RISC V Pipelined Processor

Final Report

Submitted by: Ali Imran, Muhammad Saad, Abdul Rafay

Research Assistant:
Abeera Farooq

Course Instructor:
Farhan Khan

Computer Architecture Spring 2024

Introduction:

Our project focused on creating a 5-stage pipelined RISC-V processor to run a bubble sort algorithm. The project encompassed several key tasks:

1. Converting the bubble sort algorithm's pseudocode into RISC-V assembly language and verifying its accuracy using the Venus simulator.
2. Modifying a single-cycle processor from a previous lab (Lab 11) to execute the bubble sort algorithm written in Lab 4 Task 3 on the Venus simulator.
3. Introducing pipelining to the processor and performing a series of test cases to validate the pipelined variant's functionality.
4. Implementing hazard detection mechanisms to recognize various hazards such as data, control, and structural hazards, and mitigating these hazards using techniques like data forwarding, stalling, and pipeline flushing.
5. Comparing the execution times needed to sort an array using both the Single Cycle Processor and the newly developed pipelined RISC-V Processor.

Task 1:

Bubble Sort Assembly Code to Machine Code

We first executed the sorting algorithm using RISC V assembly language within the Venus simulator environment

```
# Bubble sort
# x10: address of the array
# x11: number of elements in the array

li x10, 0x100
li x11, 4

li x2, 5
li x3, 9
li x4, 1
li x5, 2

sw x2, 0(x10)
sw x3, 4(x10)
sw x4, 8(x10)
sw x5, 12(x10)

sort:
    addi x21, x10, 0     # x21 = address of the array (v)
    addi x22, x11, 0     # x22 = number of elements (n)

```

```
    addi x19, x0, 0      # i = 0
for1tst:
    blt x19, x22, continue1  # if i < n, continue; else, end of routine
    beq x0, x0, finish       # Jump to end of routine (finish)
continue1:
    addi x20, x19, -1    # j = i - 1

for2tst:
    blt x20, x0, exit2   # if j < 0, exit inner loop
    slli x5, x20, 2      # x5 = j * 8
    add x5, x21, x5      # x5 = v + (j * 8)
    lw x6, 0(x5)         # x6 = v[j]
    lw x7, 4(x5)         # x7 = v[j + 1]
    blt x6, x7, exit2    # if x6 < x7, exit inner loop

    # Swap elements
    sw x7, 0(x5)         # v[j] = x7
    sw x6, 4(x5)         # v[j + 1] = x6

    addi x20, x20, -1    # j -= 1
    beq x0, x0, for2tst  # repeat the inner loop

exit2:
    addi x19, x19, 1     # i += 1
    beq x0, x0, for1tst  # repeat the outer loop

finish:
    # Program reaches here and simply ends.
    # No loop, no return—just the end of the sorting routine.

```

Before:

0x0000010c	02	00	00	00
0x00000108	01	00	00	00
0x00000104	09	00	00	00
0x00000100	05	00	00	00

After:

0x0000010c	09	00	00	00
0x00000108	05	00	00	00
0x00000104	02	00	00	00
0x00000100	01	00	00	00

Bubble Sort Implementation Single Cycle

We made slight adjustments to the Lab 11 module, where we instantiated all modules together to construct the processor. The modifications were made to the Instruction Memory, Data Memory, Register File, Branch, ALU Control, and ALU_64_Bit modules. The most significant changes were observed in the Instruction Memory module, while the other modules underwent minor modifications. These minor modifications included adding support for blt and slli instructions which were needed for the successful running of bubble sort.

Instruction Memory

Task (a) Initializing Values:

//initializing array of length 4 with different values
//the binary values are assigned on basis of instruction format it is of 32-bit

