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

Owais Aijaz (oa07610)

Muhammad Khubaib Mukaddam (mk07218)

Contents

1 Introduction

The goal of this project is to create a five-stage pipelined processor that can run a single array sorting algorithm. Verilog HDL is used to design the processor, and RISC-V assembly language is used to write the sorting algorithm. We first implemented the sorting algorithm on a single-cycle processor, which is then converted into a pipelined one. We also implemented the forwarding unit and hazard detection module to speed up the execution as much as possible. The report is divided into sections for each task that we were required to complete in accordance with the project's criteria.

2 Task 1 - Sorting Algorithm on a Single Cycle Processor

2.1 Bubble Sort Assembly Code

```
    addi x11 , x0 , 6                # an arbitrary value to
        append in array
    addi x29 , x0 , 6                # initializing size of
        the array to be 6
    addi x30 , x0 , 0                # initializing offset to
         store values inarray after one another
    addi x31 , x0 , 0                # initializing i = 0 to
        loop through array toenter values .
    addi x28 , x0 , 6                # temporary reg for
        checking length

    #The code below is to intialize random values in the
        array
    Array:
    sw x11 , 0x100 ( x30 )           # store values in array
    addi x31 , x31 , 1               # performs i = i + 1
    addi x30 , x30 , 8               # offset + 4 to jump to
        next memoryaddress to store value
    addi x11 , x11 , -1              # subtracting 1 to add
        next value inarray (6 - >5 - >4....)
    beq x28 , x31 , filled           # if i = size of array ,
         stop .
    beq x0 , x0 , Array

    filled:
    addi x11, x0, 6
    addi x30, x0, 0  #i

    I_Loop:                          # Code for i loop
    beq x11, x30 , Exit              # if i = size of array ,
         array hasbeen sorted
    sub x20, x11, x30                    # x20 is (n-i)
    addi x20, x20, -1                    # x20 is (n-i-1)
    addi x31,  x0,  0                    # j = 0 (for j loop )
    addi x23, x0, 0
    addi x30, x30,  1

    J_Loop:                              # Code below is for
        nested j loop
    beq x31, x20, I_Loop                 # j = n-i-1
    addi x31,x31, 1
    addi x24, x23, 8
    lw x15, 0x100(x23)                   # load Array [j]
    lw x16, 0x100(x24)                   # load Array [j+1]
    blt x16 , x15 , SWAP                 # if Array [j] < Array [
         j+1 ]
    beq x0, x0, J_Loop

    SWAP:                                # Code for swapping
    sw x16 , 0x100( x23 )                # [j] = [j+1]
    sw x15 , 0x100( x24 )                # [j+1] = [j]
    addi x23, x23, 8
    beq x0 , x0 , J_Loop

    Exit:
```

