Habib University

Dhanani School of Science and Engineering

CE/CS 321/330 Computer Architecture

# Final Lab Project

5-Stage Pipelined Processor To Execute A Single Array Sorting Algorithm

**Group Members**

Misha jessani (mj08757)

Syed Muhammad Faraz Hussain (sh09030)

Ahla Haider (ah08821)

April 21, 2025

# Contents

## 1 Introduction

The objective of this project is to design and implement a pipelined processor based on the RISC-V instruction set architecture. The task involved modifying a single-cycle processor to a five-stage pipelined processor capable of executing a sorting algorithm. The processor operates on an array of at least seven integers, stored at memory address 0x100. The primary aim was to enhance the processor's performance by implementing pipeline stages, addressing hazards, and conducting performance comparisons between the single-cycle and pipelined processors. The project provided insight into pipeline design, hazard detection, and performance optimization in processor architecture, focusing on sorting algorithms and their execution time on different processor configurations.

## 2 Sorting Algorithm on a Single Cycle Processor

### 2.1 Bubble Sort Assembly Code

Listing 1: Bubble Sort Assembly Code

```
addi x11, x0, 7          # Initial value to insert into array
addi x29, x0, 7          # Size of array = 7
addi x30, x0, 0          # Offset for memory (for storing array)
addi x31, x0, 0          # i = 0, index counter
addi x28, x0, 7          # Comparison value for size checking

# Initialize array with descending values: 7, 6, 5, ..., 1
Array:
sw x11, 0x100(x30)       # Store current value into array
addi x31, x31, 1         # i++
addi x30, x30, 8         # offset += 8 bytes (64-bit aligned)
addi x11, x11, -1        # next value = value - 1
beq x28, x31, filled     # if i == 7, go to filled
beq x0, x0, Array        # unconditional jump

filled:
addi x11, x0, 7          # Reset x11 = array size = 7
addi x30, x0, 0          # i = 0 (outer loop)

I_Loop:
beq x11, x30, Exit       # if i == size, done
sub x20, x11, x30        # x20 = n - i
addi x20, x20, -1        # x20 = n - i - 1
addi x31, x0, 0          # j = 0
addi x23, x0, 0          # offset = 0
addi x30, x30, 1         # i++

J_Loop:
beq x31, x20, I_Loop     # if j == n - i - 1, outer loop++
addi x24, x23, 8         # offset for j+1
lw x15, 0x100(x23)       # A[j]
lw x16, 0x100(x24)       # A[j+1]

# Use slt (set less than) to simulate blt
slt x25, x16, x15        # if A[j+1] < A[j], x25 = 1
beq x25, x0, no_swap     # if not less, skip swap

SWAP:
sw x16, 0x100(x23)       # A[j] = A[j+1]
sw x15, 0x100(x24)       # A[j+1] = A[j]

no_swap:
addi x31, x31, 1         # j++
addi x23, x23, 8         # offset += 8
beq x0, x0, J_Loop       # unconditional jump

Exit:
addi x0, x0, 0           # nop
```

## 2.2 Bubble Sort Python Code

Listing 2: Bubble Sort Python Code (taken from Programmiz.com)

```
def bubbleSort(array):

  # loop to access each array element
  for i in range(len(array)):

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):

      # compare two adjacent elements
      # change > to < to sort in descending order
      if array[j] > array[j + 1]:

        # swapping elements if elements
        # are not in the intended order
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp
```

## 3 Task 1 - Changes to Single Cycle Processor

In task 1, we employed the codes made in our Lab 11 and modified them to carry out the BubbleSort algorithm. This algorithm included making an array of size 7 in descending order first, i.e., the worst case, and then sorting it to the ascending order by executing one instruction in one cycle.

### 3.1 Instruction Memory

Listing 3: Changes made to Instruction Memory

```
module Instruction_Memory
(
        input [63:0] Inst_Address,
        output reg [31:0] Instruction
);
        reg [7:0] inst_mem [124:0];
        integer i;

        initial
        begin

```

```
{inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]}     = 32'h00700593;
{inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]}     = 32'h00700e93;
{inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]}   = 32'h00000f13;
{inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00000f93;
{inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h00700e13;
{inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h10bf2023;
{inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h001f8f93;
{inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h008f0f13;
{inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'hfff58593;
{inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h01fe0463;
{inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'hfe0006e3;
{inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h00700593;
{inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h00000f13;
{inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h05e58263;
{inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h41e58a33;
{inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'hfffa0a13;
{inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h00000f93;
{inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h00000b93;
{inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h001f0f13;
{inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'hff4f84e3;
{inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h008b8c13;
{inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h100ba783;
{inst_mem[91], inst_mem[90], inst_mem[89], inst_mem[88]} = 32'h100c2803;
{inst_mem[95], inst_mem[94], inst_mem[93], inst_mem[92]} = 32'h00f82cb3;
{inst_mem[99], inst_mem[98], inst_mem[97], inst_mem[96]} = 32'h000c8663;
{inst_mem[103], inst_mem[102], inst_mem[101], inst_mem[100]} = 32'
    h110ba023;
{inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]} = 32'
    h10fc2023;
{inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]} = 32'
    h001f8f93;
{inst_mem[115], inst_mem[114], inst_mem[113], inst_mem[112]} = 32'
    h008b8b93;
{inst_mem[119], inst_mem[118], inst_mem[117], inst_mem[116]} = 32'
    hfc000ce3;
{inst_mem[123], inst_mem[122], inst_mem[121], inst_mem[120]} = 32'
    h00000013;
        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
    end
endmodule
```