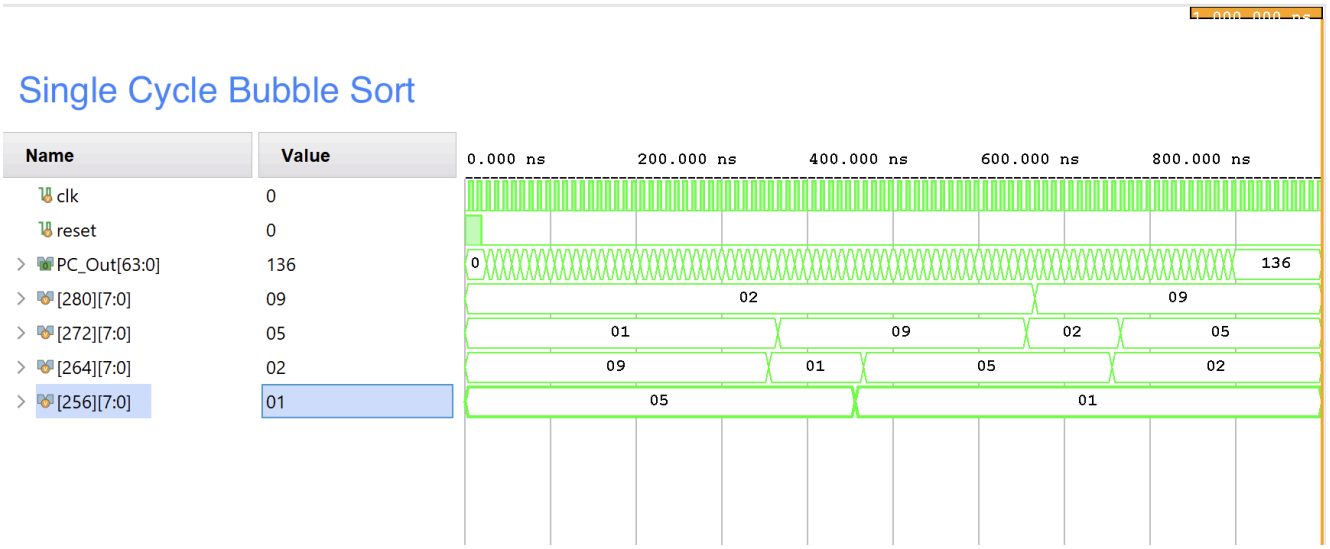
// Bubble Sort
IM[0] <= 8'h13; // addi x10, x0, 0
IM[1] <= 8'h05;
IM[2] <= 8'h00;
IM[3] <= 8'h10;
IM[4] <= 8'h93; // addi x11, x0, 4
IM[5] <= 8'h05;
IM[6] <= 8'h40;
IM[7] <= 8'h00;
IM[8] <= 8'h93; // addi x21, x10, 0
IM[9] <= 8'h0a;
IM[10] <= 8'h05;
IM[11] <= 8'h00;
IM[12] <= 8'h13; // addi x22, x11, 0
IM[13] <= 8'h8b;
IM[14] <= 8'h05;
IM[15] <= 8'h00;
IM[16] <= 8'h93; // addi x19, x0, 0
IM[17] <= 8'h09;
IM[18] <= 8'h00;
IM[19] <= 8'h00;
IM[20] <= 8'h63; // blt x19, x22, 8 (continue)
IM[21] <= 8'hc4;
IM[22] <= 8'h69;
IM[23] <= 8'h01;
IM[24] <= 8'h63; // beq x0, x0, 56 (finish)
IM[25] <= 8'h0c;
IM[26] <= 8'h00;
IM[27] <= 8'h02;
IM[28] <= 8'h13; // addi x20, x19, -1 # continue
IM[29] <= 8'h8a;

```
IM[30] <= 8'hf9;
IM[31] <= 8'hff;
IM[32] <= 8'h63; // blt x20, x0, 40 (exit2)
IM[33] <= 8'h44;
IM[34] <= 8'h0a;
IM[35] <= 8'h02;
IM[36] <= 8'h93; // slli x5 x20 2
IM[37] <= 8'h12;
IM[38] <= 8'h3a;
IM[39] <= 8'h00;
IM[40] <= 8'hb3; // add x5 x21 x5
IM[41] <= 8'h82;
IM[42] <= 8'h5a;
IM[43] <= 8'h00;
IM[44] <= 8'h03; // ld x6, 0(x5)
IM[45] <= 8'hb3;
IM[46] <= 8'h02;
IM[47] <= 8'h00;
IM[48] <= 8'h83; // ld x7, 8(x5)
IM[49] <= 8'hb3;
IM[50] <= 8'h82;
IM[51] <= 8'h00;
IM[52] <= 8'h63; // blt x6, x7, exit2
IM[53] <= 8'h4a;
IM[54] <= 8'h73;
IM[55] <= 8'h00;
IM[56] <= 8'h23; // sd x7, 0(x5)
IM[57] <= 8'hb0;
IM[58] <= 8'h72;
IM[59] <= 8'h00;
IM[60] <= 8'h23; // sd x6, 8(x5)
IM[61] <= 8'hb4;
IM[62] <= 8'h62;
IM[63] <= 8'h00;
IM[64] <= 8'h13; // addi x20, x20, -1
IM[65] <= 8'h0a;
```

```
IM[66] <= 8'hfa;
IM[67] <= 8'hff;
IM[68] <= 8'he3; // beq x0, x0, for2
IM[69] <= 8'h0e;
IM[70] <= 8'h00;
IM[71] <= 8'hfc;
IM[72] <= 8'h93; // addi x19, x19, 1
IM[73] <= 8'h89;
IM[74] <= 8'h19;
IM[75] <= 8'h00;
IM[76] <= 8'he3; // beq x0, x0, For1
IM[77] <= 8'h04;
IM[78] <= 8'h00;
IM[79] <= 8'hfc;
```

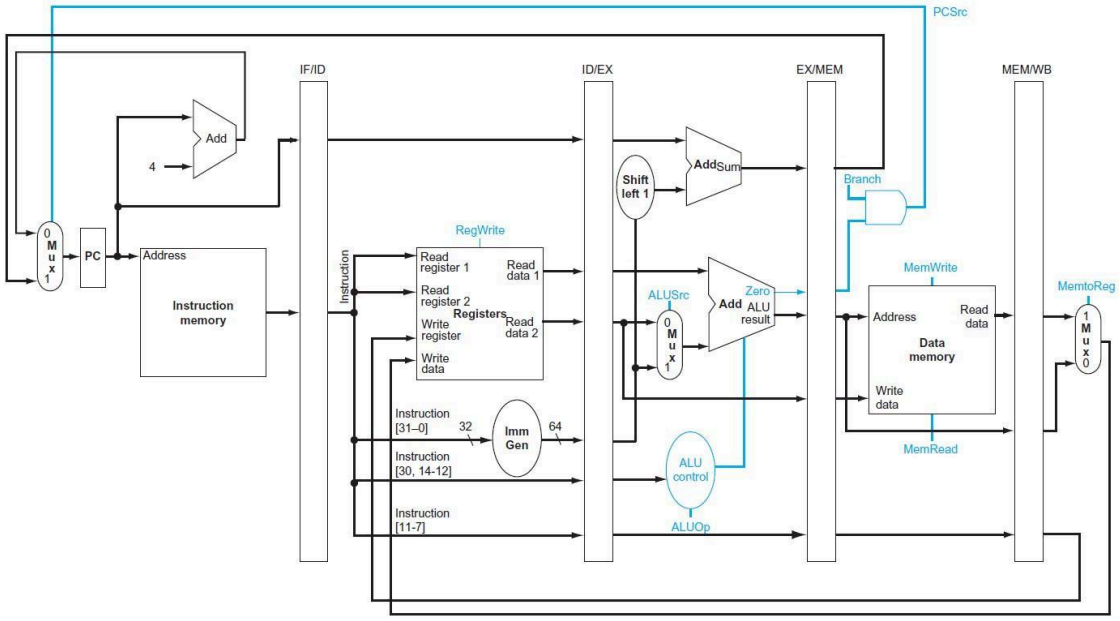
Task (b) Assigning the array to registers for viewing the values swaps:

```
assign  = memory[256]; //5
assign = memory[264]; //9
assign = memory[272]; //1
assign = memory[280]; //2
```



Task 2:

Initially, we successfully implemented the algorithm on a single-cycle processor. Subsequently, we upgraded the processor to a pipelined version. In accordance with our textbook's teachings, we incorporated pipeline registers, namely IF/ID, ID/EX, EX/MEM, and MEM/WB. These registers are responsible for retaining data from one pipeline stage to the next. To ensure the proper functioning of each pipeline stage, we meticulously tested instructions individually.