Listing 1: Bubble Sort Assembly code

2.2 Bubble Sort Python Code

```python
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
        if array[j] > array[j + 1]:

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

Listing 2: Bubble Sort Python Code (Taken from GeeksforGeeks)

3 Changes to Single Cycle Processor

In task1, 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 6 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

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

    initial
    begin
        //0x00600593
        inst_mem[0] = 8'b10010011;
        inst_mem[1] = 8'b00000101;
        inst_mem[2] = 8'b01100000;
        inst_mem[3] = 8'b00000000;

        //0x00600e93
        inst_mem[4] = 8'b10010011;
        inst_mem[5] = 8'b00001110;
        inst_mem[6] = 8'b01100000;
        inst_mem[7] = 8'b00000000;

        //0x00000f13
        inst_mem[8] = 8'b00010011;
        inst_mem[9] = 8'b00001111;
        inst_mem[10] = 8'b00000000;
        inst_mem[11] = 8'b00000000;

        //0x00000f93
        inst_mem[12] = 8'b10010011;
        inst_mem[13] = 8'b00001111;
        inst_mem[14] = 8'b00000000;
        inst_mem[15] = 8'b00000000;

        //0x00600e13
        inst_mem[16] = 8'b00010011;
        inst_mem[17] = 8'b00001110;
        inst_mem[18] = 8'b01100000;
        inst_mem[19] = 8'b00000000;

        //0x10bf2023
        inst_mem[20] = 8'b00100011;
        inst_mem[21] = 8'b00100000;
        inst_mem[22] = 8'b10111111;
        inst_mem[23] = 8'b00010000;

        //0x001f8f93
        inst_mem[24] = 8'b10010011;
        inst_mem[25] = 8'b10001111;
        inst_mem[26] = 8'b00011111;
        inst_mem[27] = 8'b00000000;

        //0x008f0f13
        inst_mem[28] = 8'b00010011;
        inst_mem[29] = 8'b00001111;
        inst_mem[30] = 8'b10001111;
        inst_mem[31] = 8'b00000000;

        //0xfff58593
        inst_mem[32] = 8'b10010011;
        inst_mem[33] = 8'b10000101;
        inst_mem[34] = 8'b11110101;
        inst_mem[35] = 8'b11111111;

        //0x01fe0463
        inst_mem[36] = 8'b01100011;
        inst_mem[37] = 8'b00000100;
        inst_mem[38] = 8'b11111110;
        inst_mem[39] = 8'b00000001;

        //0xfe0006e3
        inst_mem[40] = 8'b11100011;
        inst_mem[41] = 8'b00000110;
        inst_mem[42] = 8'b00000000;
        inst_mem[43] = 8'b11111110;

        //0x00600593
        inst_mem[44] = 8'b10010011;
        inst_mem[45] = 8'b00000101;
        inst_mem[46] = 8'b01100000;
        inst_mem[47] = 8'b00000000;

        //0x00000f13
        inst_mem[48] = 8'b00010011;
        inst_mem[49] = 8'b00001111;
        inst_mem[50] = 8'b00000000;
        inst_mem[51] = 8'b00000000;

        //0x05e58263
        inst_mem[52] = 8'b01100011;
        inst_mem[53] = 8'b10000010;
        inst_mem[54] = 8'b11100101;
        inst_mem[55] = 8'b00000101;

        //0x00000b93
        inst_mem[56] = 8'b10010011;
        inst_mem[57] = 8'b00001011;
        inst_mem[58] = 8'b00000000;
        inst_mem[59] = 8'b00000000;

        //0x00000f93
        inst_mem[60] = 8'b10010011;
        inst_mem[61] = 8'b00001111;
        inst_mem[62] = 8'b00000000;
        inst_mem[63] = 8'b00000000;

        //0xfffa0a13
        inst_mem[64] = 8'b00010011;
        inst_mem[65] = 8'b00001010;
        inst_mem[66] = 8'b11111010;
        inst_mem[67] = 8'b11111111;

        //0x41e58a33
        inst_mem[68] = 8'b00110011;
        inst_mem[69] = 8'b10001010;
        inst_mem[70] = 8'b11100101;
        inst_mem[71] = 8'b01000001;

        //0x001f0f13
        inst_mem[72] = 8'b00010011;
        inst_mem[73] = 8'b00001111;
        inst_mem[74] = 8'b00011111;
        inst_mem[75] = 8'b00000000;

        //0xff4f84e3
        inst_mem[76] = 8'b11100011;
        inst_mem[77] = 8'b10000100;
        inst_mem[78] = 8'b01001111;
        inst_mem[79] = 8'b11111111;

        //0x001f8f93
        inst_mem[80] = 8'b10010011;
        inst_mem[81] = 8'b10001111;
        inst_mem[82] = 8'b00011111;
        inst_mem[83] = 8'b00000000;

        //0x008b8c13
        inst_mem[84] = 8'b00010011;
        inst_mem[85] = 8'b10001100;
        inst_mem[86] = 8'b10001011;
        inst_mem[87] = 8'b00000000;

        //0x100ba783
        inst_mem[88] = 8'b10000011;
        inst_mem[89] = 8'b10100111;
        inst_mem[90] = 8'b00001011;
        inst_mem[91] = 8'b00010000;

        //0x100c2803
        inst_mem[92] = 8'b00000011;
        inst_mem[93] = 8'b00101000;
        inst_mem[94] = 8'b00001100;
        inst_mem[95] = 8'b00010000;

        //0x00f84463
        inst_mem[96] = 8'b01100011;
        inst_mem[97] = 8'b01000100;
        inst_mem[98] = 8'b11111000;
        inst_mem[99] = 8'b00000000;

        //0xfe0004e3
        inst_mem[100] = 8'b11100011;
        inst_mem[101] = 8'b00000100;
        inst_mem[102] = 8'b00000000;
        inst_mem[103] = 8'b11111110;

        //0x110ba023
        inst_mem[104] = 8'b00100011;
        inst_mem[105] = 8'b10100000;
        inst_mem[106] = 8'b00001011;
        inst_mem[107] = 8'b00010001;

        //0x10fc2023
        inst_mem[108] = 8'b00100011;
        inst_mem[109] = 8'b00100000;
        inst_mem[110] = 8'b11111100;
        inst_mem[111] = 8'b00010000;

        //0x008b8b93
        inst_mem[112] = 8'b10010011;
        inst_mem[113] = 8'b10001011;
        inst_mem[114] = 8'b10001011;
        inst_mem[115] = 8'b00000000;

        //0xfc000ce3
        inst_mem[116] = 8'b11100011;
        inst_mem[117] = 8'b00001100;
        inst_mem[118] = 8'b00000000;
        inst_mem[119] = 8'b11111100;

        //06400413
        inst_mem[120] = 8'b00010011;
        inst_mem[121] = 8'b00000100;
        inst_mem[122] = 8'b01000000;
        inst_mem[123] = 8'b00000110;
    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
```