## 3.2 Data Memory

Listing 4: Changes made to Data Memory

```
`timescale 1ns/1ps
module Data_Memory
(
        input [63:0] Mem_Addr,
        input [63:0] Write_Data,
        input clk, MemWrite, MemRead,
        output reg [63:0] Read_Data,
        output reg [7:0] element1,
        output reg [7:0] element2,
        output reg [7:0] element3,
        output reg [7:0] element4,
        output reg [7:0] element5,
        output reg [7:0] element6,
        output reg [7:0] element7
);
        reg [7:0] DataMemory [312:0];
    integer i;
        initial
    begin
      for (i = 0; i < 300; i = i + 1)
      begin
        DataMemory[i] = 0;
      end
      DataMemory[256]  = 8'd33;
      DataMemory[264]  = 8'd14;
      DataMemory[272] = 8'd20;
      DataMemory[280] = 8'd15;
      DataMemory[288] = 8'd36;
    end

        always @ (*)
        begin
                if (MemRead)
                begin
                        assign 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

           assign element1 = DataMemory[256];
           assign element2 = DataMemory[264];
           assign element3 = DataMemory[272];
           assign element4 = DataMemory[280];
           assign element5 = DataMemory[288];
           assign element6 = DataMemory[296];
           assign element7 = DataMemory[304];

        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
        end
endmodule
```

## 3.3 Changes to Control Unit

We made changes in the Control Unit module to incorporate branch instructions. We specified the values of control signals based on their opcode. Given that both instructions require jumping to a specific memory address without any reading or writing, the opcode for beq and blt is the same, as are their signals.

Listing 5: Changes made to Control Unit

```
module Control_Unit(
    input [6:0] Opcode,
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg MemWrite,
    output reg AluSrc,
    output reg RegWrite,
    output reg [1:0] AluOp
    );

    always@(Opcode)
    begin
    case(Opcode)
    7'b0110011: begin
    assign AluSrc = 1'b0;
    assign MemtoReg = 1'b0;
    assign RegWrite = 1'b1;
    assign MemRead  = 1'b0;
    assign MemWrite = 1'b0;
    assign Branch = 1'b0;
    assign AluOp = 2'b10;
    end

    7'b0000011: begin
    assign AluSrc = 1'b1;
     assign MemtoReg = 1'b1;
    assign  RegWrite = 1'b1;
     assign MemRead  = 1'b1;
    assign  MemWrite = 1'b0;
    assign  Branch = 1'b0;
    assign  AluOp = 2'b00;
    end

    7'b0100011: begin
    assign AluSrc = 1'b1;
    assign MemtoReg = 1'bx;
    assign RegWrite = 1'b0;
    assign MemRead  = 1'b0;
    assign MemWrite = 1'b1;
    assign Branch = 1'b0;
    assign AluOp = 2'b00;
    end

    7'b1100011: begin
    assign AluSrc = 1'b0;
    assign MemtoReg = 1'bx;
    assign RegWrite = 1'b0;
    assign MemRead  = 1'b0;
    assign MemWrite = 1'b0;
    assign Branch = 1'b1;
    assign AluOp = 2'b01;
    end


    7'b0010011: begin
    assign AluSrc = 1'b1;
    assign MemtoReg = 1'b0;
    assign RegWrite = 1'b1;
    assign MemRead  = 1'b0;
    assign MemWrite = 1'b0;
    assign Branch = 1'b0;
    assign AluOp = 2'b00;
    end

    default: begin
    assign AluSrc = 1'b0;
    assign MemtoReg = 1'b0;
    assign RegWrite = 1'b0;
    assign MemRead  = 1'b0;
    assign MemWrite = 1'b0;
    assign Branch = 1'b0;
    assign AluOp = 2'b00;
    end

    endcase

    end
endmodule
```

## 3.4 Changes to ALU Control Unit

ALU Control, which creates the 4-bit ALU Control input, had to be modified to incorporate branch type operations. The 4-bit Func Field and a 2-bit ALUOp are inputs to the control unit. The output is a 4-bit signal that, depending on Func and the ALUOp field, selects one of the six operations to be executed in our example, directly controlling the ALU. According to ALUOp, the operation that has to be carried out will either be add (00) for loads and stores or will be determined by the operation that is encoded in the funct7 and funct3 fields (10, 01). When ALUOp was "01," that is, when there was a branch type instruction, we added an additional case structure.

Listing 6: Changes made to ALU control unit

```
module ALU_Control
(
        input [1:0] ALUOp,
        input [3:0] Funct,
        output reg [3:0] Operation
);
        always @ (ALUOp or Funct)
        begin
                case(ALUOp)
                        2'b00: Operation <= 4'b0010;
                        2'b01:
                        begin
                        if (Funct[2:0] == 3'b100)
                             Operation <= 4'b1111;
                        else
                             Operation <= 4'b0110;
                        end
```

```
                            2'b10:
                            begin
                            case(Funct)
                                    4'b0000: Operation <= 4'b0010;
                                    4'b1000: Operation <= 4'b0110;
                                    4'b0111: Operation <= 4'b0000;
                                    4'b0110: Operation <= 4'b0001;
                                    default: begin
                                    Operation <= 4'b0000;
                                    end
                            endcase
                            end
                    endcase
            end
endmodule
```

## 3.5 Changes to ALU

We modified our ALU to execute correct branch results. If the first value is less than the second value, the Result is set to '0'. Similar to the 'beq' instruction, '0' would be assigned to Zero if Result == 0. This eliminates the need for extra hardware modifications to check for additional branch type instructions. We implemented this according to the RISC-V processor, where the result from the ALU is ANDed with the branch signal. When the selection line of the mux is Branch & Zero, the PC is unconditionally replaced with PC + 4 when the Branch control signal is 0, and the branch target is replaced with the PC if the Zero output of the ALU is high.