Pipelined Register 1 - IF/ID:

```
module IF_ID(input clk, input reset, input IF_ID_Write, input IF_Flush, input
wire [63:0] PC_Address_In, input wire [31:0] Inst_In,
output reg [63:0] PC_Address_Out, output reg [31:0] Inst_Out);

initial
   Inst_Out = 0;

always @(posedge clk)
begin
   if (IF_ID_Write)
   begin
      PC_Address_Out = PC_Address_In;
```

```
      Inst_Out = Inst_In;
   end
   if (IF_Flush || reset)
      Inst_Out = 0;
end

endmodule
```

Pipelined Register 2 - ID/EX:

```
module ID_EX(input clk,
input wire MemToReg, input wire RegWrite,
input wire Branch, input wire MemWrite, input wire MemRead,
input wire [1:0] ALUOp, input wire ALUSrc,
input wire [63:0] PC_In, input wire [63:0] imm_data, input wire [63:0]
ReadData1, input wire [63:0] ReadData2,
input wire [4:0] RS1, input wire [4:0] RS2, input wire [4:0] RD, input wire [3:0]
Funct,

output reg memtoreg, output reg regwrite,
output reg branch, output reg memwrite, output reg memread,
output reg [1:0] aluop, output reg alusrc,

output reg [63:0] BranchA, output reg [63:0] BranchB,

output reg [63:0] readdata1, readdata2,

output reg [4:0] rs1, output reg [4:0] rs2, output reg [4:0] rd, output reg [3:0]
funct);

initial
begin
   memtoreg <= 0;
   regwrite <= 0;
   branch <= 0;
   memwrite <= 0;
   memread <= 0;
   aluop <= 0;
```

```
    alusrc <= 0;
end

always @(posedge clk)
begin
    memtoreg <= MemToReg;
    regwrite <= RegWrite;
    branch <= Branch;
    memwrite <= MemWrite;
    memread <= MemRead;
    aluop <= ALUOp;
    alusrc <= ALUSrc;
    BranchA <= PC_In;
    BranchB <= imm_data;
    readdata1 <= ReadData1;
    readdata2 <= ReadData2;
    rs1 <= RS1;
    rs2 <= RS2;
    rd <= RD;
    funct <= Funct;
end
endmodule
```

Pipelined Register 3 - EX/MEM:

```
module EX_MEM(
    input clk, input flush,

    input wire MemToReg, input wire RegWrite,
    input wire MemRead, input wire MemWrite, input wire Branch,

    input wire [63:0] BranchAddress, input Zero, input Lt,
    input wire [63:0] ALUResult, input wire [63:0] MemWriteData, input wire
[4:0] RD,
    input [3:0] Funct,

    output reg memtoreg, regwrite, memread, memwrite, branch,
```

```
    output reg [63:0] branchaddress,
    output reg zero, lt,
    output reg [63:0] aluresult, memwritedata,

    output reg [4:0] rd,
    output reg [3:0] funct
    );

initial
begin
    memtoreg <= 0;
    regwrite <= 0;
    branch <= 0;
    memwrite <= 0;
    memread <= 0;
end

    always @ (posedge clk)
    begin
        if (flush)
        begin
            memtoreg <= 0;
            regwrite <= 0;
            memread <= 0;
            memwrite <= 0;
            branch <= 0;
        end
        else
        begin
            memtoreg <= MemToReg;
            regwrite <= RegWrite;
            memread <= MemRead;
            memwrite <= MemWrite;
            branch <= Branch;
        end
        branchaddress <= BranchAddress;
        zero = Zero;
```

```
        lt = Lt;
        aluresult <= ALUResult;
        memwritedata <= MemWriteData;
        rd <= RD;
        funct <= Funct;
    end

endmodule
```

Pipeline Register 4 - MEM/WB:

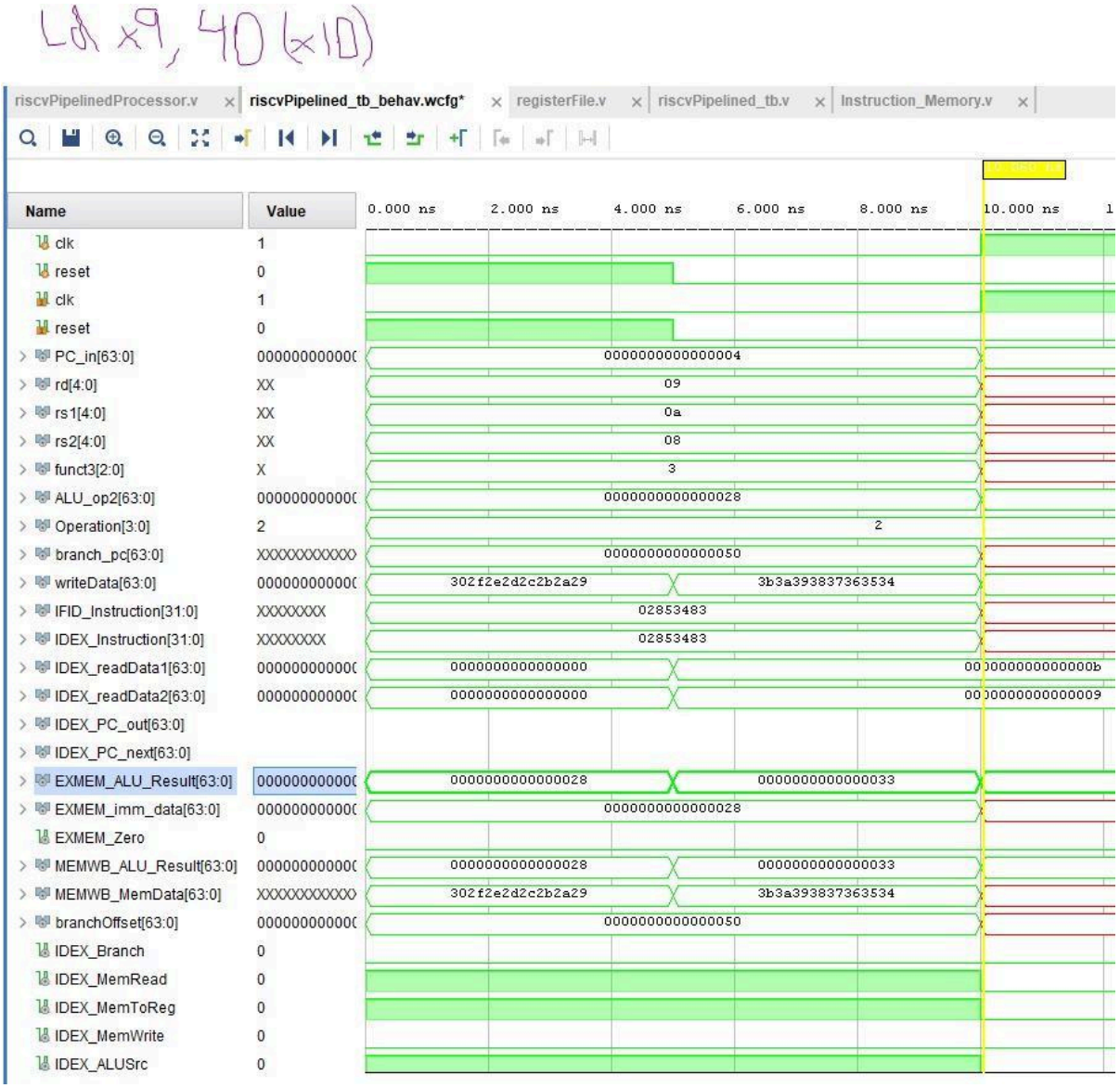
```
module MEM_WB(
    input clk,

    input wire MemToReg, RegWrite,

    input wire [63:0] Read_Data, ALUResult,

    input wire [4:0] RD,

    output reg memtoreg, regwrite,

    output reg [63:0] read_data, aluresult,

    output reg [4:0] rd

    );

initial
begin
    memtoreg <= 0;
    regwrite <= 0;
end

    always @ (posedge clk)
    begin
        memtoreg <= MemToReg;
        regwrite <= RegWrite;
```

```
        read_data <= Read_Data;
        aluresult <= ALUResult;
        rd <= RD;
    end
endmodule
```