Listing 3: Changes to Instruction Memory

3.2 Data Memory

```
    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
    );
        reg [7:0] DataMemory [304:0];
        integer i;
        initial
        begin
          for (i = 0; i < 300; i = i + 1)
          begin
            DataMemory[i] = 0;
          end
          DataMemory[256]  = 8'd32;
          DataMemory[264]  = 8'd24;
          DataMemory[272] = 8'd62;
          DataMemory[280] = 8'd59;
          DataMemory[288] = 8'd0;
        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];

        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
```

Listing 4: Changes to Data Memory

3.3 Changes to Control Unit

We made changes in the Control Unit module to incorporate branch instructions. In which, 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.

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

    always @ (Opcode)
    begin
        case (Opcode)
            7'b0110011: //R type (51)
                begin
                    Branch = 0;
                    MemRead = 0;
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc = 0;
                    RegWrite = 1;
                    ALUOp = 2'b10;
                end
            7'b0000011: //ld (3)
                begin
                    Branch = 0;
                    MemRead = 1;
                    MemtoReg = 1;
                    MemWrite = 0;
                    ALUSrc = 1;
                    RegWrite = 1;
                    ALUOp = 2'b00;
                end
            7'b0010011: //addi (19)
                begin
                    Branch = 0;
                    MemRead = 0;
                    MemtoReg = 0;
                    MemWrite = 0;
                    ALUSrc = 1;
                    RegWrite = 1;
                    ALUOp = 2'b00;
                end

            7'b0100011: // I type SD  (35)
                begin
                    Branch = 0;
                    MemRead = 0;
                    MemtoReg = 1'bx;
                    MemWrite = 1;
                    ALUSrc = 1;
                    RegWrite = 0;
                    ALUOp = 2'b00;
                end
            7'b1100011://SB type blt and beq  99
```

```
                begin
                    Branch = 1;
                    MemRead = 0;
                    MemtoReg = 1‘bx;
                    MemWrite = 0;
                    ALUSrc = 0;
                    RegWrite = 0;
                    ALUOp = 2‘b01;
                end
            default:
               begin
                ALUSrc   = 1‘b0;
                MemtoReg = 1‘b0;
                RegWrite = 1‘b0;
                MemRead  = 1‘b0;
                MemWrite = 1‘b0;
                Branch   = 1‘b0;
                ALUOp    = 2‘b00;
               end
        endcase
    end
endmodule
```

Listing 5: Changes to Control Unit

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.