Listing 7: Changes made to ALU 64

```
module ALU_64(
    output reg Zero,
    input [63:0] a,
    input [63:0] b,
    input [3:0] ALUOp,
    output reg [63:0] Result,
    );
    always @(*)
    begin
    case (ALUOp)
    4'b0000: Result = a&b;
    4'b0001: Result = a|b;
    4'b0010: Result = a+b;
    4'b0110: Result = a-b;
    4'b1100: Result = ~(a|b);
    4'b1111: Result = a<b ? 0:1;
    default: Result = 0;

    endcase

    if (Result ==0)
    Zero = 1;
    else
    Zero = 0;
    poss <= ~Result[63];
    end
endmodule
```

## 3.6 Result for Single Cycle Pipeline

Firstly, we build an array in the data memory by initializing 7 values in jumbled order. Final

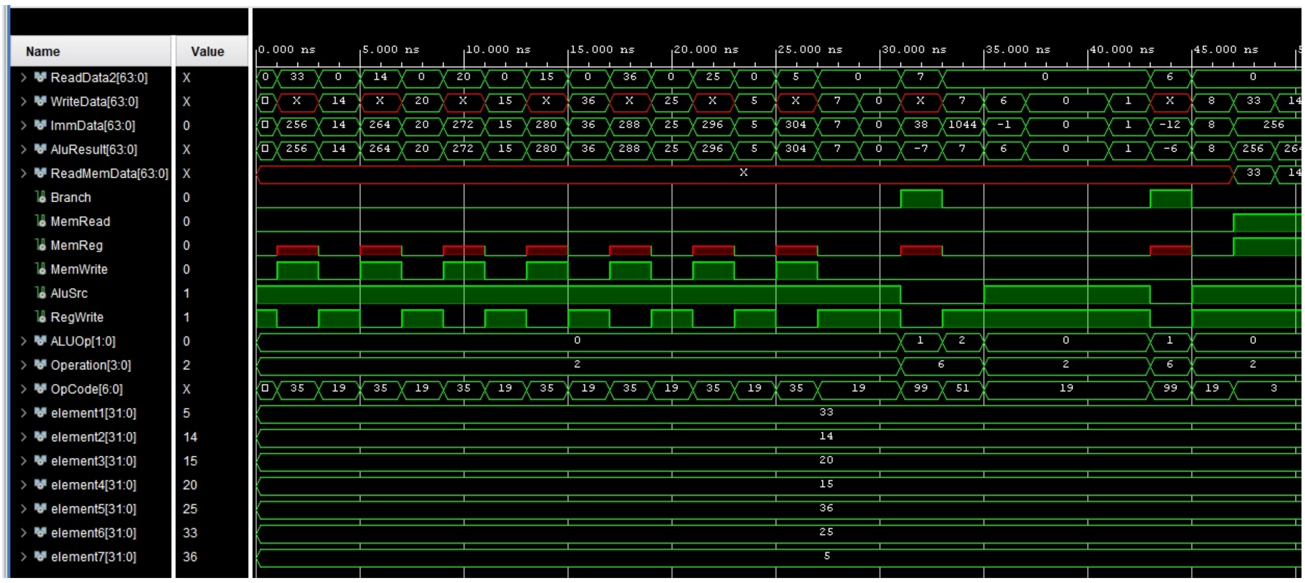


Figure 1: loading the set of inputs

sorted elements in the ascending order.

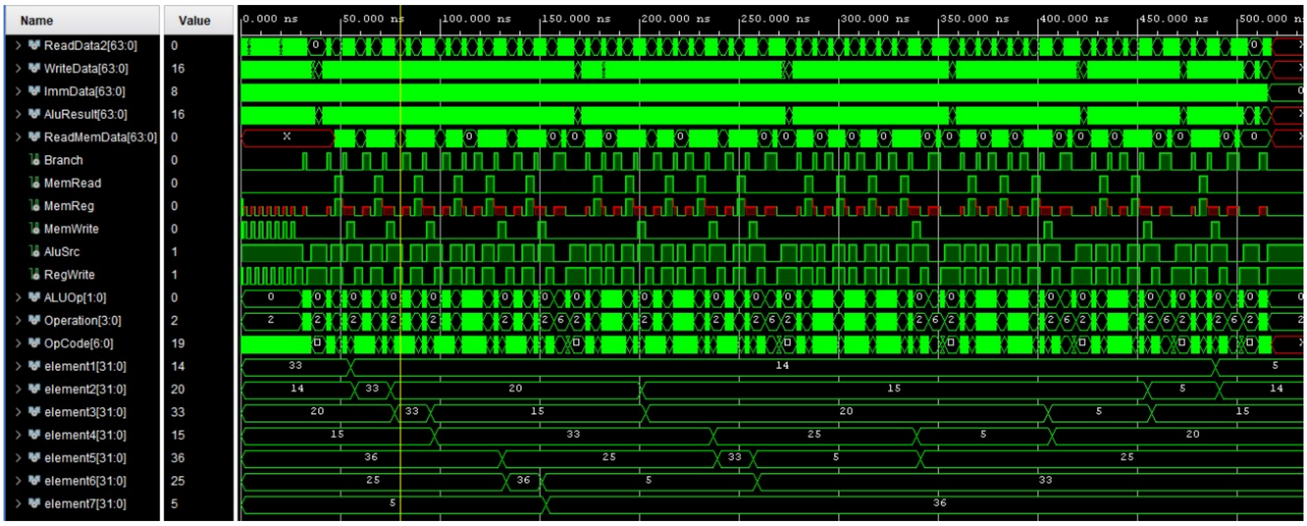


Figure 2: loading the set of inputs

# 4 Task 2 - Introducing Pipeline Stages

One issue with single cycle processor implementation is that the processor only executes one instruction at a time, which is wasteful. Only after that instruction is completed does execution of the next instruction start. It is immediately apparent how wasteful this would be and how much processing power it would spend given that the bulk of the components in our processors would be inactive. Because of this, we'll attempt to address it in this part by including pipelining into our single-cycle processor.