Explanation:

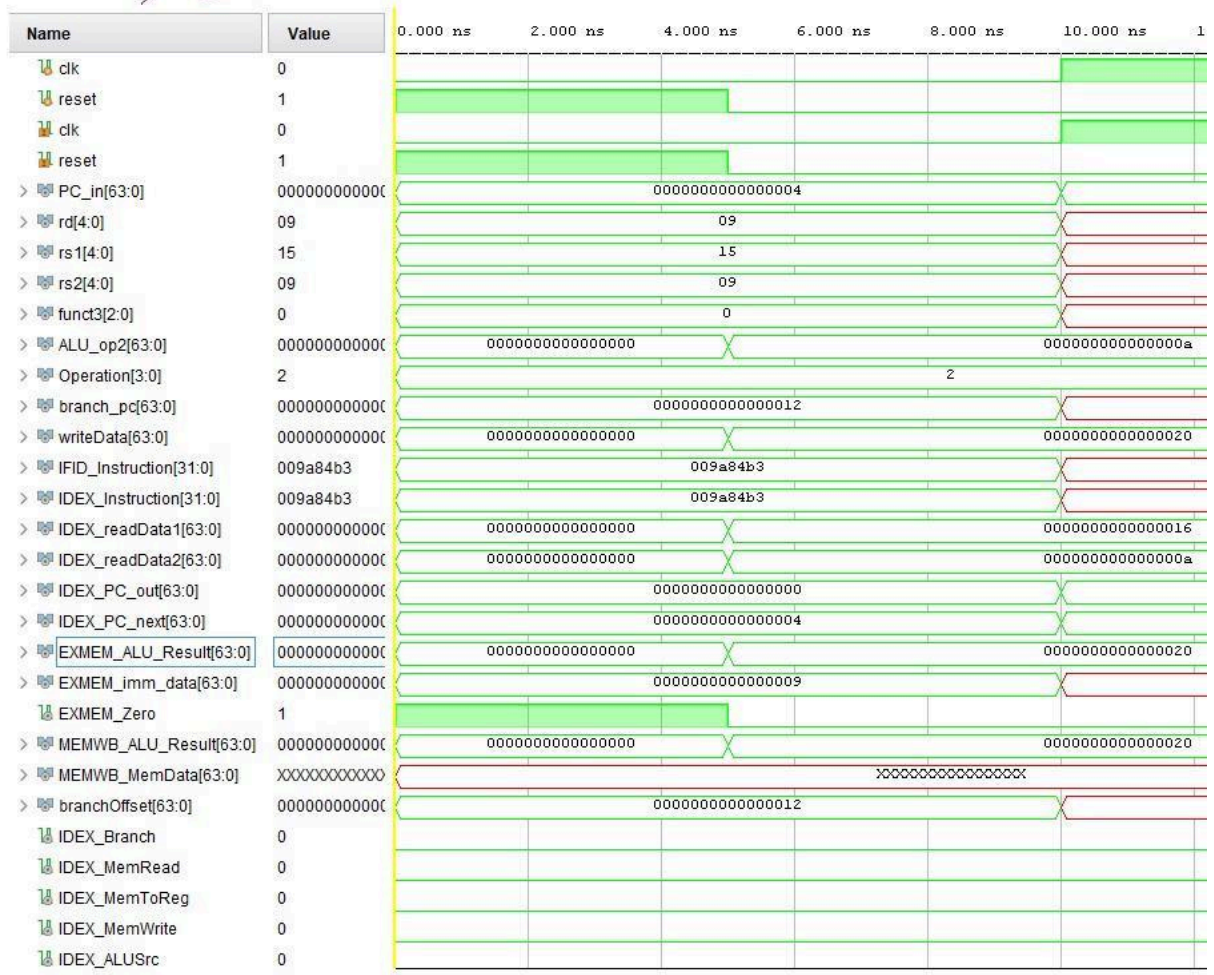
Each pipelined stage contains input and output registers needed for forwarding the registers from one pipelined stage to the next. The top module sends the registers value from the previous to the next.