```
module ALU_Control
(
    input [1:0] ALUOp,
    input [3:0] Funct,
    output reg [3:0] Operation
);

    always @(*)
    begin
        case(ALUOp)
    2‘b00:
        begin
        Operation = 4'b0010;
        end
        2'b01:                          // branch type instructions
            begin
            case(Funct[2:0])
            3'b000:                  // beq
                begin
                Operation = 4'b0110;  // subtract
                end
            3'b100:                  // blt
                begin
                Operation = 4'b0100; // less than operation
                end
                endcase
            end


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

Listing 6: Changes to ALU Control Unit

3.5 Changes to ALU

We modified our ALU to execute correct branch results. If first value is less than second value, 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 result from ALU is anded with the branch signal. When selection line of 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.

```verilog
module ALU_64_bit
    (
        input [63:0]a, b,
        input [3:0] ALUOp,

        output reg [63:0] Result,
        output ZERO
    );

    localparam [3:0]
    AND = 4'b0000,
    OR  = 4'b0001,
    ADD = 4'b0010,
    Sub = 4'b0110,
    NOR = 4'b1100,
    Less = 4'b0100;

    assign ZERO = (Result == 0);

    always @ (ALUOp, a, b)
    begin
        case (ALUOp)
            AND: Result = a & b;
            OR:  Result = a | b;
            ADD: Result = a + b;
            Sub: Result = a - b;
            NOR: Result = ~(a | b);
            Less: Result = (a < b) ? 0 : 1;    //less than
                operation

            default: Result = 0;
        endcase
    end

endmodule
```

Listing 7: Changes to ALU 64 bit

3.6 Results for Single Cycle Pipeline

Firstly, we build an array in the data memory by initializing 6 values in descending order.

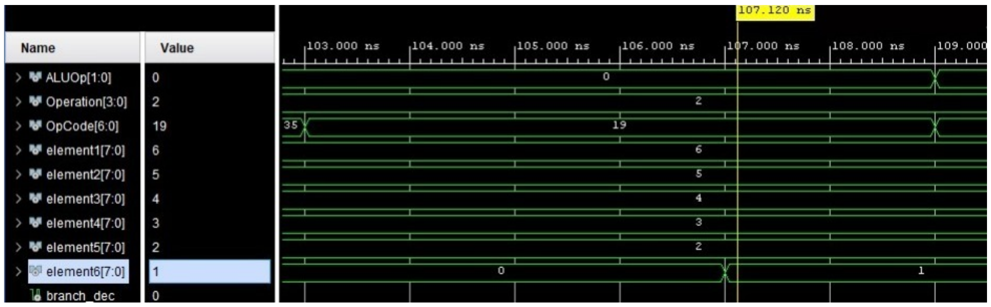


Figure 1: Loading the set of inputs

Final sorted elements in the ascending order.

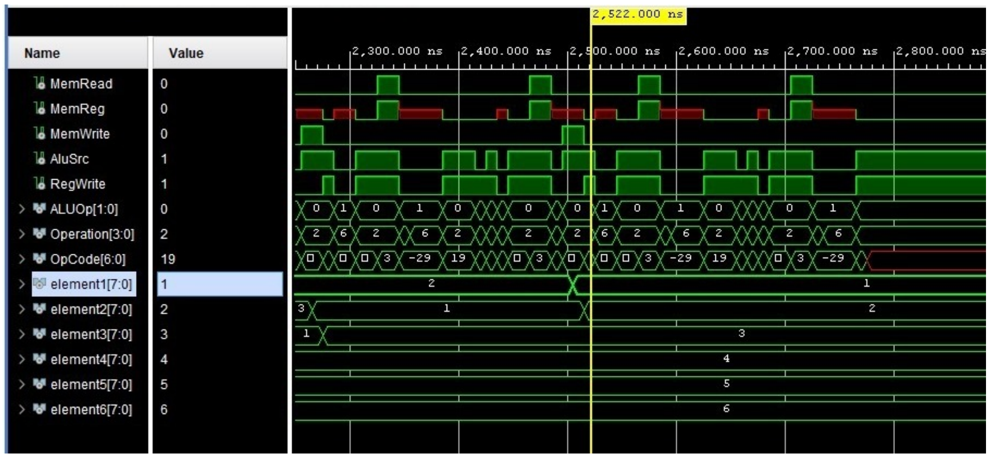


Figure 2: Final Sorted set of elements

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:

1. IF: Instruction Fetch
2. ID: Instruction Decode
3. EX: Execution or address calculation
4. MEM: Data Memory Access
5. 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:

1. 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.
2. 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.
3. EX/MEM register: This register stores the result of the execution stage.
4. 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.

