RISC-V PIPELINED PROCESSOR

FINAL REPORT

COMPUTER ARCHITECTURE
SPRING 2024

Group Members: Jibran Sheikh, Muhammad Ansab Chaudhary, Daniyal Shadab

Index

1. Introduction:

For this project, we were tasked to build a Risc V Processor that is able to execute Bubble Sort Algorithm. Our Project included building a single cycle processor and then a pipelined version which is able execute the same task. Our aim was to improve the performance of the single-cycle processor by pipelining it and also implementing the Hazard detection and then comparing the 2 processors to find out which is better for bubble sort.

a. Task1:

2a) Task1 of our CA project was to implement a sorting algorithm on an array using RISC-V assembly language within the venus simulator. We broke our Task1 in the following steps:

1) Firstly we chose the bubble sort algorithm, tested it on Kwakil Venus to see if it was working or not,
2) Then we converted each line of code from Assembly to Binary and created our Instruction memory to be used for this project.
3) Then we modified the single cycle processor we created in lab 11 to run the sorting algorithm.

2b) The following figure shows our Bubble Sort algorithm in Assembly language:

```
li x10, 0x100   # array base address
addi x11, x0, 5 # number of elements = 5
li x12, 0       # i = outer loop counter
li x13, 0       # j = inner loop counter

li x5, 14 # our array of unsorted numbers
li x6, 31
li x7, 6
li x8, 17
li x9, 20

sw x5, 0x100(x0) # storing the numbers in array addresses
sw x6, 0x104(x0)
sw x7, 0x108(x0)
sw x8, 0x10c(x0)
sw x9, 0x110(x0)

loop1:
  bge x12, x11, exit # exiting loop if i >= number of elements
  addi x12, x12, 1   # i++
  li x13, 0           # resetting j to zero at the beginning of each outer loop

loop2:
  bge x13, x11, loop1 # exiting inner loop if j >= number of elements
  slli x16, x12, 2    # calculating offset for a[i]
  slli x18, x13, 2    # calculating offset for a[j]
  add x15, x16, x10   # calculating address of a[i]
  add x17, x18, x10   # calculating address of a[j]
  lw x28, 0(x15)      # loading a[i] into x28
  lw x29, 0(x17)      # loading a[j] into x29

  blt x28, x29, no_swap # skipping swap if a[i] <= a[j]
  # swapping a[i] and a[j]
  sw x29, 0(x15) # store a[j] at a[i]
  sw x28, 0(x17) # store a[i] (original value of a[j]) at a[j]
no_swap:
  addi x13, x13, 1 # increment j
  j loop2           # repeat inner loop
exit:

```

The figure below shows before sorting:

0x00000110	20	0	0	0
0x0000010c	17	0	0	0
0x00000108	6	0	0	0
0x00000104	31	0	0	0
0x00000100	14	0	0	0

The figure below shows after sorting:

0x00000110	6	0	0	0
0x0000010c	14	0	0	0
0x00000108	17	0	0	0
0x00000104	20	0	0	0
0x00000100	31	0	0	0

After we had tested out our Assembly code on Venus, we converted the bubble sort algorithm to machine code instructions, these instructions were put in instruction memory. We used the Risc-V green card to convert the code.

2c)

```
module Instruction_Memory
(
  input [63:0] Inst_Address,
  output reg [31:0] Instruction
);
  reg [7:0] inst_mem[160:0]; // array of 8-bit registers with 161 elements

  initial
  begin
      {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000913;//1 32 bit instruction for 4 consecutive byte addresses
      {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00500993;//2
      {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00000413;//3
      {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00050513;//4
      {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h00500113;//5
      {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h00200193;//6
      {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h00100213;//7
      {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h00700593;//8
      {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h00400613;//9
      {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h07340663;//10
      {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00000493;//11
      {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h00000513;//12
      {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'hfff98313;//13
      {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h40830333;//14
      {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h04648663;//15
      {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h00349393;//16
      {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h012383b3;//17
      {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h0003b283;//18
      {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h00148e93;//19
      {inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'h003e9e13;//20
      {inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h012e0e33;//21
      {inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h000e3f03;//22
      {inst_mem[91], inst_mem[90], inst_mem[89], inst_mem[88]} = 32'h005f4663;//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'hfc000ce3;//25
      {inst_mem[103], inst_mem[102], inst_mem[101], inst_mem[100]} = 32'h00028f93;//26
      {inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]} = 32'h000f0293;//27
      {inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]} = 32'h0053b023;//28
      {inst_mem[115], inst_mem[114], inst_mem[113], inst_mem[112]} = 32'h000f8f13;//29
      {inst_mem[119], inst_mem[118], inst_mem[117], inst_mem[116]} = 32'h01ee3023;//31
      {inst_mem[123], inst_mem[122], inst_mem[121], inst_mem[120]} = 32'h00100513;//32
      {inst_mem[127], inst_mem[126], inst_mem[125], inst_mem[124]} = 32'h00148493;//33
      {inst_mem[131], inst_mem[130], inst_mem[129], inst_mem[128]} = 32'hfa000ce3;//34
      {inst_mem[135], inst_mem[134], inst_mem[133], inst_mem[132]} = 32'h00140413;//35
      {inst_mem[139], inst_mem[138], inst_mem[137], inst_mem[136]} = 32'h00050463;//36
      {inst_mem[143], inst_mem[142], inst_mem[141], inst_mem[140]} = 32'hf8000ce3;//37
    end

  always @(Inst_Address)
  begin
    Instruction={inst_mem[Inst_Address+3],inst_mem[Inst_Address+2],inst_mem[Inst_Address+1],inst_mem[Inst_Address]};
  end
endmodule
```