Screenshots of Task 2 Testing:

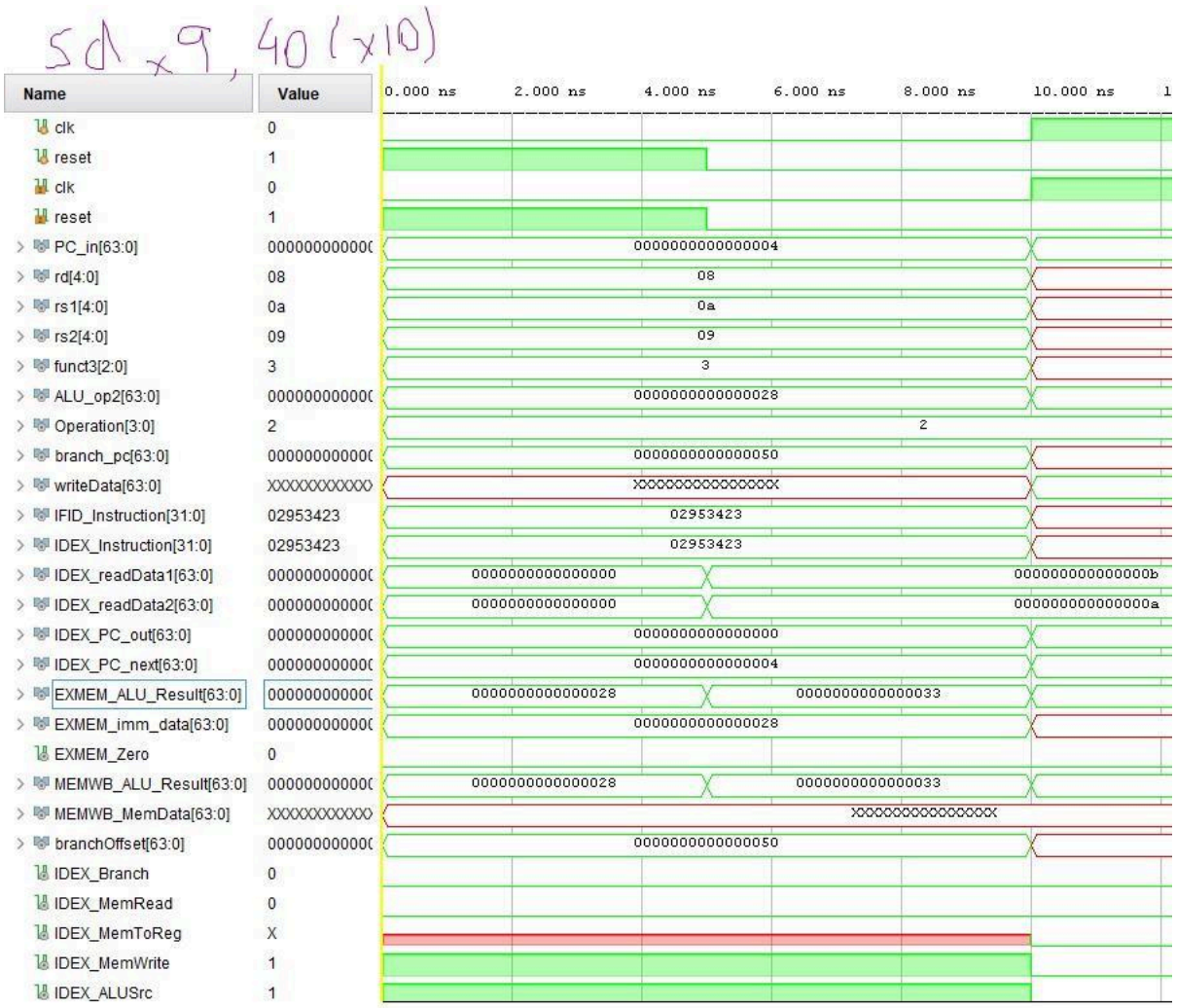


Testing Instruction 1: // ld x9, 40(x10)