The following is the module used in the stage.

```
module IF_ID(
    input clk,
    input reset,
    input [31:0] instruction,
    input [63:0] PC_Out,
    input IF_write,
    output reg [31:0] IF_ID_instruction,
    output reg [63:0] IF_ID_PCOut
    );

    always @(posedge clk or reset)
        begin
            if (reset == 1‘b1)
                begin
                    IF_ID_instruction = 0;
                    IF_ID_PCOut = 0;
                end
            else if (clk==1 || IF_write == 1)
                begin
                    IF_ID_instruction = instruction;
                    IF_ID_PCOut = PC_Out;
                end
        end
endmodule
```

Listing 8: IF/ID Register

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.

```
module ID_EX(
    input clk,
    input reset,
    input branch,
    input MemRead,
    input MemtoReg,
    input MemWrite,
    input ALUsrc,
    input RegWrite,
    input [1:0] ALU_Op,
    input [63:0] readdata1,
    input [63:0] readdata2,
    input [63:0] immediate,
    input [63:0] pc_out,
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] rd ,
    input[3:0] func,
    output reg branch_out, MemRead_out, MemtoReg_out,
       MemWrite_out, ALUsrc_out, RegWrite_out,
    output reg [1:0] AlU_Op_out,
    output reg [63:0]  readdata1_out,readdata2_out,
       immediate_out,pc_out_out,
    output reg [4:0] rs1_out, rs2_out, rd_out ,
    output reg [3:0] func_out
    );

        always @(*)
        begin
            if (reset==1'b1)
            begin

                branch_out = 0;
                MemRead_out=0;
                MemtoReg_out=0;
                MemWrite_out=0;
                ALUsrc_out=0;
                AlU_Op_out=0;
                RegWrite_out=0;
                readdata1_out=0;
                readdata2_out=0;
                immediate_out=0;
                pc_out_out=0;
                rs1_out= 0;
                rs2_out=0;
                rd_out=0;
                func_out=0;

            end
    else if (clk==1)
    begin
        MemRead_out=MemRead;
        MemtoReg_out=MemtoReg;
        MemWrite_out=MemWrite;
        ALUsrc_out=ALUsrc;
        AlU_Op_out=ALU_Op;
        RegWrite_out=RegWrite;
        readdata1_out=readdata1;
        readdata2_out=readdata2;
        immediate_out=immediate;
        pc_out_out=pc_out;
        rs1_out= rs1;
        rs2_out=rs2;
        rd_out=rd;
        func_out=func;
        end
    end
endmodule
```

Listing 9: ID/EX Register

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.

```
module EX_MEM(
    input clk, reset,
    input [4:0] rd,
    input [63:0] write_data ,
    //input branch_MUX,
    input [63:0] ALU_result, PC_out,
    input zero, branch, MemRead, MemWrite, RegWrite,
        MemtoReg,
    output reg [4:0] rd_out,
    output reg [63:0] write_data_out,
    output reg [63:0] ALU_result_out,
    output reg zero_out, branch_out, MemRead_out,
        MemWrite_out, RegWrite_out, MemtoReg_out,
    output reg [63:0] PC_out_out,
    output reg branch_MUX_out
    );

    always @(posedge clk ,posedge reset)
        begin
            if (reset==1)
                begin
                    PC_out_out=0;
                    rd_out = 0;
                    branch_out=0;
                    MemRead_out=0;
                    MemWrite_out=0;
                    RegWrite_out=0;
                    MemtoReg_out=0;
                    write_data_out=0;
                    ALU_result_out = 0;
                    branch_MUX_out=0;
                    zero_out=0;
                end

        else if (clk==1)
        begin
            PC_out_out=PC_out;
            rd_out=rd ;
            write_data_out=write_data;
            MemRead_out=MemRead;
            MemWrite_out=MemWrite;
            RegWrite_out= RegWrite ;
            MemtoReg_out=MemtoReg ;
            ALU_result_out=ALU_result ;
            branch_MUX_out=ALU_result ;
            zero_out= zero;
            branch_out=branch;
        end
    end
endmodule
```

Listing 10: EX/MEM Register

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:

```
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
```

Listing 11: MEM/WB Register

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.