Pipelining would enable us to run many instructions simultaneously. The next section will go into more detail about how this works, but for now just think of it as one component working on one part of the instruction while the other is working on a different part at the same time, boosting the effectiveness of the entire programme. Our Risc-V processor will have a five-stage pipeline that will enable it to process five instructions simultaneously. The following are the five stages we put into the processor:

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execution or address calculation
- MEM: Data Memory Access
- WB: Write back

We will be introducing four new registers to implement the pipelining stage and to make our program more efficient. These registers are as follows:

- IF/ID register: This register will be used to store the instruction fetched in the IF stage and will be used in the ID stage.
- ID/EX register: This register will be used to store the instruction decoded in the ID stage and will be used in the EX stage.
- EX/MEM register: This register stores the result of the execution stage.
- MEM/WB register: This register stores the result of the memory access stage.

The pipelining procedure is aided by these four recently introduced pipeline registers. These registers monitor each instruction's progress through the pipeline and enable the pipeline to process numerous instructions at once. By allowing the execution of many instructions in parallel, the utilisation of these registers enhances the processor's performance.

A pipeline that continuously moves forward and just provides and moves instructions would be great. With the pipeline that was explained to us, however, this is not the case. It can choose between the branch address from the MEM stage and the PC's incremented PC.

We will add a control line, a forwarding unit, and the four intermediate pipeline registers. To store the control lines that are passed from one stage to another, we extend these registered. These registers would be timed to the clock and would either flush on each positive edge or send the stored contents for additional processing.

Now let's examine the modifications made to the single cycle processor in order to incorporate pipelining. We shall describe each stage of the pipelining process separately, along with its importance, in order to clarify the adjustments that have been made.

## 4.1 Stage 1 - Instruction Fetch (IF)

The first action of our CPU is the instruction fetch (IF) stage. This step is in charge of reading the instruction from memory, as the name suggests. To accomplish this, it first uses the PC counter to determine the address of the instruction to be read, reads the instruction from the Instruction memory module, and then uses the IF/ID register to pass it to the following stage. If there is a problem, this also takes care of the jump address.

Listing 8: IF/ID Register

```
module IFID(clk, reset, PC_In, Inst_input,Inst_output, PC_Out, IFIDWrite,
    flush);

  input wire clk;
  input reset;
  input wire [63:0] PC_In;
  input  [31:0] Inst_input;
  input wire IFIDWrite;
  output reg [31:0]Inst_output;
  output reg [63:0] PC_Out;
  input flush;

  always @ (posedge clk or posedge reset)
    begin
    if (reset)
      begin
         PC_Out <= PC_Out;
         Inst_output <= 0;
      end
     else if (~IFIDWrite)
        begin
          PC_Out <= PC_Out;
          Inst_output <= Inst_output;
        end
     else
       begin
         PC_Out = PC_In;
         Inst_output <= Inst_input;
       end

       if (flush) begin
      Inst_output <= 32'd0;
    end
    end
endmodule
```

## 4.2 Stage 2 - Instruction Decode (ID)

The second stage of our pipeline handles instruction decoding, register reading, and register writing. So, it starts by instructing the IF stage to fetch the instruction. The 32-bit instruction is passed on to the instruction parser and the data extractor module after being decoded and having its opcode, rd, rs1, and rs2 identified. The RegisterFile then reads the register contents or writes back to them. It should be noted that writing back requires signals from the MEM/WEB register, indicating that it is a right-to-left operation, but it doesn't stop programme flow.

Listing 9: IDEX Register

```
module IDEX(clk, reset, Funct_inp, ALUOp_inp, MemtoReg_inp, RegWrite_inp,
    Branch_inp, MemWrite_inp, MemRead_inp, ALUSrc_inp, ReadData1_inp,
    ReadData2_inp, rd_inp, rs1_in, rs2_in, imm_data_inp, PC_In, PC_Out,
    Funct_out, ALUOp_out, MemtoReg__out, RegWrite_out, Branch_out,
    MemWrite_out, MemRead_out, ALUSrc_out, ReadData1_out, ReadData2_out,
    rs1_out, rs2_out, rd_out, imm_data_out, flush, fun3, fun3_out);

  input wire clk;
  input reset;
  input [3:0] Funct_inp;
  input [1:0] ALUOp_inp;
  input MemtoReg_inp;
  input RegWrite_inp;
  input wire Branch_inp;
  input MemWrite_inp;
  input MemRead_inp;
  input ALUSrc_inp;
  input [63:0] ReadData1_inp;
  input [63:0]ReadData2_inp;
  input [4:0] rd_inp;
  input [4:0] rs1_in;
  input [4:0] rs2_in;
  input [63:0] imm_data_inp;
  input wire [63:0] PC_In;
  output reg [63:0] PC_Out;
  output reg [3:0] Funct_out;
  output reg [1:0] ALUOp_out;
  output reg MemtoReg__out;
  output reg RegWrite_out;
  output reg Branch_out;
  output reg MemWrite_out;
  output reg MemRead_out;
  output reg ALUSrc_out;
  output reg [63:0] ReadData1_out;
  output reg [63:0]ReadData2_out;
  output reg [4:0] rs1_out;
  output reg [4:0] rs2_out;
  output reg [4:0] rd_out;
  output reg [63:0] imm_data_out;
  input flush;
  input [2:0] fun3;
  output reg [2:0] fun3_out;

   always @ (posedge clk or posedge reset)
     begin
       if (reset)
         begin
           PC_Out<= 0;
           Funct_out <= 0;
           ALUOp_out <= 0;
           MemtoReg__out <= 0;
           RegWrite_out <= 0;
           Branch_out <= 0;
           MemWrite_out <= 0;
           MemRead_out <= 0;
           ALUSrc_out <= 0;
           ReadData1_out <= 0;
           ReadData2_out <= 0;
           rs1_out <= 0;
           rs2_out <= 0;
           rd_out <= 0;
           imm_data_out <= 0;
           fun3_out <= 0;
         end
       else
         begin
           PC_Out<= PC_In;
           Funct_out <= Funct_inp ;
           ALUOp_out <= ALUOp_inp;
           MemtoReg__out <= MemtoReg_inp;
           RegWrite_out <= RegWrite_inp;
```

```
            Branch_out <= Branch_inp;
            MemWrite_out <= MemWrite_inp;
            MemRead_out <= MemRead_inp;
            ALUSrc_out <= ALUSrc_inp;
            ReadData1_out <=  ReadData1_inp;
            ReadData2_out <=  ReadData2_inp;
            rs1_out <= rs1_in;
            rs2_out <= rs2_in;
            rd_out <= rd_inp;
            imm_data_out <= imm_data_inp;
            fun3_out <= fun3;
          end
          if (flush == 1'b1) begin
         ALUOp_out <= 0;
         MemtoReg__out <= 0;
         RegWrite_out <= 0;
         Branch_out <= 0;
         MemWrite_out <= 0;
         MemRead_out <= 0;
         ALUSrc_out <= 0;
                end
     end
endmodule
```