The data memory is modified to adjust according to our sorting algorithm. We initialise the array and its elements in the Data Memory,

```
module Data_Memory(
  input clk,
  input MemWrite,
  input MemRead,
  input [63:0] Mem_Addr,
  input [63:0] Write_Data,
  output reg [63:0] Read_Data,
  //
output [63:0] index0,
  output [63:0] index1,
  output [63:0] index2,
  output [63:0] index3,
  output [63:0] index4
  );

  reg [7:0] DataMemory [63:0];

integer i;
initial
  begin
    for (i=0; i<64; i=i+1)
      begin
        DataMemory[i] = 0;

      end
    DataMemory[0] = 8'd14; // We initialize the array here
    DataMemory[8] = 8'd31;
    DataMemory[16] = 8'd6;
    DataMemory[24] = 8'd17;
    DataMemory[32] = 8'd20;
  end

assign index0 = {DataMemory[7],DataMemory[6],DataMemory[5],DataMemory[4],DataMemory[3],DataMemory[2],DataMemory[1],DataMemory[0]};
assign index1 = {DataMemory[15],DataMemory[14],DataMemory[13],DataMemory[12],DataMemory[11],DataMemory[10],DataMemory[9],DataMemory[8]};
assign index2 = {DataMemory[23],DataMemory[22],DataMemory[21],DataMemory[20],DataMemory[19],DataMemory[18],DataMemory[17],DataMemory[16]};
assign index3 = {DataMemory[31],DataMemory[30],DataMemory[29],DataMemory[28],DataMemory[27],DataMemory[26],DataMemory[25],DataMemory[24]};
assign index4 = {DataMemory[39],DataMemory[38],DataMemory[37],DataMemory[36],DataMemory[35],DataMemory[34],DataMemory[33],DataMemory[32]};


always @ (*)
begin
  if (MemRead)
    Read_Data = {DataMemory[Mem_Addr+7],DataMemory[Mem_Addr+6],DataMemory[Mem_Addr+5],DataMemory[Mem_Addr+4],DataMemory[Mem_Addr+3],DataMemory[Mem_Addr+2],DataMemory[Mem_Addr+1],DataMemory[Mem_Addr]};
end

always @ (posedge clk)
begin
  if (MemWrite)
  begin
    DataMemory[Mem_Addr] = Write_Data[7:0];
    DataMemory[Mem_Addr+1] = Write_Data[15:8];
    DataMemory[Mem_Addr+2] = Write_Data[23:16];
    DataMemory[Mem_Addr+3] = Write_Data[31:24];
    DataMemory[Mem_Addr+4] = Write_Data[39:32];
    DataMemory[Mem_Addr+5] = Write_Data[47:40];
    DataMemory[Mem_Addr+6] = Write_Data[55:48];
    DataMemory[Mem_Addr+7] = Write_Data[63:56];
  end
```

Here is a closer look to the initialization,

```
40    initial
41      begin
42        for (i=0; i<64; i=i+1)
43          begin
44            DataMemory[i] = 0;
45
46          end
47        DataMemory[0] = 8'd14; // We initialize the array here
48        DataMemory[8] = 8'd31;
49        DataMemory[16] = 8'd6;
50        DataMemory[24] = 8'd17;
51        DataMemory[32] = 8'd20;
52      end
```

A branch control unit is introduced to deal with branch (blt) type instructions since this unit was not present in the single cycle processor we made in lab 11. The figure below shows the Branch Unit,

```
22  module Branch_unit(
23      input [2:0] Funct3, // Specifies which type of branch condition
24      input [63:0] ReadData1,
25      input [63:0] ReadData2,
26      output reg addermuxselect
27      );
28
29      initial
30        begin
31          addermuxselect = 1'b0;
32        end
33
34      always @(*) // Branch should be taken or not is done here based on Funct3 and Read_Data1 and 2
35      begin
36          case (Funct3)
37            3'b000:
38              begin
39                if (ReadData1 == ReadData2)
40                  addermuxselect = 1'b1; // branch should be taken
41                else
42                  addermuxselect = 1'b0; // opposite
43                end
44             3'b100:
45            begin
46                      if (ReadData1 < ReadData2)
47                      addermuxselect = 1'b1;
48                    else
49                      addermuxselect = 1'b0;
50                    end
51            3'b101:
52              begin
53                if (ReadData1 > ReadData2)
54                addermuxselect = 1'b1;
55               else
56                  addermuxselect = 1'b0;
57              end
58          endcase
59        end
60  endmodule
```

The main role of the Branch Unit for our Bubble sort algorithm to work was so we can cater the sb-type instructions.

This is our ALU Control module,

```
module ALU_Control(
    input [1:0] ALUOp,   // Input: ALU Operation Type (2 bits)
    input [3:0] Funct,   // Input: Function Code (4 bits)
    output reg [3:0] Operation  // Output: ALU Operation Code (4 bits)
);

    always @ (ALUOp or Funct)
    begin
        case(ALUOp)
            2'b00: // I Type (Immediate Type)
            begin
                case(Funct[2:0])
                    4'b001: Operation = 4'b1000; // Funct[2:0] = 001 -> Operation = SLLI
                    default: Operation = 4'b0010; // Default I-Type Operation (e.g., ld, sd)
                endcase
            end

            2'b01: Operation = 4'b0110; // SB Type (Branch Type - Beq)

            2'b10: // R Type (Register Type)
            begin
                case(Funct)
                    4'b0000: Operation = 4'b0010; // Funct = 0000 -> Operation = ADD
                    4'b1000: Operation = 4'b0110; // Funct = 1000 -> Operation = SUB
                    4'b0111: Operation = 4'b0000; // Funct = 0111 -> Operation = AND
                    4'b0110: Operation = 4'b0001; // Funct = 0110 -> Operation = OR
                endcase
            end
        endcase
    end
endmodule
```