add x9, x21, x9

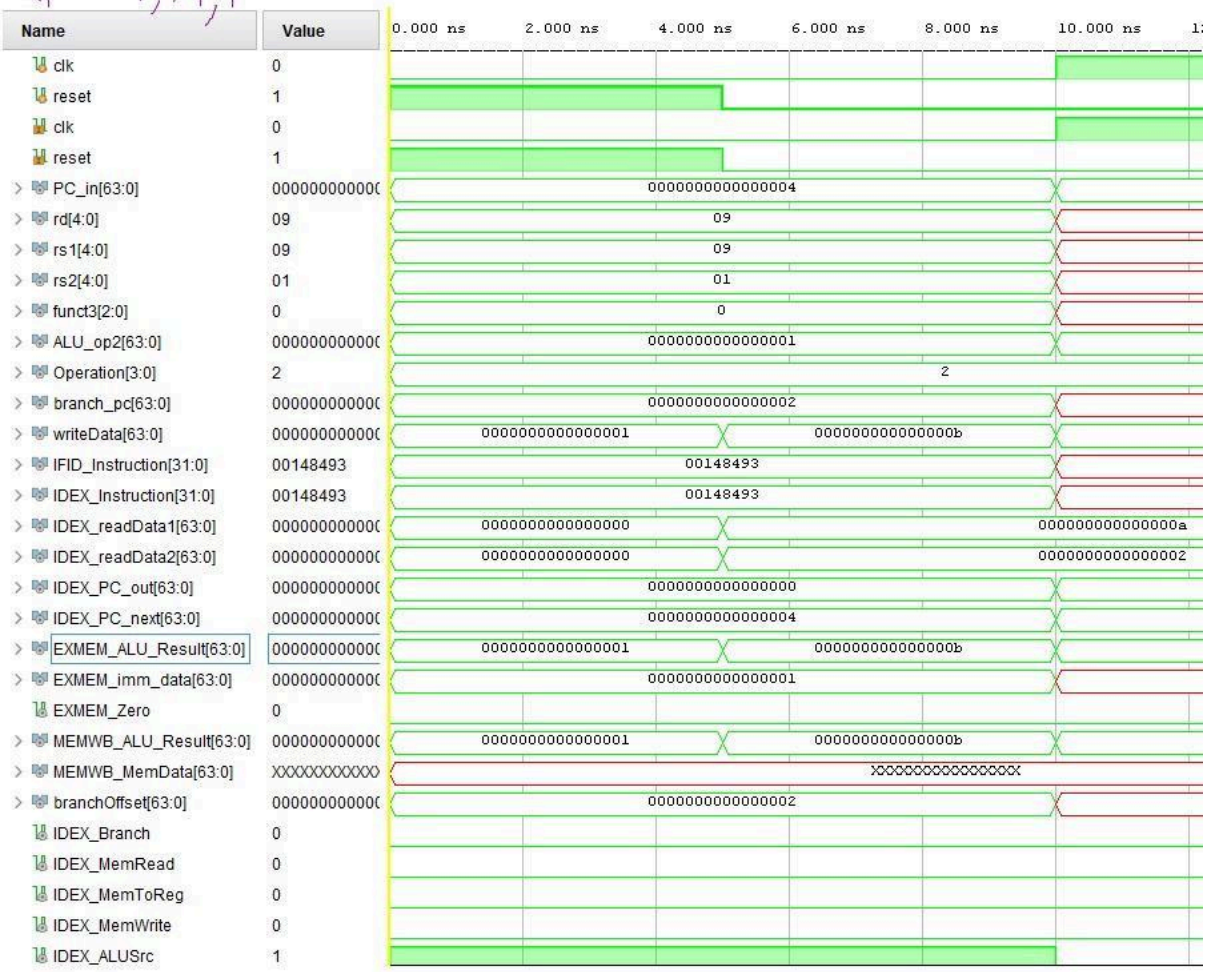


Testing Instruction 2: // add x9, x21, x9



Testing Instruction 3: // sd x9, 40(x10)

addi x9, x9, 1



Testing Instruction 3: // addi x9, x9, 1

Task 3:

Hazard Detection Unit:

Processors rely on pipelines to execute instructions efficiently. However, these pipelines can be vulnerable to hazards like data dependencies. To prevent these issues, a hazard detection unit is incorporated into the pipeline.
This unit acts like a guardian, monitoring the instruction flow during the ID stage (instruction decode). If it detects an instruction waiting for data (e.g., a load instruction), it triggers a stall. This stall mechanism pauses the pipeline, preventing the program counter and instruction register from updating until the data is ready. By doing this, the hazard detection unit ensures smooth pipeline operation and avoids errors caused by data dependencies. In essence, the hazard detection unit acts as a proactive measure, safeguarding the pipeline from potential roadblocks and ensuring instructions are executed in the correct order.

Code:

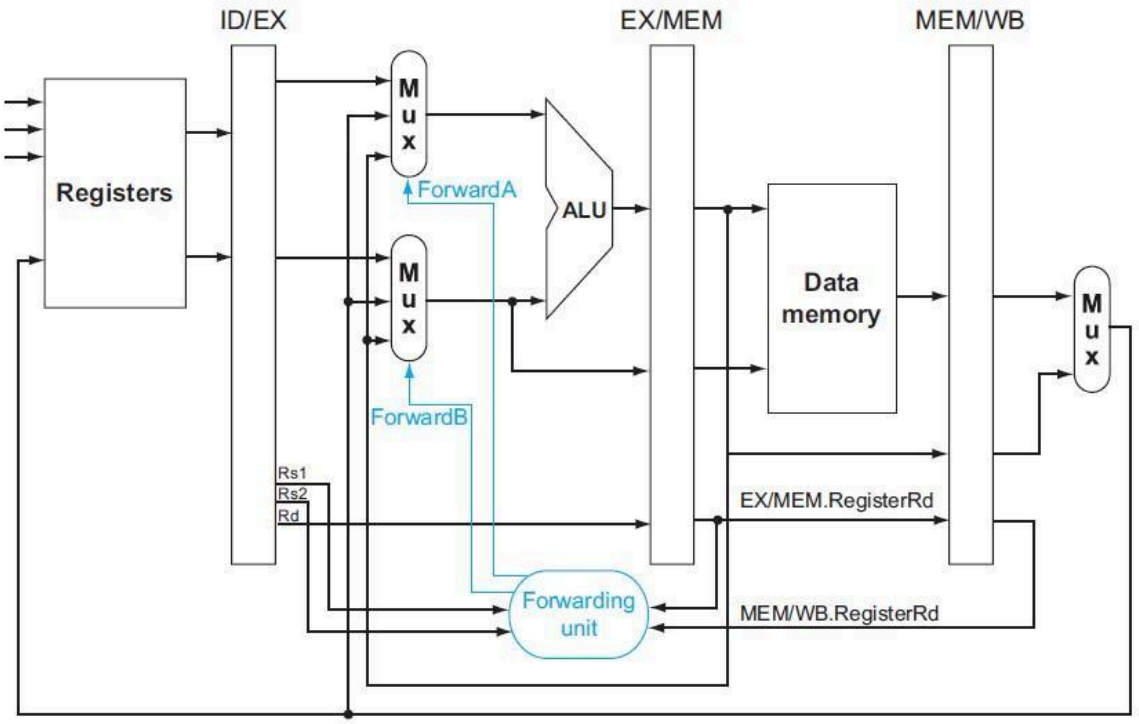
```
module hazard_unit(
   input clk,
   input wire [4:0] IF_ID_RS1, IF_ID_RS2, ID_EX_RD,
   input wire ID_EX_MemRead,
   output wire PCWrite, IF_ID_Write,
   output wire MuxSignal
   );

   assign MuxSignal = ~(ID_EX_MemRead & ((ID_EX_RD == IF_ID_RS1) ||
(ID_EX_RD == IF_ID_RS2)));
   assign PCWrite = ~(ID_EX_MemRead & ((ID_EX_RD == IF_ID_RS1) ||
(ID_EX_RD == IF_ID_RS2)));
   assign IF_ID_Write = ~(ID_EX_MemRead & ((ID_EX_RD == IF_ID_RS1) ||
(ID_EX_RD == IF_ID_RS2)));

endmodule
```

Forwarding Unit:

Modern processors employ pipelining techniques to enhance instruction throughput, yet these pipelines are susceptible to hazards such as data dependencies. To mitigate these issues and maintain efficient operation, a forwarding unit is integrated into the pipeline architecture. This unit serves as a facilitator, actively monitoring data flow within the pipeline. During the execution stage, if it detects that an instruction requires data that has been recently produced by a preceding instruction, it forwards this data directly to the dependent instruction, bypassing the memory or register file. By doing so, the forwarding unit eliminates the need for unnecessary stalls, enabling instructions to proceed without delay. In essence, the forwarding unit optimizes pipeline efficiency by dynamically routing data to where it's needed, thereby maximizing throughput and reducing latency in instruction execution.