## 4.3 Stage 3 - Execution (EX)

The execution stage is the third stage in our workflow. The following two major tasks must be completed by this stage.

1. If the instruction is a branch instruction, the adder determines the offset value that must be added in order to determine the address of the subsequent location.
2. The ALU resides here, so all the operations are executed here.

After we acquired the ALUop from the Instruction Decode register, which is the control line for the ALU, the value that is to be transmitted to the registers is controlled by the two MUX.

Listing 10: EX/MEM Register

```
module EXMEM(clk, reset, rd_inp, Branch_inp, MemWrite_inp, MemRead_inp,
    MemtoReg_inp, RegWrite_inp, PC_In, Result_inp, ZERO_inp, data_inp,
    data_out, PC_Out, rd_out, Branch_out, MemWrite_out, MemRead_out,
    MemtoReg_out, RegWrite_out, Result_out, ZERO_out, flush, pos, pos_mem,
    f3, f3_out);

  input clk;
  input reset;
  input [4:0] rd_inp;
  input wire Branch_inp;
  input MemWrite_inp;
  input MemRead_inp;
  input MemtoReg_inp;
  input RegWrite_inp;
  input wire [63:0] PC_In;
  input [63:0] Result_inp;
  input ZERO_inp;
  input [63:0] data_inp;
  output reg [63:0] data_out;
  output reg [63:0] PC_Out;
  output reg [4:0] rd_out;
   output reg Branch_out;
  output reg MemWrite_out;
  output reg MemRead_out;
  output reg MemtoReg_out;
  output reg RegWrite_out;
  output reg [63:0] Result_out;
  output reg ZERO_out;
  input flush;
  input pos;
  output reg pos_mem;
  input [2:0] f3;
  output reg [2:0] f3_out;


  always @ (posedge clk or posedge reset)
     begin
       if (reset)
         begin
           PC_Out<= 0;
           Result_out <=0;
           ZERO_out <= 0;
                   MemtoReg_out <= 0;
                   RegWrite_out <= 0;
                   Branch_out <= 0;
                   MemWrite_out <= 0;
                   MemRead_out <= 0;
                   rd_out <= 0;
                   data_out <= 0;
                   pos_mem<=0;
                   f3_out <= 0;
         end
       else
         begin
           PC_Out<= PC_In;
           Result_out <= Result_inp;
           ZERO_out <= ZERO_inp ;
                   MemtoReg_out <= MemtoReg_inp;
                   RegWrite_out <= RegWrite_inp;
                   Branch_out <= Branch_inp;
                   MemWrite_out <= MemWrite_inp;
                   MemRead_out <= MemRead_inp;
                   rd_out <= rd_inp;
                   data_out <= data_inp;
                   pos_mem <= pos;
                   f3_out <= f3;
         end
         if (flush == 1'b1) begin
                   MemtoReg_out <= 0;
                   RegWrite_out <= 0;
                   Branch_out <= 0;
                   MemWrite_out <= 0;
                   MemRead_out <= 0;
         end
     end
endmodule
```

## 4.4 Stage 4 - Memory Access (MEM)

Data Memory is the only module at this stage, but it also functions as a register for sending back signals. So, before performing the operation and setting the control lines to write data to or retrieve data from the memory, it checks to see if MemRead or MemWrite is high. Results might be forwarded in order to manage data risks. As a result, the MEM/WB sends additional control signals along with the contents of the register to the pipeline's last stage. The next phase of pipelining is implemented as follows:

Listing 11: MEM/WB Register

```
module MEM_WB (
    input clk ,
    input reset ,
    input reg_write ,
    input memtoreg ,
    input [4:0] rd ,
    input [63:0] ALU_result ,
    input [63:0] read_data ,
    output reg reg_write_out ,
    output reg mem_to_reg_out ,
    output reg [4:0] rd_out ,
    output reg [63:0] ALU_result_out ,
    output reg [63:0] read_data_out
);

    always @ (posedge clk or reset)
    begin
        if (reset == 1'b1)
        begin
            rd_out = 0;
            ALU_result_out = 0;
            read_data_out = 0;
            reg_write_out = 0;
            mem_to_reg_out = 0;
        end
        else if (clk)
        begin
            rd_out = rd;
            ALU_result_out = ALU_result;
            read_data_out = read_data;
            reg_write_out = reg_write;
            mem_to_reg_out = memtoreg;
        end
    end
endmodule
```