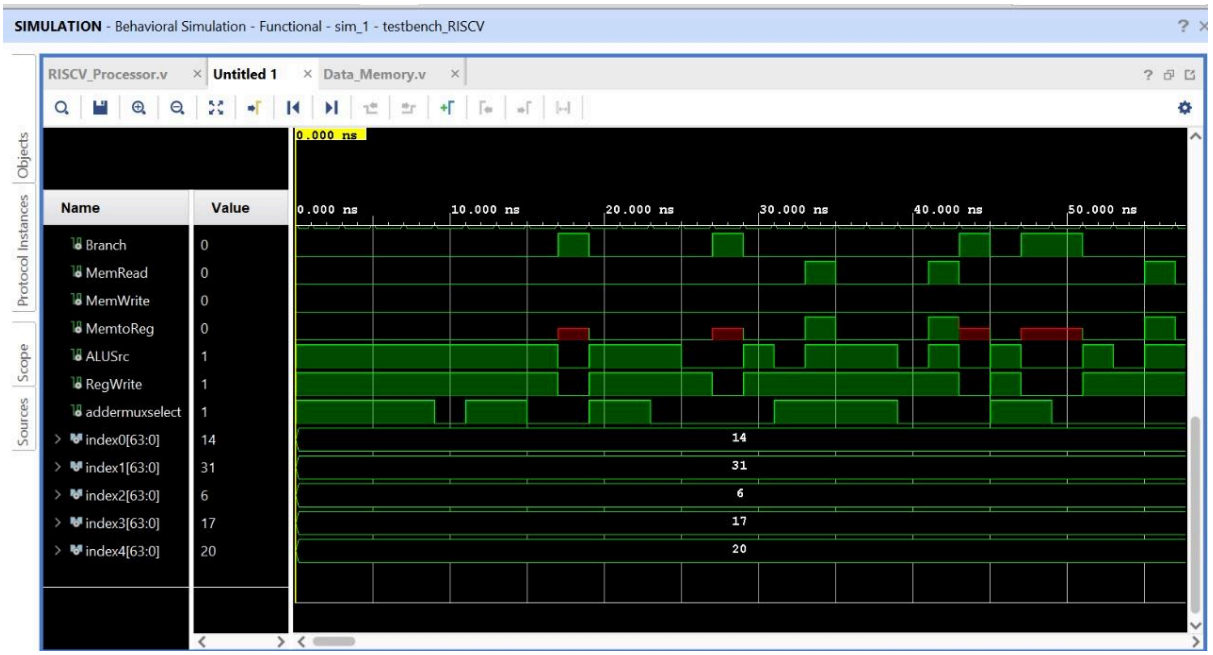
Top Module,

```
module RISCV_Processor(input clk,
    input reset,
    output reg [63:0] PC_In, PC_Out, ReadData1, ReadData2, WriteData, Result, Read_Data, imm_data,
    output reg [31:0] Instruction,
    output reg [6:0] opcode,
    output reg [4:0] rs1, rs2, rd,
    output reg [1:0] ALUOp,
    output reg [63:0] adder_out1, adder_out2,
    output reg Branch, MemRead, MemWrite, MemtoReg, ALUSrc, RegWrite, addermuxselect,
    output reg [63:0] index0, index1, index2, index3, index4
    );

wire [63:0] PC_In, PC_Out, adder_out1, adder_out2, imm_data, WriteData, ReadData1, ReadData2, Result, Read_Data;
wire [63:0] muxmid_out;
wire [31:0] Instruction;
wire [6:0] opcode, funct7;
wire [4:0] rd, rs1, rs2;
wire [3:0] Funct, Operation;
wire [2:0] funct3;
wire [1:0] ALUOp;
wire Branch, MemRead, MemWrite, MemtoReg, ALUSrc, RegWrite, Zero, addermuxselect, branch_sel;
wire [63:0] index0, index1, index2, index3, index4;

// ---- Instruction Fetch ----- //
Adder A1(.A(PC_Out), .B(64'd4), .Out(adder_out1)); // adder a1 adds current pc counter with a constant offset to get next pc
Mux_2x1 muxfirst(.A(adder_out1), .B(adder_out2), .S(Branch && addermuxselect), .Out(PC_In)); // mux selects between the added counter or the branch target address based on branch control signal
Program_Counter PC(.clk(clk), .reset(reset), .PC_In(PC_In), .PC_Out(PC_Out)); // updating the program counter to next pc
Instruction_Memory IM(.Inst_Address(PC_Out), .Instruction(Instruction)); // Reads the instruction from the instruction memory

// Instruction Decode / Register File Read //
Instruction_Parser IP(.Instruction(Instruction), .Opcode(opcode), .RD(rd), .Funct3(funct3), .RS1(rs1), .RS2(rs2), .Funct7(funct7)); // Decodes the fetched instruction into its components
Imm_Gen Immgen(.Instruction(Instruction), .Imm(imm_data)); // Generates the immediate value based on the instruction type
Control_Unit cu(.Opcode(opcode), .Branch(Branch), .MemRead(MemRead), .MemtoReg(MemtoReg), .ALUOp(ALUOp), .MemWrite(MemWrite), .ALUSrc(ALUSrc), .RegWrite(RegWrite)); // Decodes instruction to generate control signals
RegisterFile rf(.clk(clk), .reset(reset), .WriteData(WriteData), .RS1(rs1), .RS2(rs2), .RD(rd), .RegWrite(RegWrite), .ReadData1(ReadData1), .ReadData2(ReadData2)); // Reads operand values from register file
assign Funct = {Instruction[30], Instruction[14:12]}; //                                                                                                  // based on the decoded register identifiers

// ----- Execute / Address Calculation ------ //
Adder A2(.A(PC_Out), .B(imm_data * 2), .Out(adder_out2)); // adds current pc with branch target-offset to calculate branch target address
Mux_2x1 muxmid(.A(ReadData2), .B(imm_data), .S(ALUSrc), .Out(muxmid_out)); //selects between read-data-2 or imm_data value based on ALU src signal
ALU_Control aluc(.ALUOp(ALUOp), .Funct(Funct), .Operation(Operation)); // determines specific alu operation based on instruction type
ALU64bit ALU(.A(ReadData1), .B(muxmid_out), .ALUOp(Operation), .Result(Result)); // performs arithmetic/ logical operation based on ALUop using readdata1 and muxmid_out as operands
Branch_unit BU(.Funct3(funct3), .ReadData1(ReadData1), .ReadData2(ReadData2), .addermuxselect(addermuxselect)); // determines branch outcome based on comparision of readdata1 and readdata2

// ------ MEM: Memory Access  ------ //
//Data_Memory DM(.Mem_Addr(Result), .Write_Data(ReadData2), .clk(clk), .MemWrite(MemWrite), .MemRead(MemRead), .Read_Data(Read_Data)); // does mem read or mem write based on alu result
Data_Memory DM(.clk(clk), .MemWrite(MemWrite), .MemRead(MemRead), .Mem_Addr(Result), .Write_Data(ReadData2), .Read_Data(Read_Data), .index0(index0), .index1(index1), .index2(index2), .index3(index3), .index4(index4));

// ----- Write Back ----- //
Mux_2x1 muxlast(.A(Result), .B(Read_Data), .S(MemtoReg), .Out(WriteData)); // selects between alu/mem read data based on memtoreg signal
endmodule
```

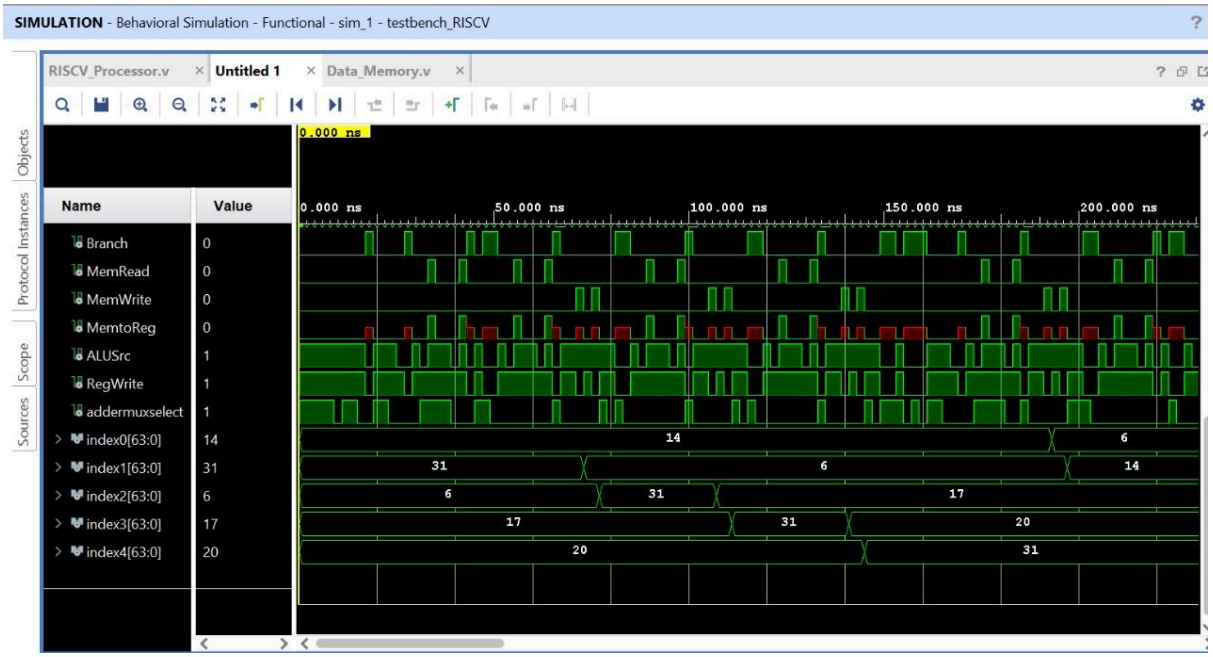
2d)