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.



Code:

```
module fwdUnit(
   input clk,

   input [4:0] ex_mem_rd,
   input ex_mem_RW,
   input [4:0] mem_wb_rd,
   input mem_wb_RW,
   input [4:0] id_ex_rs1, id_ex_rs2,

   output wire [1:0] forwardA, forwardB
   );

   assign forwardA = (ex_mem_RW & (ex_mem_rd != 0) & (ex_mem_rd == id_ex_rs1)) ? 2'b10 : (mem_wb_RW & (mem_wb_rd != 0) & (mem_wb_rd == id_ex_rs1)) ? 2'b01 : 0;
   assign forwardB = (ex_mem_RW & (ex_mem_rd != 0) & (ex_mem_rd == id_ex_rs2)) ? 2'b10 : (mem_wb_RW & (mem_wb_rd != 0) & (mem_wb_rd == id_ex_rs2)) ? 2'b01 : 0;

endmodule
```

Instructions for testing pipelining, hazarding and forwarding:

```
 // addi x10 x0 0x100
//IM[0] <= 8'h13;
//IM[1] <= 8'h05;
//IM[2] <= 8'h00;
//IM[3] <= 8'h10;

// ld x9 0x100(x0)
//IM[0] <= 8'h83;
//IM[1] <= 8'h34;
//IM[2] <= 8'h00;
//IM[3] <= 8'h10;

// add x10 x9 x9
//IM[4] <= 8'h33;
//IM[5] <= 8'h85;
//IM[6] <= 8'h94;
//IM[7] <= 8'h00;

// load hazard + forwarding
// ld x2 0x100(x0)
// IM[0] <= 8'h03;
// IM[1] <= 8'h31;
// IM[2] <= 8'h00;
// IM[3] <= 8'h10;

// addi x2 x0 50
//IM[0] <= 8'h13;
//IM[1] <= 8'h01;
//IM[2] <= 8'h20;
//IM[3] <= 8'h03;

//// add x3 x2 x2
// IM[4] <= 8'hb3;
// IM[5] <= 8'h01;
// IM[6] <= 8'h21;
```

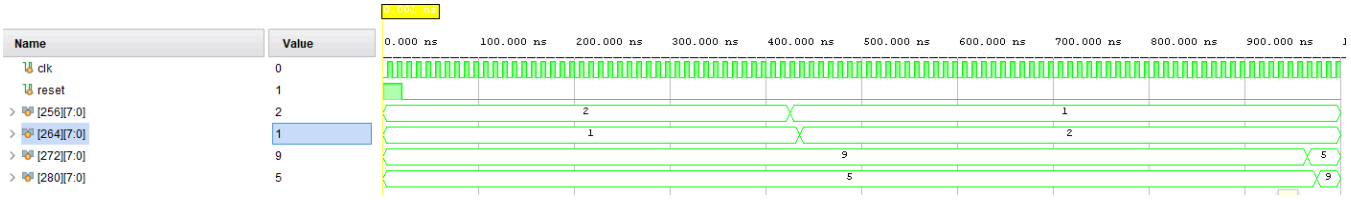
```
// IM[7] <= 8'h00;

//// add x4 x3 x2
// IM[8] <= 8'h33;
// IM[9] <= 8'h82;
// IM[10] <= 8'h21;
// IM[11] <= 8'h00;

//// add x4 x2 x2
// IM[12] <= 8'h33;
// IM[13] <= 8'h02;
// IM[14] <= 8'h21;
// IM[15] <= 8'h00;

// control hazard
////beq x0 x0 20
// IM[0] <= 8'h63;
// IM[1] <= 8'h0a;
// IM[2] <= 8'h00;
// IM[3] <= 8'h00;

////addi x1 x0 2
// IM[4] <= 8'h93;
// IM[5] <= 8'h00;
// IM[6] <= 8'h20;
// IM[7] <= 8'h00;

////addi x1 x0 3
// IM[8] <= 8'h93;
// IM[9] <= 8'h00;
// IM[10] <= 8'h30;
// IM[11] <= 8'h00;

////addi x1, x0, 4
// IM[12] <= 8'h93;
// IM[13] <= 8'h00;
// IM[14] <= 8'h40;
```

```
// IM[15] <= 8'h00;

////addi x1 x0 5
// IM[16] <= 8'h93;
// IM[17] <= 8'h00;
// IM[18] <= 8'h50;
// IM[19] <= 8'h00;

////addi x1 x0 6
// IM[20] <= 8'h93;
// IM[21] <= 8'h00;
// IM[22] <= 8'h60;
// IM[23] <= 8'h00;
```

Simulation:

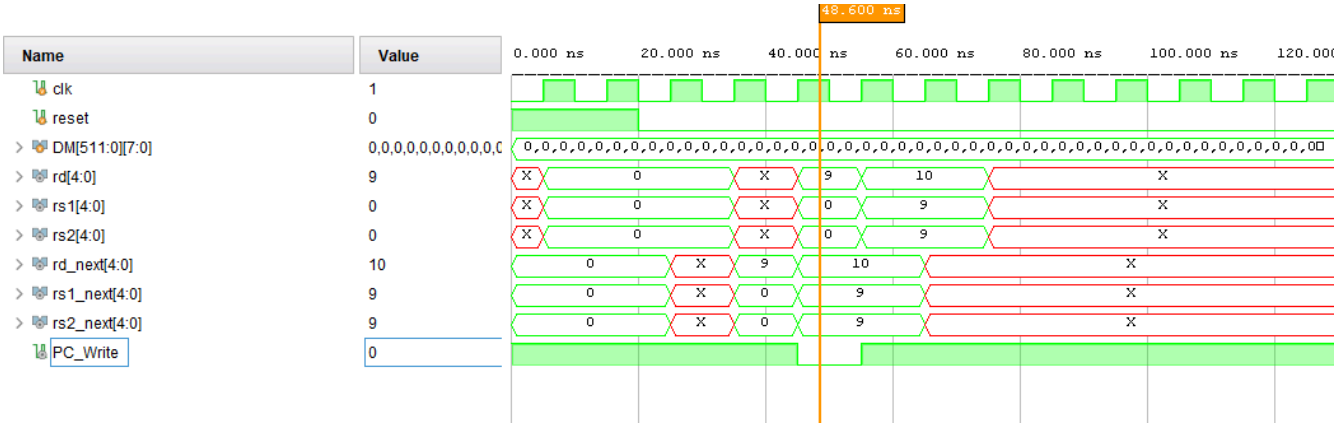
Pipelined Bubble Sorting

2, 1, 5, 9 have been sorted to 1, 2, 5, 9.