## 4.5 Testcases

In order to test our implementation, we run the following testcases:

Listing 12: testcases

```
inst_mem[0]=8'b0;
inst_mem[1]=8'b0;
inst_mem[2]=8'b0;
inst_mem[3]=8'b0;

// sub x1, x3, x2
```

```
inst_mem[4]=8'b00110011;
inst_mem[5]=8'b00010000;
inst_mem[6]=8'b00010000;
inst_mem[7]=8'b01000000;

// add x4, x1, x2
inst_mem[8]=8'b00110011;
inst_mem[9]=8'b00010001;
inst_mem[10]=8'b00010010;
inst_mem[11]=8'b00000000;

// add x5, x4, x1
inst_mem[12]=8'b00110011;
inst_mem[13]=8'b00010010;
inst_mem[14]=8'b00010001;
inst_mem[15]=8'b00000000;
```

### 4.6 Results

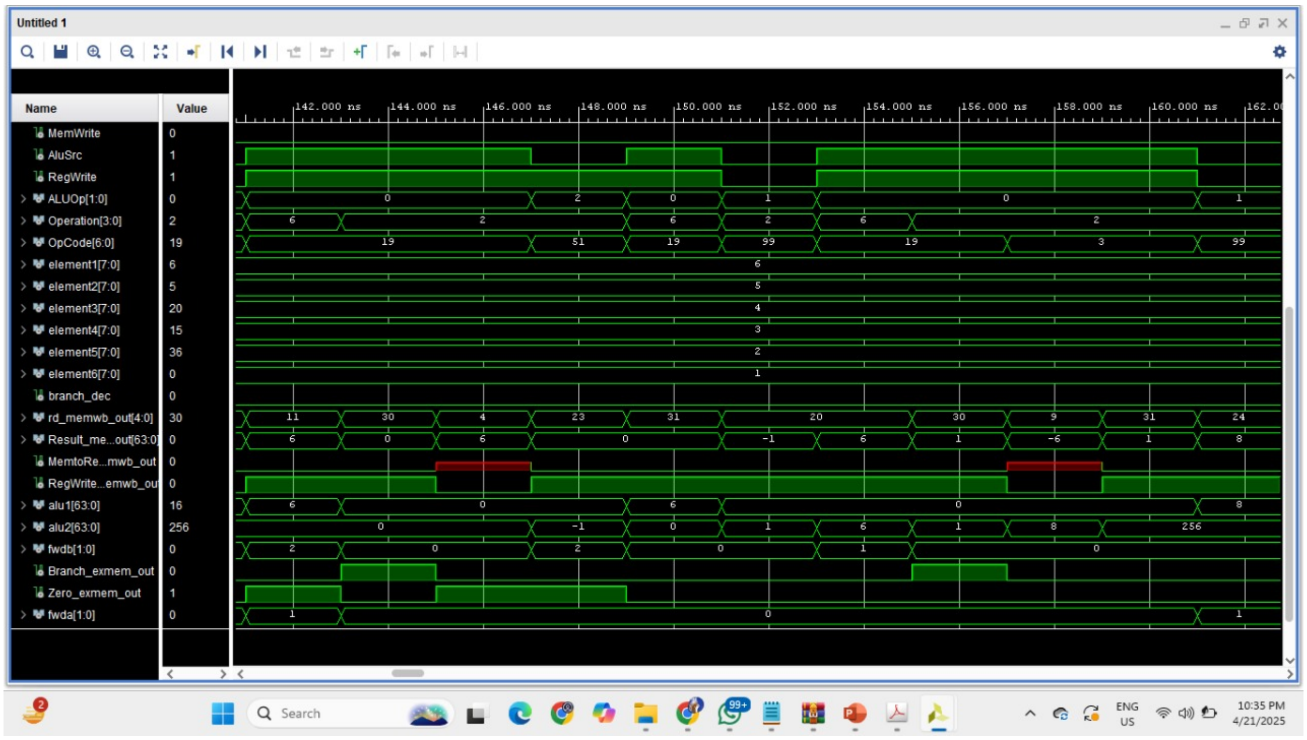


Figure 3: TestCases Results

## 5 Task 3 - Circuitry to Detect Hazards

### 5.1 Forwarding Unit

Let us say we have to run an arbitrary set of instructions on the pipelined version of the processor.

Listing 13: Arbitrary Set of instructions

```
sub x1 , x3 , x2
add x4 , x1 , x2
add x5 , x4 , x1
```

The first instruction runs without any issue. The second instruction would be in the Instruction decoding stage when the first instruction would be in the Execution stage. In this particular case, the value in x1 for the second instruction should be the sum of the values in x2 and x3, which would not be the value that the second instruction reads. The use of forwarding can eliminate such a data hazard. Forwarding sends the value instantaneously after it has been calculated in the execution stage and is essential in the ID stage so that we do not have to wait for it to be loaded into the register before we read from it. The following is the implementation of a forwarding unit in RISC-V:

Listing 14: Forwarding Unit

```
module Forwarding_Unit
(
    input [4:0] EXMEM_rd, MEMWB_rd,
    input [4:0] IDEX_rs1, IDEX_rs2,
    input EXMEM_RegWrite, EXMEM_MemtoReg,
    input MEMWB_RegWrite,
    output reg [1:0] fwd_A, fwd_B
);

always @(*) begin

    if (EXMEM_rd == IDEX_rs1 && EXMEM_RegWrite && EXMEM_rd != 0)
        begin
            fwd_A = 2'b10;
        end
    else if (MEMWB_RegWrite && MEMWB_rd!=0 && MEMWB_rd==IDEX_rs1 )
        begin
            fwd_A = 2'b01;
        end
    else
        begin
            fwd_A = 2'b00;
        end


    if ((EXMEM_rd == IDEX_rs2) && (EXMEM_RegWrite) && (EXMEM_rd != 0))
        begin
            fwd_B = 2'b10;
        end

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

    else
        begin
            fwd_B = 2'b00;
        end
end

endmodule // Forwarding_Unit
```

There are three scenarios for forwarding. The first one is EX Hazard, where we select the value from register EX/MEM, which is the output of the preceding instruction to either of the ALU's inputs. The second scenario is the one where the result is occasionally required directly from the MEM stage for which we take it directly from the MEM stage. third scenario is the one where we send the original input of the instruction.