Below are the results of our Bubble Sort algorithm,

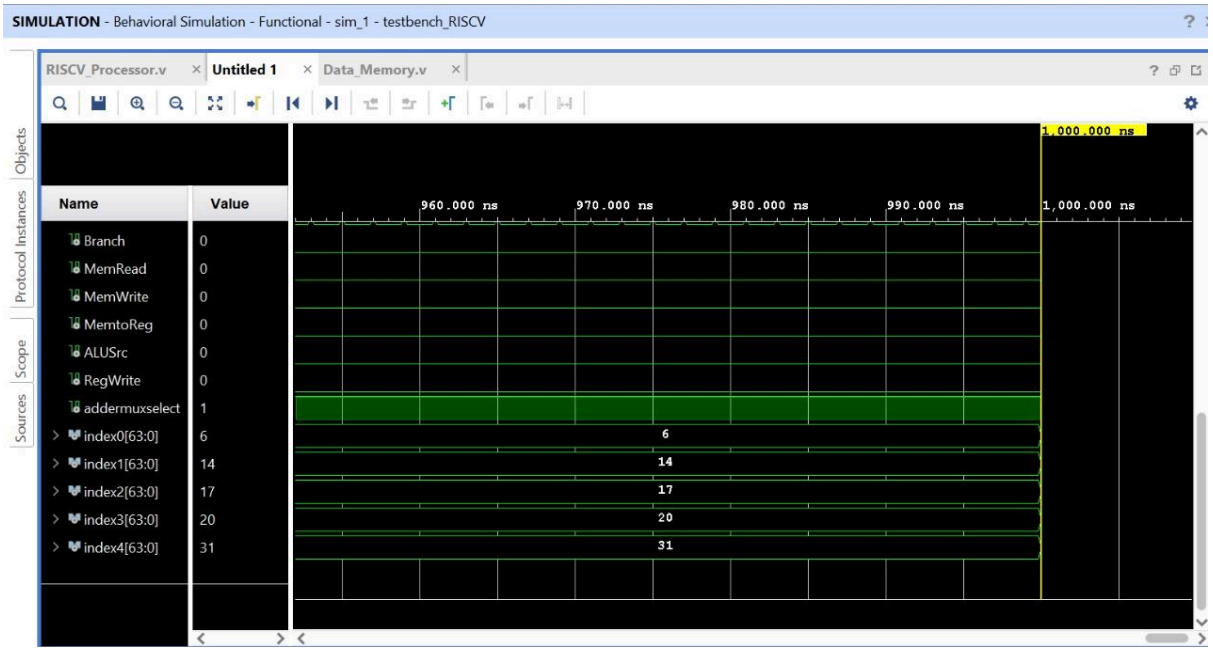
Before Sorting



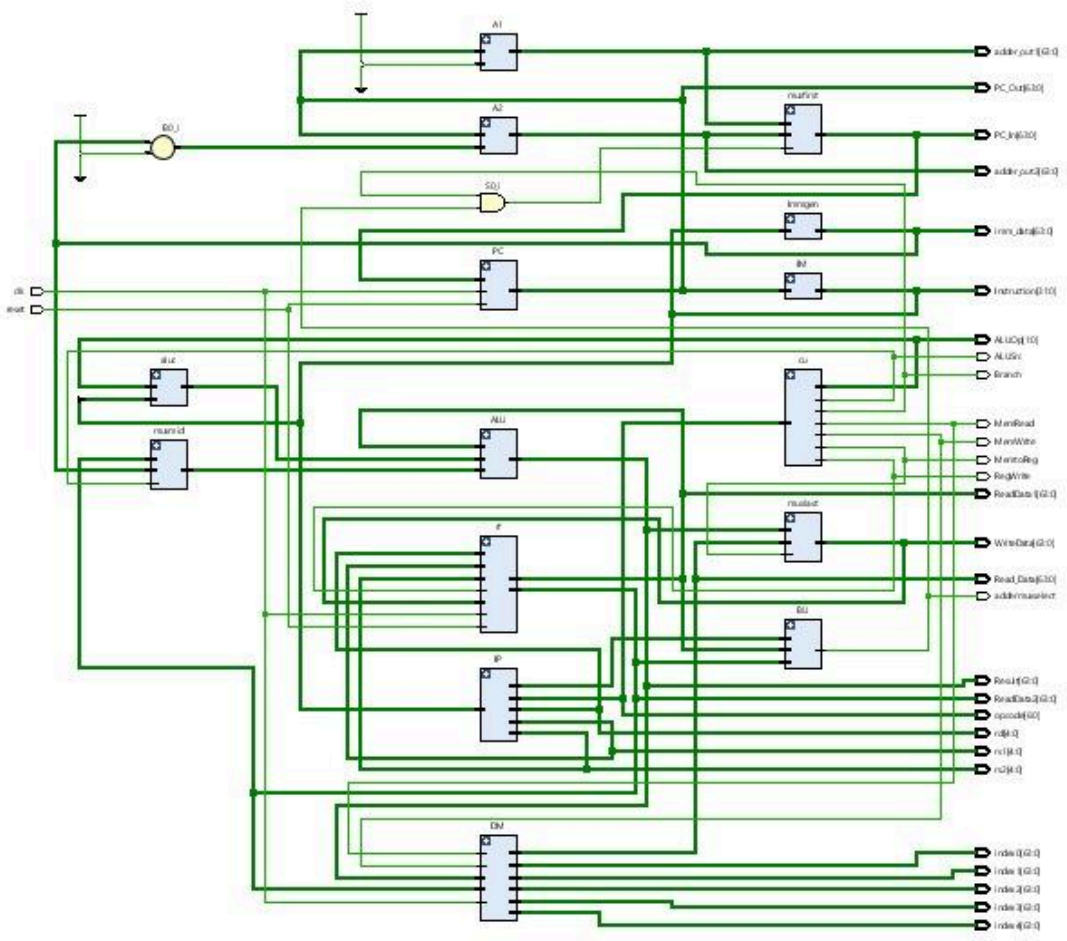
In between sorting,



After sorting,



Schematic,



2. Task 2:

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.