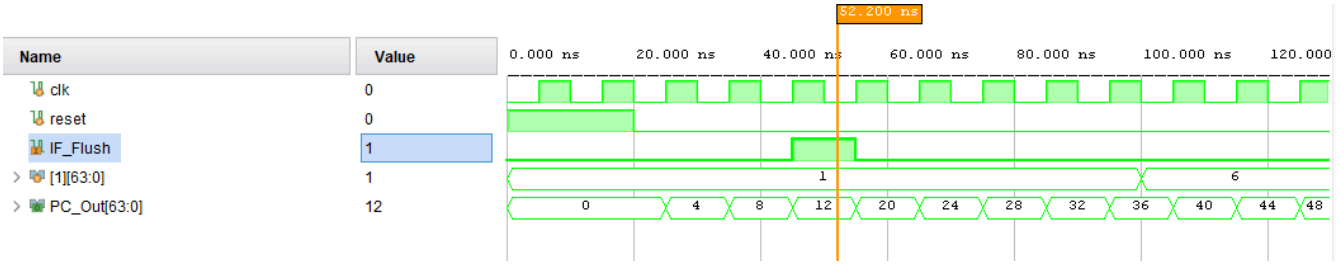
Load Use Data Hazard

PC_Write is 0 so next instruction is not fetched. A stall happens.



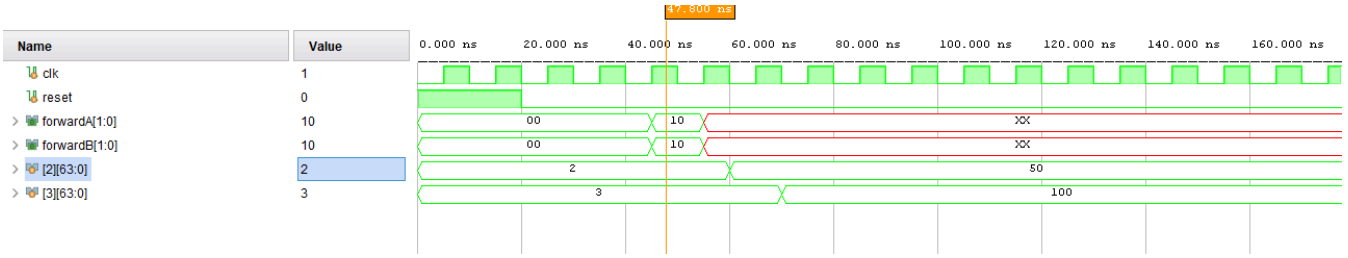
Control Hazard

IF_Flush is 1, which flushes the register values.



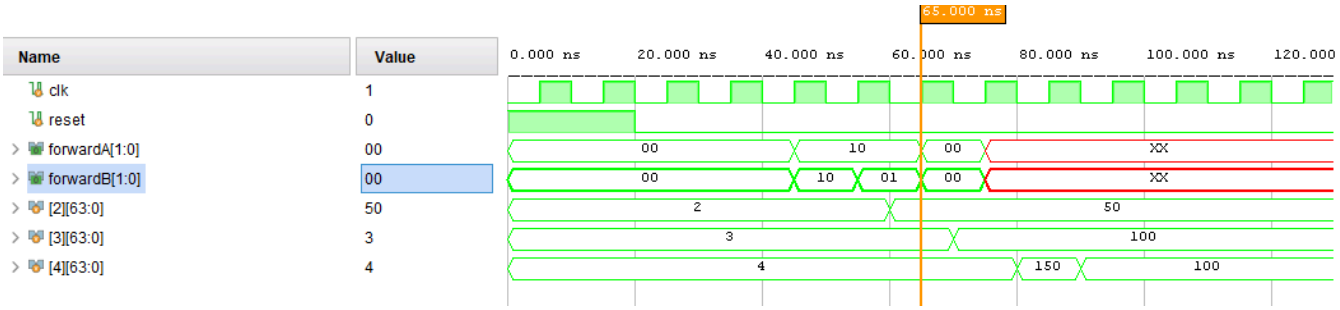
1a & 1b Forwarding

Forward and ForwardB are 10.



2b & 1a Forwarding

Forward and ForwardB are 10 initially, then forward B is 01.



Task4:

Performance Comparison

The pipelined RISC-V processor takes approximately 1450 nanoseconds to execute the bubble sort algorithm, exceeding the single-cycle processor's 1000 nanoseconds. This indicates a performance penalty of 450 nanoseconds for the pipelined design.

While pipelining offers potential performance improvements, unavoidable pipeline stalls can occur even with hazard detection and data forwarding mechanisms. These stalls introduce delays, negating some of the pipelining benefits. In this case, the stalls experienced by your pipeline outweigh the gains from pipelining, resulting in slower execution compared to the single-cycle processor.

Challenges

This project presented valuable learning experiences alongside its successful completion.

One key challenge was the limited support for double-precision instructions (load double and store double) in the Venus Simulator. We overcame this hurdle by adapting our code to function within the simulator's constraints.

Another obstacle emerged when implementing the branch-on-equal instruction. This proved to be a complex task, requiring extensive research to identify a solution that delivered the desired functionality.

Despite these challenges, we successfully navigated them, demonstrating the importance of adaptability and resourcefulness in hardware design projects.

Task Division

We worked on this entire project together.

Conclusions:

Our project successfully developed a 5-stage pipelined RISC-V processor for executing a bubble sort algorithm. We effectively integrated pipelining and implemented hazard detection mechanisms to mitigate data dependencies. However, performance comparison revealed a 450 nanosecond penalty for the pipelined processor compared to the single-cycle version, primarily due to pipeline stalls. Despite challenges, the project provided valuable learning experiences, highlighting areas for optimization in future iterations.

References:

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

Appendix:

The complete code for our project can be found here:
AuraGuardian23/riscv_pipelined_processor: Implementation of riscv pipelined processor using verilog (github.com)