
12         begin
13             case({Funct[2:0]})
14             3'b001:
15             begin
16                 Operation= 4'b0111; //slli
17             end
18             default:
19             begin
20                 Operation= 4'b0010; //Add
21             end
22             endcase
23         end
24         2'b01: //branch case
```

```

24 begin
25     case ({Funct[2:0]})
26     3'b000:
27         begin
28             Operation= 4'b0110; //BEQ
29         end
30     3'b001:
31         begin
32             Operation = 4'b0110; //BNE
33         end
34     3'b101:
35         begin
36             Operation=4'b0110; //BGE
37         end
38     endcase
39 end
40
41 2'b10: // check of and or add sub case
42 begin
43     case(Funct)
44     4'b0000:
45         begin
46             Operation = 4'b0110; //sub
47         end
48     4'b0111:
49         begin
50             Operation = 4'b0000; //and
51         end
52     4'b0110:
53         begin
54             Operation = 4'b0001; //or
55         end
56     endcase
57 end
58 endcase
59 end
60 endmodule

```

In Branch case, we used funct3 values from RISC-V 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
jalu	I	1100111	000	67/0

## 2.2.4 Branch Module

```

1 //`timescale 1ns / 1ps
2 module Branch(
3     input Branch,
4     input ZERO,
5     input Isgreater,
6     input [3:0] funct,
7     output reg switch_branch
8 );
9
10 always @(*) begin
11     if(Branch) begin
12         case({funct[2:0]})
13         3'b000: switch_branch = ZERO ? 1:0;
14         3'b001: switch_branch = ZERO ? 0:1;
15         3'b101: switch_branch = Isgreater ? 1:0;
16         endcase
17     end
18     else
19         switch_branch=0;
20     end
21 endmodule
22

```

## 2.2.5 Control Unit

```
1 module Control_Unit
2 (
3     input [6:0] Opcode,
4     output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite,
5     output reg [1:0] ALUOp
6 );
7     always @ (*)
8     begin
9         case (Opcode)
10            7'b0110011: //R type
11            begin
12                Branch = 0;
13                MemRead = 0;
14                MemtoReg = 0;
15                MemWrite = 0;
16                ALUSrc = 0;
17                RegWrite = 1;
18                ALUOp = 2'b10;
19            end
20            7'b0000011: //ld
21            begin
22                Branch = 0;
23                MemRead = 1;
24                MemtoReg = 1;
25                MemWrite = 0;
26                ALUSrc = 1;
27                RegWrite = 1;
28                ALUOp = 2'b00;
29            end
30            7'b0010011: //addi
31            begin
32                Branch = 1'b0;
33                MemRead = 1'b0;
34                MemtoReg = 1'b0;
35                MemWrite = 1'b0;
36                ALUSrc = 1'b1;
37                RegWrite = 1'b1;
38                ALUOp = 2'b00;
39            end
40            7'b0100011: //S type
41            begin
42                Branch = 0;
43                MemRead = 0;
44                MemtoReg = 1'bX;
```