2a)

We also added the forwarding unit and 3-to-1 multiplexer to make the unit work. It selects the forwardA and forwardB output depending on if the current instruction is dependent on the previous instruction. We use the following table to assign the forwardA and forwardB values.:

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.

The Forwarding Module is here:

```
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
    // Forwarding logic for operand A
    if (EXMEM_rd == IDEX_rs1 && EXMEM_RegWrite && EXMEM_rd != 0) begin
        fwd_A = 2'b10;  // Forward value from the EX/MEM pipeline stage
    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;  // Forward value from the MEM/WB pipeline stage
    end else begin
        fwd_A = 2'b00;  // No forwarding for operand A
    end

    // Forwarding logic for operand B
    if ((EXMEM_rd == IDEX_rs2) && EXMEM_RegWrite && EXMEM_rd != 0) begin
        fwd_B = 2'b10;  // Forward value from the EX/MEM pipeline stage
    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;  // Forward value from the MEM/WB pipeline stage
    end else begin
        fwd_B = 2'b00;  // No forwarding for operand B
    end
end

endmodule
```

In order to select the forward values, two 3-1 mux are introduced into the hardware. The design module for this mux is:

```
module mux3x1(
    input [63:0] a, b, c,
    input [1:0] sel,
    output reg [63:0] data_out
);

always @(*) begin
    if (sel == 2'b00) begin    // If sel is 00, select input A
        data_out = a;
    end
    else if (sel == 2'b01) begin    // If sel is 01, select input B
        data_out = b;
    end
    else if (sel == 2'b10) begin    // If sel is 10, select input C
        data_out = c;
    end
    else begin    // For all other cases, output X (undefined)
        data_out = 2'bX;
    end
end


endmodule

```

2b) IF/ID Module:

```
module IF_ID(
    input clk, IFID_Write, Flush,
    input [63:0] PC_addr,
    input [31:0] Instruc,
    output reg [63:0] PC_store,
    output reg [31:0] Instr_store
);

always @(posedge clk) begin
    // Check if Flush signal is active
    if (Flush) begin
        // Flush active: Reset stored values
        PC_store <= 0;
        Instr_store <= 0;
    end else if (!IFID_Write) begin
        // IFID_Write inactive: Preserve stored values
        PC_store <= PC_store;
        Instr_store <= Instr_store;
    end else begin
        // Store new values in IF/ID pipeline registers
        PC_store <= PC_addr;
        Instr_store <= Instruc;
    end
end

endmodule
```

ID/EX Module:

```
module ID_EX(
    input        clk,                          // Clock signal
    input        Flush,                        // Flush control signal
    input [63:0] program_counter_addr,    // Program counter address input
    input [63:0] read_data1,              // Data 1 input
    input [63:0] read_data2,              // Data 2 input
    input [63:0] immediate_value,         // Immediate value input
    input [3:0]  function_code,           // Function code input
    input [4:0]  destination_reg,         // Destination register input
    input [4:0]  source_reg1,             // Source register 1 input
    input [4:0]  source_reg2,             // Source register 2 input
    input        MemtoReg,                      // Memory-to-register control signal
    input        RegWrite,                      // Register write control signal
    input        Branch,                         // Branch control signal
    input        MemWrite,                       // Memory write control signal
    input        MemRead,                        // Memory read control signal
    input        ALUSrc,                         // ALU source control signal
    input [1:0]  ALU_op,                   // ALU operation control signal

    output reg [63:0] program_counter_addr_out,    // Output: Stored program counter address
    output reg [63:0] read_data1_out,              // Output: Stored Data 1
    output reg [63:0] read_data2_out,              // Output: Stored Data 2
    output reg [63:0] immediate_value_out,         // Output: Stored Immediate value
    output reg [3:0] function_code_out,            // Output: Stored Function code
    output reg [4:0] destination_reg_out,          // Output: Stored Destination register
    output reg [4:0] source_reg1_out,                    // Output: Stored Source register 1
    output reg [4:0] source_reg2_out,                    // Output: Stored Source register 2
    output reg       MemtoReg_out,                       // Output: Stored Memory-to-register control
    output reg       RegWrite_out,                           // Output: Stored Register write control
    output reg Branch_out,                              // Output: Stored Branch control
    output reg MemWrite_out,                           // Output: Stored Memory write control
    output reg MemRead_out,                         // Output: Stored Memory read control
    output reg ALUSrc_out,                               // Output: Stored ALU source control
    output reg [1:0] ALU_op_out                    // Output: Stored ALU operation control
```

```
    else
    begin
        // Pass input values to output registers
    program_counter_addr_out = program_counter_addr;
    read_data1_out = read_data1;
    read_data2_out = read_data2;
    immediate_value_out = immediate_value;
    function_code_out = function_code;
    destination_reg_out = destination_reg;
    source_reg1_out = source_reg1;
    source_reg2_out = source_reg2;
    RegWrite_out = RegWrite;
    MemtoReg_out = MemtoReg;
    Branch_out = Branch;
    MemWrite_out = MemWrite;
    MemRead_out = MemRead;
    ALUSrc_out = ALUSrc;
    ALU_op_out = ALU_op;
    end
end

endmodule

     );

     always @(posedge clk) begin
         if (Flush)
         begin
         // Reset all output registers to 0
         program_counter_addr_out = 0;
         read_data1_out = 0;
         read_data2_out = 0;
         immediate_value_out = 0;
         function_code_out = 0;
         destination_reg_out = 0;
         source_reg1_out = 0;
         source_reg2_out = 0;
         MemtoReg_out = 0;
         RegWrite_out = 0;
         Branch_out = 0;
         MemWrite_out = 0;
         MemRead_out = 0;
         ALUSrc_out = 0;
         ALU_op_out = 0;
         end
```

EX/MEM Module:

```
module EX_MEM(
    input clk,                    // Clock
    input Flush,                  // Flush control
    input RegWrite,
    input MemtoReg,               // Control signal for selecting memory or ALU result for register write
    input Branch,                 // Branch Control SIgnal
    input Zero,                   // Control singal indicating the ALU result is zero
    input MemWrite,
    input MemRead,
    input is_greater,             // Control signal indicating the comparison result of the ALU operation
    input [63:0] immvalue_added_pc, // Immediate value added to the program counter
    input [63:0] ALU_result,      // Result of the ALU operation
    input [63:0] WriteData,       // Data to be written to memory or register file
    input [3:0] function_code,    // Function code for ALU operation
    input [4:0] destination_reg,  // Destination register for register write

    output reg RegWrite_out,      // register write
    output reg MemtoReg_out,      // MEM or ALU result for register write
    output reg Branch_out,        // branch signal
    output reg Zero_out,          // signal for ALU result is zero
    output reg MemWrite_out,      // MEM write
    output reg MemRead_out,       // MEM read
    output reg is_greater_out,    // when a greater than check for comparison
    output reg [63:0] immvalue_added_pc_out, // immediate value added to the program counter
    output reg [63:0] ALU_result_out,      // ALU result
    output reg [63:0] WriteData_out,       // Write data
    output reg [3:0] ALU_OP,    // ALU operation code
    output reg [4:0] dest_reg_out    // destination register for reg write operation
);
```