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

Listing 12: Arbitrary Set of instructions

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.

```
module Forwarding_Unit(
    input [4:0] ID_EX_Rs1,
    input [4:0] ID_EX_Rs2,
    input [4:0] EX_MEM_Rd,
    input EX_MEM_RegWrite,
    input [4:0] MEM_WB_Rd,
    input MEM_WB_RegWrite,
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
    );

        always @(*)
            begin
            if (EX_MEM_RegWrite == 1 && EX_MEM_Rd ==
                ID_EX_Rs1 && EX_MEM_Rd != 0)
            begin
                Forward_A = 2'b10;   //10
            end
        else if (MEM_WB_Rd == ID_EX_Rs1 && MEM_WB_RegWrite
            == 1 && MEM_WB_Rd != 0 &&
                !(EX_MEM_RegWrite == 1 && EX_MEM_Rd != 0 &&
                    EX_MEM_Rd == ID_EX_Rs1))
            begin
                Forward_A = 2'b01;  //01
            end
        else

            begin
                Forward_A = 2'b00; //00
            end

        //FORWARD B LOGIC
        if (EX_MEM_RegWrite == 1 && EX_MEM_Rd == ID_EX_Rs2
            && EX_MEM_Rd != 0)
            begin
                Forward_B = 2'b10;   //10
            end
        else if (MEM_WB_Rd == ID_EX_Rs2 && MEM_WB_RegWrite
            == 1 && MEM_WB_Rd != 0 &&
                 !(EX_MEM_RegWrite == 1 && EX_MEM_Rd != 0 &&
                     EX_MEM_Rd == ID_EX_Rs2))
            begin
                Forward_B = 2‘b01;  //01
            end
        else
            begin
                Forward_B = 2‘b00;  //00
            end
        end
endmodule
```

Listing 13: 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.

```
module Four_MUX(
    input [63:0] a, b, c, d,
    input [1:0] sel,
    output reg [63:0] mux_result
    );

    always @(*)
        begin
          if (sel==2‘b00)
            mux_result=a;
          else if (sel ==2‘b01)
            mux_result=b;
          else if (sel==2‘b10)
            mux_result=c;
          else if (sel==2‘b11)
            mux_result=d;
    end
endmodule
```

Listing 14: Four MUX

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.

```
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
```

Listing 15: Hazard Detection Unit

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.

```
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);
```

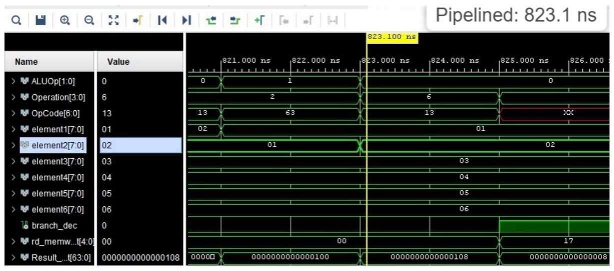
```

  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
```

Listing 16: Hazard Detection MUX

This Mux changes the values of the control signal to zero if the Ctrl output of the hazard detection unit is zero otherwise the values remain the same.

5.3 Result for the Hazard Detection unit

When Ctrl is zero, stalling occurs and 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 becomes zero. The control signals are also set to zero.

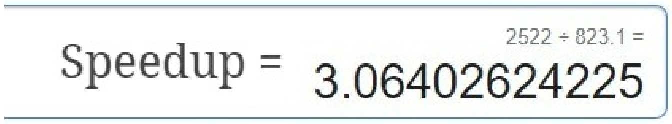


Name	Value
ALUOp[1:0]	0
Operation[3:0]	6
OpCode[6:0]	13
element1[7:0]	01
element2[7:0]	02
element3[7:0]	03
element4[7:0]	04
element5[7:0]	05
element6[7:0]	06
branch_dec	0
rd_memw...t[4:0]	00
Result_...t[63:0]	0000000000000108

Figure 3: Hazard detection final sorted

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 non pipelined version, each instruction is executed in a single clock cycle, while in pipeline version, clock cycle is reduced to one execution stage of the instruction which reduces the clock cycle time.



Speedup = 2522 ÷ 823.1 = 3.06402624225

Figure 4: Speedup of Non-Pipelined vs Pipelined Processor

7 Task Division

Assembly language code for bubble sort was written and the forwarding unit was implemented by Muhammad Khubaib. The single cycle processor and hazard detection was implemented by Owais Aijaz.

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 multi-cycle, 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.

10 Github Repository

`https://github.com/Khubaib2002/Computer_Architecture_project`