Listing 15: MUX3to1

```
module MUX3to1
(
    input [63:0] a, b, c,
    input [1:0] sel,
    output reg [63:0] data_out
);

always@(*)
begin
    case (sel)
    2'b00: data_out = a;
    2'b01: data_out = b;
    2'b10: data_out = c;
        default: data_out = 2'bX;
    endcase
end
endmodule
```

## 5.2 Hazard Detection Unit

A crucial part of pipelined processors, the hazard detection unit identifies and resolves possible hazards introduced because of the pipelining of instructions. It makes it possible for the processor to manage instruction dependencies and prevent pipeline stalls or data hazards, which boosts processor performance.

Listing 16: Hazard Detection Unit

```
module HazardDetection (
    input [4:0] rs1_ID ,
    input [4:0] rs2_ID ,
    input [4:0] rd_EX ,
    input MemRead_Ex ,
    output reg PC_Write ,
    output reg Ctrl ,
    output reg IF_ID_Write
);
    always @ (*) begin
        if (MemRead_Ex && (rd_EX == rs1_ID || rd_EX == rs2_ID))
        begin
            IF_ID_Write <= 1'b0;
            PC_Write <= 1'b0 ;
            Ctrl <= 1'b0;
        end
        else begin
            IF_ID_Write <= 1'b1 ;
            PC_Write <= 1'b1;
            Ctrl <= 1'b1 ;
        end
    end
endmodule
```

Three input signals rs1 ID, rs2 ID, rd EX, and MemRead Ex, and outputs three signals Ctrl, PC Write, and IF ID write are used by the hazard detection unit. The inputs rs1 ID and rs2 ID stand in for the instruction's two source registers, which were fetched in the cycle before. The destination register of the instruction that was decoded in the preceding cycle is represented by the input rd EX. If the present instruction is a load instruction that reads data from memory, it will be indicated by the control signal MemRead that is present in the input. The hazard detection unit determines if the previous instruction was a load instruction that read data from memory and whether any of the source registers of the present instruction match the destination register of the previous instruction. A data hazard occurs if both conditions are met, in which case the hazard detection unit adjusts the output signals. The Ctrl signal is set to 0, and a stall is introduced. The result from the MEM/WB pipeline stage instead of the EX/MEM pipeline stage. If there is no data hazard, then normal forwarding occurs.

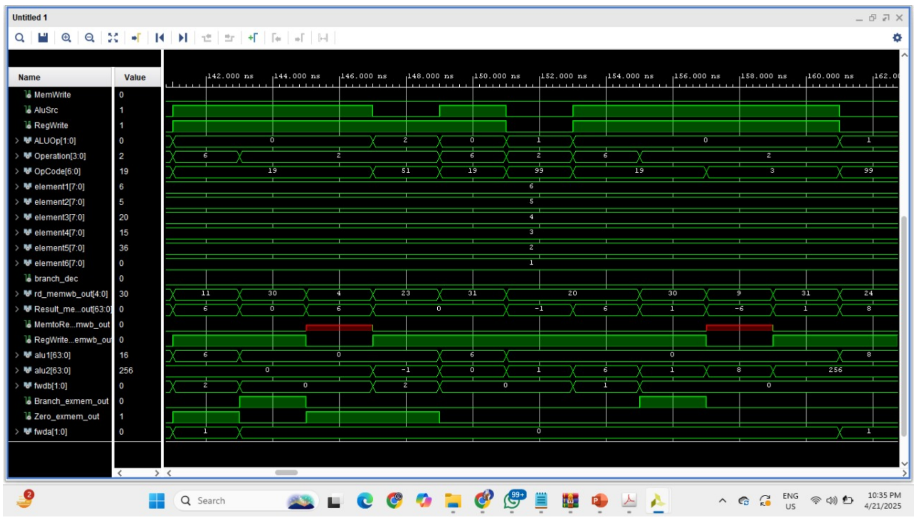
Listing 17: Hazard Detection MUX

```
module CU_mux ( Mux_Write , Branch , MemRead , MemtoReg , MemWrite
, ALUSrc , RegWrite , ALUOp , Branch_out , MemRead_out ,
MemtoReg_out , MemWrite_out , ALUSrc_out , RegWrite_out ,
ALUOp_out ) ;
input Mux_Write ;
input Branch ;
input MemRead ;
input MemtoReg ;
input MemWrite ;
input ALUSrc ;
input RegWrite ;
input [1:0] ALUOp ;
output reg Branch_out ;
output reg MemRead_out ;
output reg MemtoReg_out ;
output reg MemWrite_out ;
output reg ALUSrc_out ;
output reg RegWrite_out ;
output reg [1:0] ALUOp_out ;

always@ (*)
begin
    if (~Mux_Write)
    begin
        Branch_out = 0;
        MemRead_out = 0;
        MemtoReg_out = 0;
        MemWrite_out = 0;
        ALUSrc_out = 0;
        RegWrite_out = 0;
        ALUOp_out = 0;
    end
    else
    begin
        Branch_out = Branch ;
        MemRead_out = MemRead ;
        MemtoReg_out = MemtoReg ;
        MemWrite_out = MemWrite ;
        ALUSrc_out = ALUSrc ;
        RegWrite_out = RegWrite ;
        ALUOp_out = ALUOp ;
    end
end
endmodule
```