MEM/WB Module:

```
module MEM_WB(
    input clk,                      // Clock signal
    input RegWrite,                 // Control signal for enabling register write
    input MemtoReg,                 // Control signal for selecting memory or ALU result for register write
    input [63:0] ReadData,          // Data read from memory or register file
    input [63:0] ALU_result,        // Result of the ALU operation
    input [4:0] destination_reg,    // Destination register for register write

    output reg RegWrite_out,        // Output signal for enabling register write
    output reg MemtoReg_out,        // Output signal for selecting memory or ALU result for register write
    output reg [63:0] ReadData_out, // Output signal for data read from memory or register file
    output reg [63:0] ALU_result_out, // Output signal for the ALU result
    output reg [4:0] destination_reg_out // Output signal for destination register for register write
);

    // Assign output values based on input signals
    always @(posedge clk) begin
        RegWrite_out = RegWrite;
        MemtoReg_out = MemtoReg;
        ReadData_out = ReadData;
        ALU_result_out = ALU_result;
        destination_reg_out = destination_reg;
    end

endmodule
```

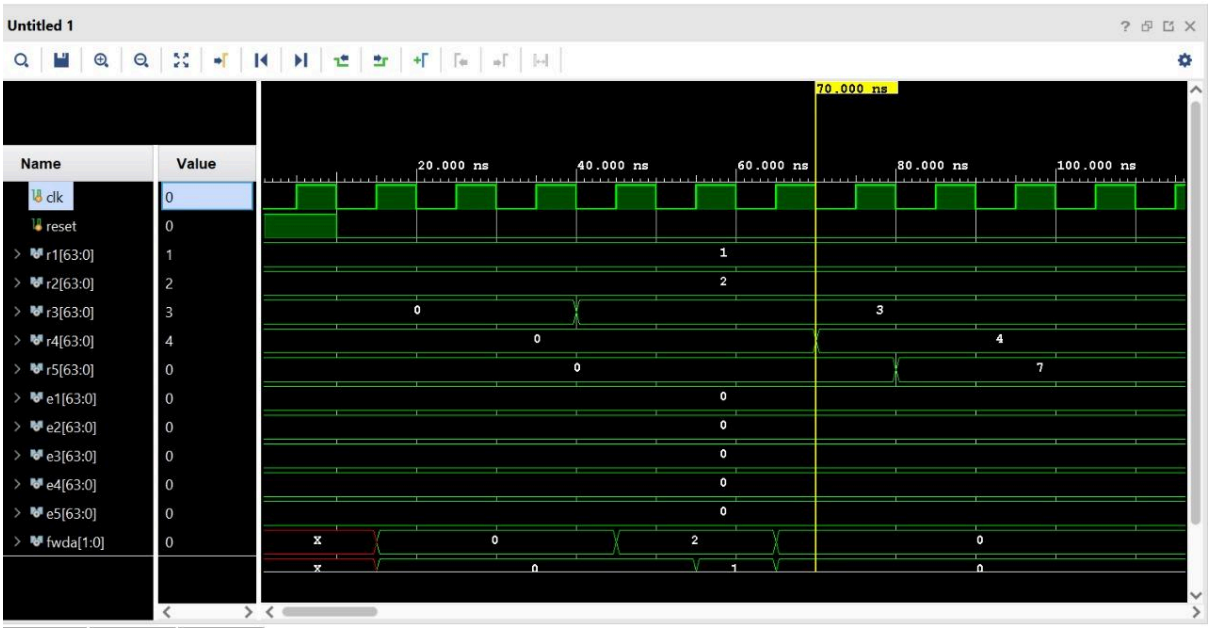
After making these registers and forwarding module, we executed 3 add instructions that are dependent on each other to show the result of forwarding. Our test cases were initialised in the instruction memory:

```
initial begin
inst_mem[0] = 8'b00110011; //add x10,x12,x13
inst_mem[1] = 8'b00000101;
inst_mem[2] = 8'b11010110;
inst_mem[3] = 8'b0;

inst_mem[4] = 8'b00110011; //add x8,x10,x12
inst_mem[5] = 8'b00000100;
inst_mem[6] = 8'b11000101;
inst_mem[7] = 8'b0;

inst_mem[8] = 8'b10110011; //add x1,x8,x10
inst_mem[9] = 8'b00000000;
inst_mem[10] = 8'b10100100;
inst_mem[11] = 8'b00000000;
```

2c) We initialised the x12 and x13 registers in the registerFile module with the values 1 and 2:

```
initial
begin
    for (i = 0; i < 32; i = i + 1)
            Registers[i] = 64'd0;
    Registers[12] = 64'd1;
    Registers[13] = 64'd2;
end


always @(negedge clk ) begin // doing regwrite on the negative clock edge

    if (RegWrite) begin
        Registers[RD] = WriteData;
    end
end

always@(*) begin
    ReadData1 = reset ? 0 : Registers[RS1];
    ReadData2 = reset ? 0 : Registers[RS2];
end

  assign r1 = Registers[12];
  assign r2 = Registers[13];
  assign r3 = Registers[10];
  assign r4 = Registers[8];
  assign r5 = Registers[1];
```

Simulation Output:



3. Task 3:

4a) For Task 3 we were required to implement the Bubble Sorting algorithm of Task 1 into the pipelined processor developed in Task 2. In order to do this, we had to include the 'Data Hazard Detection Unit' into our project.

According to the different sets of given instructions, the Data Hazard Detection module stalls the pipeline by setting the control signals of the multiplexers and relevant hardware to zero. This ensures that another instruction is not accepted into the pipeline for 1 clock cycle.