```

45 | ○ | MemWrite = 1;
46 | ○ | ALUSrc = 1;
47 | ○ | RegWrite = 0;
48 | ○ | ALUOp = 2'b00;
49 | ○ | end
50 | ○ | 7'b1100011: //SB
51 | ○ | begin
52 | ○ | Branch = 1;
53 | ○ | MemRead = 0;
54 | ○ | MemtoReg = 1'bX;
55 | ○ | MemWrite = 0;
56 | ○ | ALUSrc = 0;
57 | ○ | RegWrite = 0;
58 | ○ | ALUOp = 2'b01;
59 | ○ | end
60 | ○ | endcase
61 | ○ | end
62 | ○ | endmodule
63 |
64 |

```

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

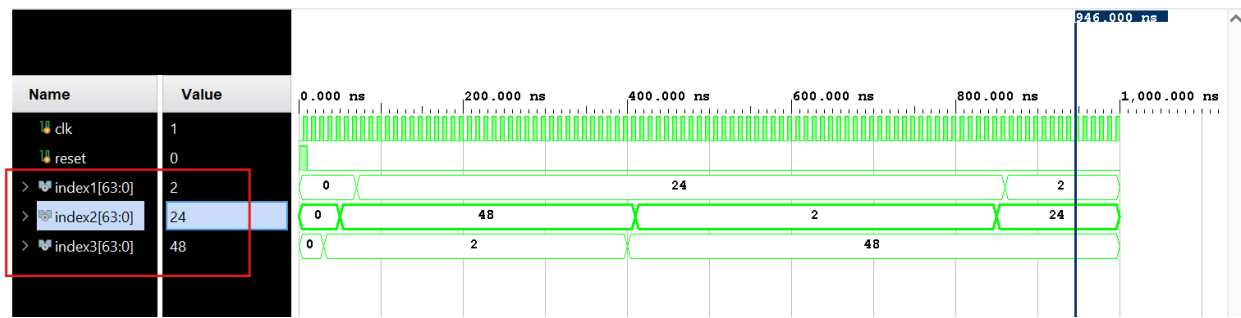


Figure 2.3: Sorted Array



## 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] readdatal,  
    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] readdatal_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;  
        readdatal_store = readdatal;  
        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

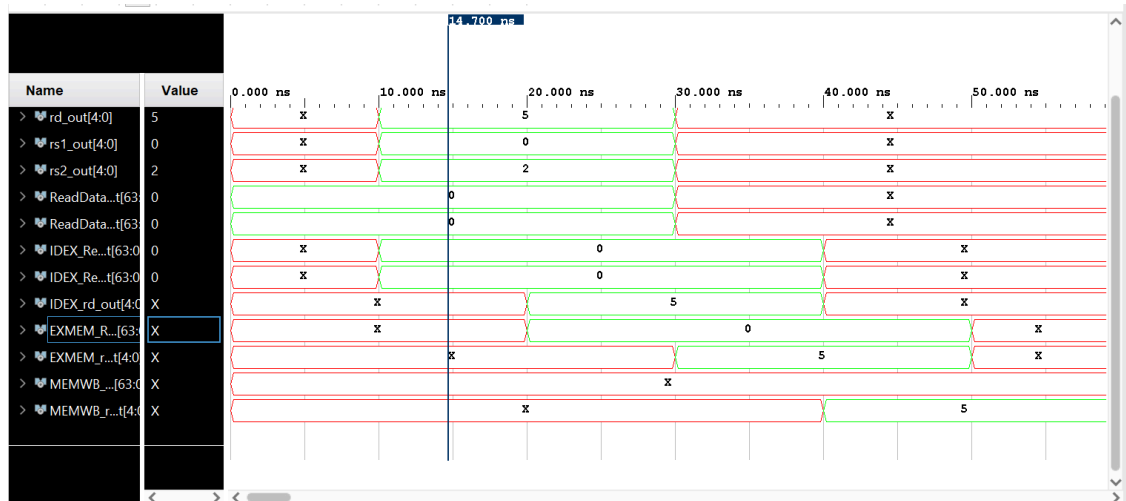


Figure 3.2.1: Snippet of Simulation Output for Test Case

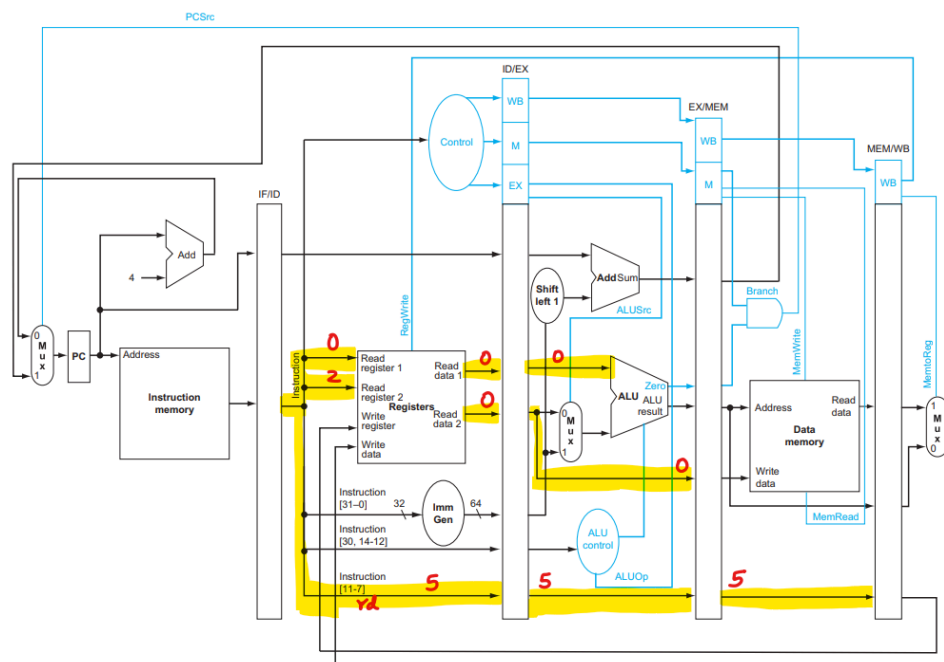


Figure 3.2.2: Tracing of instruction on Pipelined Data Path

### 3.3 Simulation Output

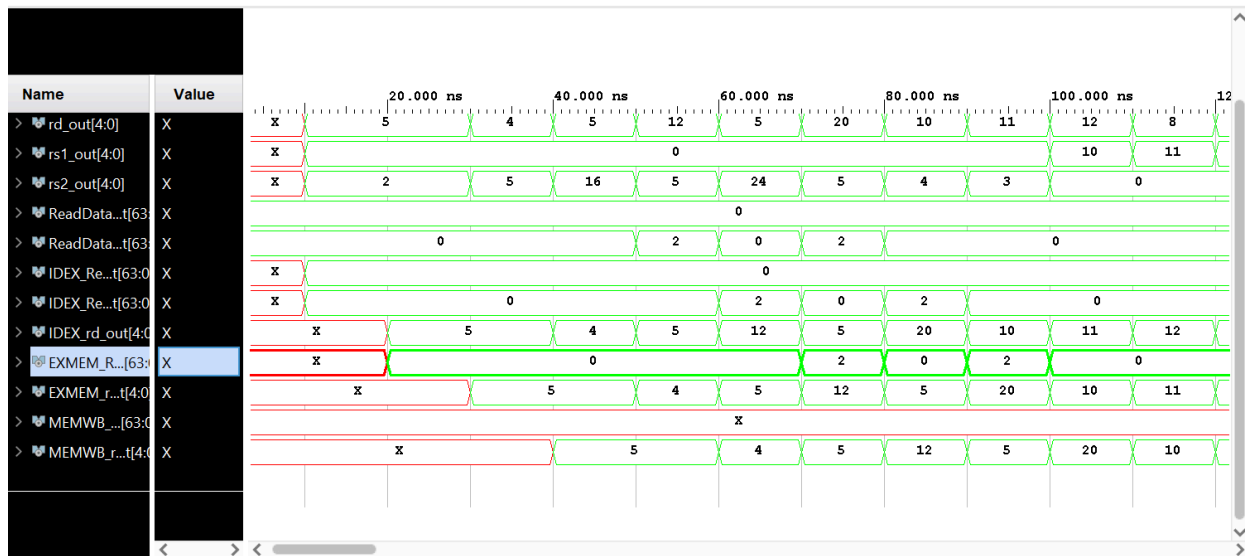


Figure 3.3: Snippet of Simulation Output for all Instructions

## 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