## 5.3 Result for the Hazard Detection Unit

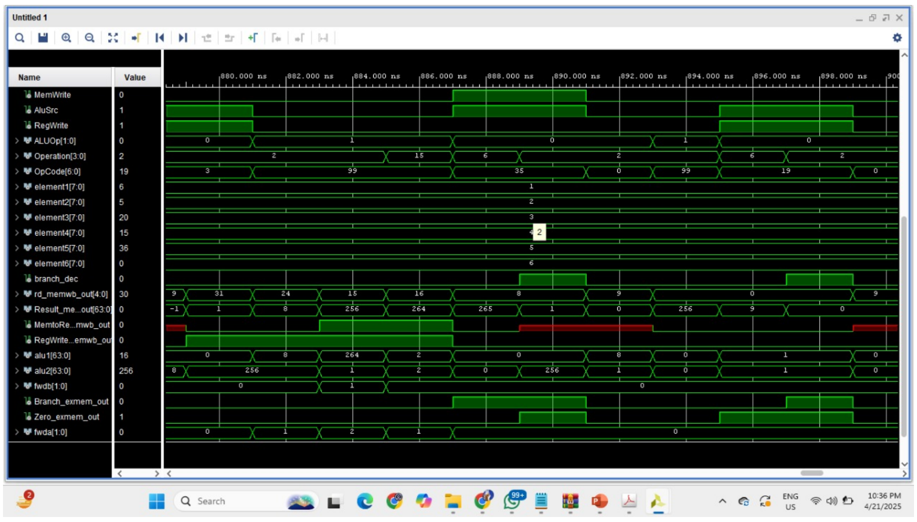
When Ctrl is zero, stalling occurs and the instruction becomes zero. This is because the instruction is not being executed and is being stalled. As a result of this, RS and RD also become zero. The control signals are also set to zero.



Untitled 1

| Name | Value |
|---|---|
| MemWrite | 0 |
| AluSrc | 1 |
| RegWrite | 1 |
| ALUOp[1:0] | 0 |
| Operation[3:0] | 2 |
| OpCode[6:0] | 19 |
| element1[7:0] | 6 |
| element2[7:0] | 5 |
| element3[7:0] | 20 |
| element4[7:0] | 15 |
| element5[7:0] | 36 |
| element6[7:0] | 0 |
| branch_dec | 0 |
| rd_memwb_out[4:0] | 30 |
| Result_me...out[63:0] | 0 |
| MemtoRe...mwb_out | 0 |
| RegWrite...emwb_out | 0 |
| alu1[63:0] | 16 |
| alu2[63:0] | 256 |
| fwdb[1:0] | 0 |
| Branch_exmem_out | 0 |
| Zero_exmem_out | 1 |
| fwda[1:0] | 0 |

Figure 4: Results before sorting



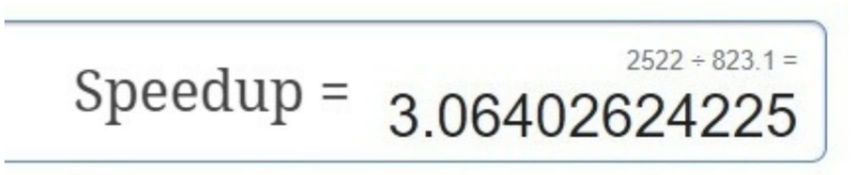
Untitled 1

| Name | Value |
|---|---|
| MemWrite | 0 |
| AluSrc | 1 |
| RegWrite | 1 |
| ALUOp[1:0] | 0 |
| Operation[3:0] | 2 |
| OpCode[6:0] | 19 |
| element1[7:0] | 6 |
| element2[7:0] | 5 |
| element3[7:0] | 20 |
| element4[7:0] | 15 |
| element5[7:0] | 36 |
| element6[7:0] | 0 |
| branch_dec | 0 |
| rd_memwb_out[4:0] | 30 |
| Result_me...out[63:0] | 0 |
| MemtoRe...mwb_out | 0 |
| RegWrite...emwb_out | 0 |
| alu1[63:0] | 16 |
| alu2[63:0] | 256 |
| fwdb[1:0] | 0 |
| Branch_exmem_out | 0 |
| Zero_exmem_out | 1 |
| fwda[1:0] | 0 |

Figure 5: Results after sorting

# 6 Comparison between Pipelined and Non-Pipelined Single Cycle Processor

We can observe that our speedup is almost three times greater when comparing the pipelined and non-pipelined versions of the processor. This occurs because the instructions are carried out concurrently in the pipelined version. In the non-pipelined version, each instruction is executed in a single clock cycle, while in the pipelined version, the clock cycle is reduced to one execution stage of the instruction, which reduces the clock cycle time.


2522 ÷ 823.1 =

Speedup = 3.06402624225

Figure 6: speedup

## 7 Task Division

The entire project was an a group effort by the three members of the group. To be specific, Faraz made the bubble sort logic, and then we worked on Task 1 together. Task 2 was done by misha and Ahla. Lastly, we all worked on task 3 together.

## 8 Final Comments

The project presented a special difficulty because it necessitated rigorous amounts of debugging the code and modules to identify the issue. Our experiment was a success since our processor could use the Bubble Sort algorithm to sort an unsorted array and return its sorted form. Despite facing a number of obstacles during the project, we overcame them and fixed mistakes to produce a multicycle, pipelined processor that, in principle, should be more effective than its single-cycle counterpart.

## 9 Challenges

We did not have Vivado on our laptops so the only time that we could do the project was when we free during the university hours. Furthermore, certain modules such as the forwarding and hazard detection were a bit tricky to implement. Being able to sort 7 elements throughout was also a challenge on its own.

## 10 GitHub Repository

GitHub Repository: CA Project Spring 2025