Following is the code snippet of our Hazard Detection module:

```
module Hazard_Detection
(
    input [4:0] current_rd, previous_rs1, previous_rs2,
    input current_MemRead,
    output reg mux_out,
    output reg enable_Write, enable_PCWrite
);

always @(*) begin
    // Hazard detection logic
    if (current_MemRead && (current_rd == previous_rs1 || current_rd == previous_rs2)) begin
        // Hazard detected: control signals set accordingly
        mux_out = 0;             //multiplexer output is set to zero
        enable_Write = 0;       // nothing is written into the next pipeline stage
        enable_PCWrite = 0;    // the program counter is not increased so next instruction is not fetched
    end else begin
        // No hazard detected: control signals set accordingly
        mux_out = 1;             // the multiplexer output proceeds as normal
        enable_Write = 1;       // next pipeline stage mux is enabled
        enable_PCWrite = 1;    // program counter mux is enabled.
    end
end

endmodule
```

Once again for the bubble sorting, we initialise the values in the data memory:

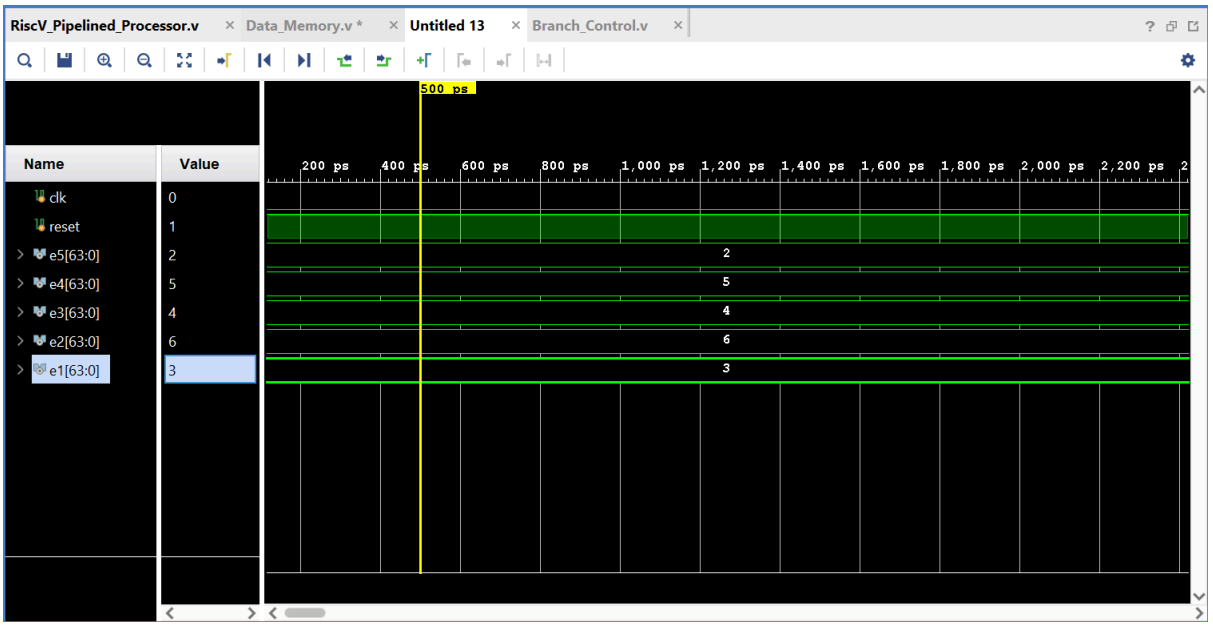
```
initial
    begin
    for (i = 0; i < 512; i = i + 1)
    begin
        DataMemory[256] = 8'd2;
        DataMemory[264]=  8'd5;
        DataMemory[272]= 8'd4;
        DataMemory[280] = 8'd6;
        DataMemory[288]= 8'd3;
    end
end
```

We also made modifications to the Branch module in order to support the flushing of pipeline based on the control signals:

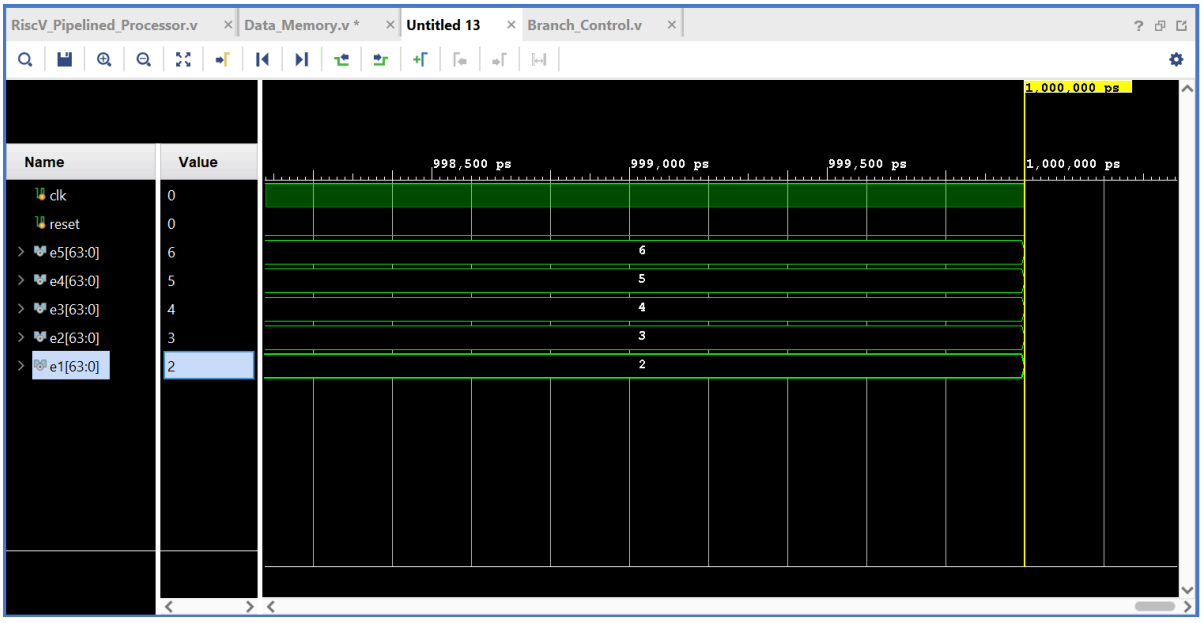
```
module Branch_Control
(
input Branch, Zero, Is_Greater_Than,
input [3:0] funct,
output reg switch, Flush
);


always @(*) begin
// Check if Branch signal is 1 then we swithc branch and flush else not
if (Branch) begin

    // Use a case statement based on funct[2:0] value
    case ({funct[2:0]})

        // Case when funct[2:0] is 3'b000
        3'b000: begin
            // Check if Zero signal is active
            if (Zero)
                switch = 1;  // Set switch_branch to 1
            else
                switch = 0;  // Set switch_branch to 0
        end

        // Case when funct[2:0] is 3'b001
        3'b001: begin
            // Check if Zero signal is active
            if (Zero)
                switch = 0;  // Set switch_branch to 0
            else
                switch = 1;  // Set switch_branch to 1
        end

        // Case when funct[2:0] is 3'b101
        3'b101: begin
            // Check if Is_Greater_Than signal is active
            if (Is_Greater_Than)
                switch = 1;  // Set switch_branch to 1
            else
                switch = 0;  // Set switch_branch to 0
        end

        // Case when funct[2:0] is 3'b100
        3'b100: begin
            // Check if Is_Greater_Than signal is active
            if (Is_Greater_Than)
                switch = 0;  // Set switch_branch to 0
            else
                switch = 1;  // Set switch_branch to 1
        end

        // Default case
        default: switch = 0;  // Set switch_branch to 0

    endcase

end

else
    switch = 0;  // Set switch_branch to 0 if Branch signal is inactive
end

always @(switch) begin
    // Based on the switch_branch value
    if (switch)
        Flush = 1;  // switch_branch is 1
    else
        Flush = 0;  // switch_branch is 0

end

endmodule
```

After making these modifications and using the rest of the modules developed in task 1 and 2, we were able to achieve the sorting on the pipelined processor.