```

Here stalling happens

## 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
    end
end

```

## 4.2 Simulation Output

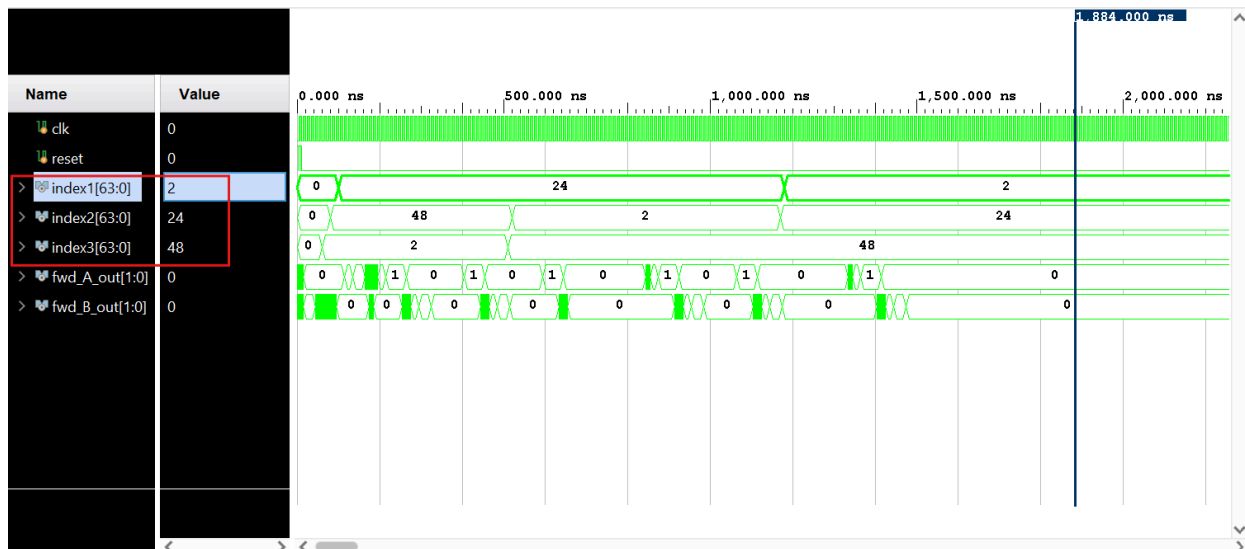


Figure 4.2: Sorted Array

## 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>