4b) Before sorting:



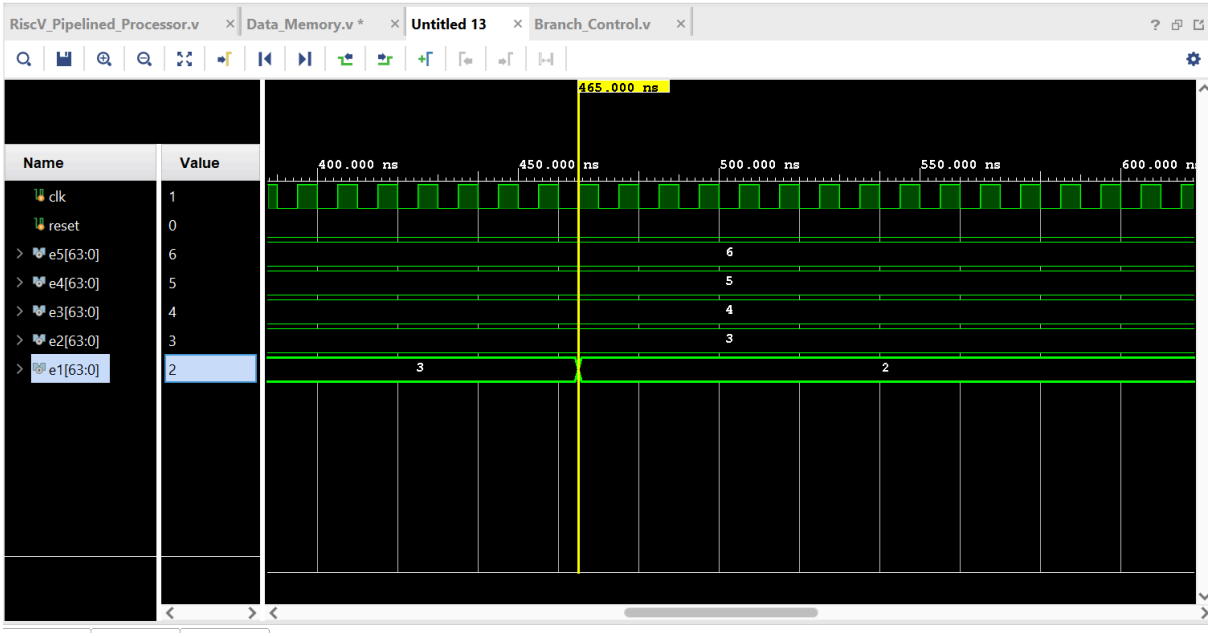
After sorting:



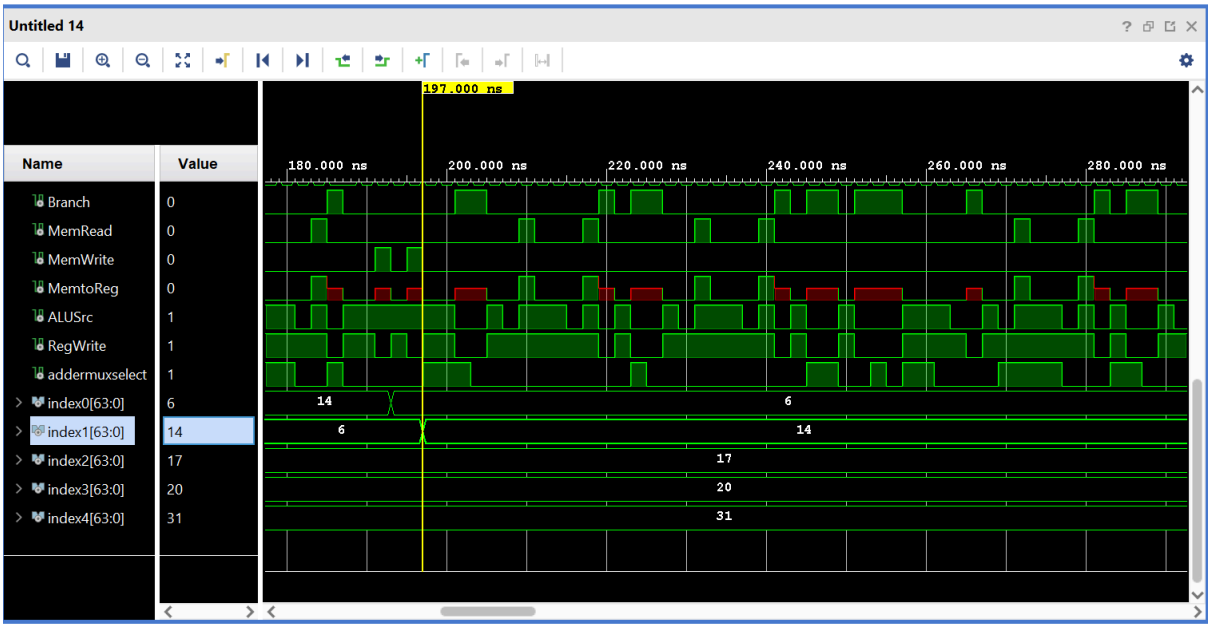
4. Task 4:

In this task we are required to analyse the performance of the pipelined and single cycle processor.

The pipeline processor takes 465ns as seen in the snippet below:



The single cycle processor takes 197ns as seen in the snippet below:



We can calculate the speed up for our test case to be:

Execution Time (pipelined) / Execution time (single cycle) = 465ns/197ns = 2.36.

Therefore we can deduce the fact that the single cycle processor is 2.36 times faster in sorting a 5-element array compared to a pipelined processor. This is the case because in the pipelined processor there is a hazard detection unit that inserts stalls into the pipelined processor. These stalls/flushes increase the execution time significantly.

5. Challenges

- In Task 2, making the modules for the pipeline registers was a challenging task. We had to carefully identify the inputs and outputs of each register, give appropriate names to the wires and regs, and then make them work.
- Integrating each register into the top module was also a complex task and took us a lot of time.
- In Task 3 the logic of the hazard detection module was straightforward to implement, but we struggled to integrate it into the top level module.

6. Task Division

We collaborated in making the project code and did all the tasks together in the lab and outside university as well. For the report we divided each of the three main tasks between us. Jibran did task 1, Daniyal did task 2, and Ansab compiled task 3.

7. Conclusion

We were able to implement the bubble sort algorithm on both single cycle and pipelined processor to obtain the desired results. As expected, the single cycle processor was faster in executing the algorithm as it does not include stalls like the pipelined processor.

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

Github Repository: https://github.com/chaudhary8503/CA-Project