



CE-321L/CS-330L: Computer Architecture

5-State Pipelined Processor

Ahmed Khalid, Salman Adnan, Ibrahim Rana

30th April 2024

Contents:

1. Introduction
2. Task 1 – Sorting Algorithm (Single Cycled)
 - a. Bubble Sort Implementation
 - b. Simulation
3. Changes to the Single Cycled Processor
4. Task 2
 - a. IF/ID
 - b. ID/EX
 - c. EX/MEM
 - d. MEM/WB
 - e. Simulation
5. Task 3
 - a. Forwarding Unit
 - b. Mux 3x1
 - c. Hazard Detection Unit
 - d. Mux 2x1
6. Task 4
7. Challenges
8. Task Division
9. Conclusion
10. GitHub

1. Introduction

Project Overview: Implementing a pipelined processor for array sorting

The purpose of this project is to implement array sorting with a 5-stage pipelined processor written using the RISC-V assembly language. The software used is Verilog HDL.

Goal:

This project's primary goal is to gradually improve processor architecture. It starts with building a 5-stage pipelined processor and progresses to implementing a sorting algorithm in RISC-V assembly on a single-cycle CPU. The sorting algorithm should be effectively executed by this enhanced processor in a significantly shorter amount of time. Achieving functional implementation is not the only goal; data, control, and structural dangers present in pipelined designs must also be addressed and minimized. This guarantees that the array sorting method will run smoothly.

Project Structure:

- We start by using a traditional single-cycle processor to lay the foundation.
- The architecture is then improved by switching to a five-stage pipeline and optimizing it for faster execution.
- Every phase of the project's development, following the given instructions, will be recorded in the report.

2. Task 1 – Sorting Algorithm

Bubble Sort Code: (This is implemented within the Instruction Memory)

```
module InstructionMemory(  
    input [63:0] Inst_Address,  
    output reg [31:0] Instruction  
);  
    reg [7:0] im [95:0];  
  
    initial  
    begin  
        im[0] = 8'h63;//blt x22, x8, CompareElements #8  
        im[1] = 8'h44;  
        im[2] = 8'h8B;  
        im[3] = 8'h00;  
  
        im[4] = 8'h63;// beq x0, x0, EndBubbleSort #92  
        im[5] = 8'h0E;  
        im[6] = 8'h00;  
        im[7] = 8'h04;  
  
        im[8] = 8'h93;//CompareElements:  
        //addi x23, x22, 0  
        im[9] = 8'h0B;  
        im[10] = 8'h0B;  
        im[11] = 8'h00;  
  
        im[12] = 8'h93;//addi x29, x28, 0  
        im[13] = 8'h0E;
```

im[14] = 8'h0E;

im[15] = 8'h00;

im[16] = 8'h63;//blt x23, x8, SwapElements #8

im[17] = 8'hC4;

im[18] = 8'h8B;

im[19] = 8'h00;

im[20] = 8'hE3;//beq x0, x0, BubbleSort # -20

im[21] = 8'h06;

im[22] = 8'h00;

im[23] = 8'hFE;

im[24] = 8'h03;//SwapElements:

// lw x12, 0(x28)

im[25] = 8'h36;

im[26] = 8'h0E;

im[27] = 8'h00;

im[28] = 8'h83;//lw x13, 0(x29)

im[29] = 8'hB6;

im[30] = 8'h0E;

im[31] = 8'h00;

im[32] = 8'h63;// blt x12, x13, IncrementIndex #28

im[33] = 8'h4E;

im[34] = 8'hD6;

im[35] = 8'h00;

im[36] = 8'h93;//addi x23, x23, 1

im[37] = 8'h8B;

im[38] = 8'h1B;

im[39] = 8'h00;

im[40] = 8'h93;// addi x29, x29, 8

im[41] = 8'h8E;

im[42] = 8'h8E;

im[43] = 8'h00;

im[44] = 8'hE3;// blt x23, x8, SwapElements # -20

im[45] = 8'hC6;

im[46] = 8'h8B;

im[47] = 8'hFE;

im[48] = 8'h13;//addi x22, x22, 1

im[49] = 8'h0B;

im[50] = 8'h1B;

im[51] = 8'h00;

im[52] = 8'h13;//addi x28, x28, 8

im[53] = 8'h0E;

im[54] = 8'h8E;

im[55] = 8'h00;

im[56] = 8'hE3;//beq x0, x0, BubbleSort

im[57] = 8'h04;

im[58] = 8'h00;

im[59] = 8'hFC;

im[60] = 8'hB3;//IncrementIndex:

// add x5, x12, x0

im[61] = 8'h02;

im[62] = 8'h06;

im[63] = 8'h00;

im[64] = 8'h23;//sw x13, 0(x28)

im[65] = 8'h30;

im[66] = 8'hDE;

im[67] = 8'h00;

im[68] = 8'h23;//sw x5, 0(x29)

im[69] = 8'hB0;

im[70] = 8'h5E;

im[71] = 8'h00;

im[72] = 8'h93;//addi x23, x23, 1

im[73] = 8'h8B;

im[74] = 8'h1B;

im[75] = 8'h00;

im[76] = 8'h93;//addi x29, x29, 8

im[77] = 8'h8E;

im[78] = 8'h8E;

im[79] = 8'h00;

```
im[80] = 8'hE3;//blt x23, x8, SwapElements
```

```
im[81] = 8'hC4;
```

```
im[82] = 8'h8B;
```

```
im[83] = 8'hFC;
```

```
im[84] = 8'h13;// addi x22, x22, 1
```

```
im[85] = 8'h0B;
```

```
im[86] = 8'h1B;
```

```
im[87] = 8'h00;
```

```
im[88] = 8'h13;// addi x28, x28, 8
```

```
im[89] = 8'h0E;
```

```
im[90] = 8'h8E;
```

```
im[91] = 8'h00;
```

```
im[92] = 8'hE3;//beq x0, x0, BubbleSort
```

```
im[93] = 8'h02;
```

```
im[94] = 8'h00;
```

```
im[95] = 8'hFA;
```

```
end//EndBubbleSort:
```

```
always @(Inst_Address)// making one instruction from 4 parts of im (instruction memory)
```

```
begin
```

```
    Instruction = {im[Inst_Address + 3], im[Inst_Address + 2], im[Inst_Address + 1],  
im[Inst_Address]};
```


```
end
```

```
endmodule
```


2.2. Simulation

Initial:

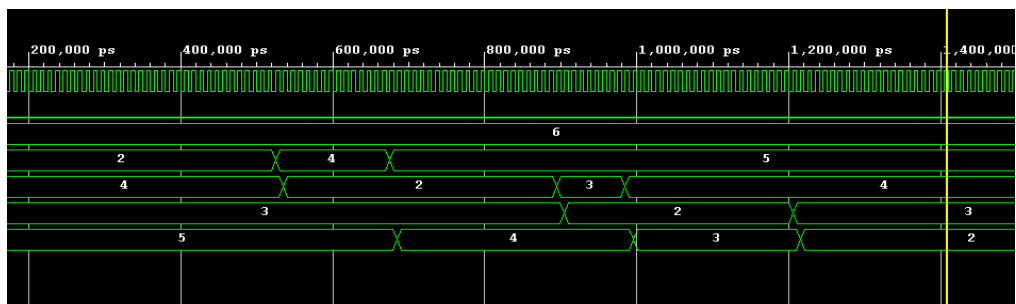
Name	Value
clk	0
reset	1
> e5[63:0]	6
> e4[63:0]	2
> e3[63:0]	4
> e2[63:0]	3
> e1[63:0]	5



Final:

Name	Value
clk	1
reset	0
> e5[63:0]	6
> e4[63:0]	5
> e3[63:0]	4
> e2[63:0]	3
> e1[63:0]	2

Process:



3. Changes To the Single Cycled Processor

3.1. ALU

```
module ALU64bit(
    input [63:0] A,
    input [63:0] B,
    input [3:0] ALUOp,
    output reg [63:0] Result,
    output Zero //output for zero result
)
always @(*) begin
    case (ALUOp)
        4'b0000: Result = A & B; //AND
        4'b0001: Result = A | B; //OR
        4'b0010: Result = A + B; //ADD
        4'b0110: Result = A - B; //SUB
        4'b1100: Result = ~(A | B); //NOR
        4'b0100: Result = (A < B) ? 0 : 1; // Lesser than comparison
        default: Result = 0; //Default case for unsupported ALUOP values
    endcase
end
assign Zero = (Result == 0); // checks if the result is zero.
endmodule
```

Explanation: We made a couple of changes here, firstly handling branch instructions where when comparing two values if one is smaller we set it to zero, mimicking the BEQ instruction. This was done through the mux with a selection line branch and zero. If branch is zero, we update PC + 4 directly, else if zero is high indicating a condition then the branch target replaces the PC. Executing the branch operation.

3.2. ALUControl

```
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:  
                begin  
                    case(Funct[2:0])  
                        3'b000:          // BEQ  
                            begin  
                                Operation = 4'b0110; // SUBTRACT  
                            end  
                        3'b100:          // BLT  
                            begin  
                                Operation = 4'b0100; // < THAN  
                            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

```

Explanation:

Two new inputs have been added to the improved ALU Control module: a 2-bit ALUOp control field and the Func Field. With this update, the module can now dynamically generate a 4-bit ALU Control input, which specifies what the ALU must do. The Func Field provides more precise operation selection by combining elements from the funct7 and funct3 fields.

The ALU performs addition when ALUOp is set to "00," which denotes load and

store instructions. But in the case of ALUOp values of "10" or "01," the particular operation is determined by the encoding present in the instruction's funct7 and funct3 fields.

When ALUOp is set to "01," indicating a branch-type instruction, an exception takes place. To handle the particular operation related to branch instructions in this circumstance, the ALU Control unit uses a custom case structure.

To put it simply, the updated ALU Control unit takes the combined values of Func and ALUOp to dynamically determine the ALU operation. This means that different types of instructions, like load, store, branch, and others, can be accommodated, which improves the functionality and flexibility of the processor architecture.

3.3. Control Unit

```
`timescale 1ns / 1ps
```

```
module Control_Unit(
input [6:0]Opcode,
output reg Branch,
output reg MemRead,
output reg MemtoReg,
output reg [1:0]ALUOp,
output reg MemWrite,
output reg ALUSrc,
output reg RegWrite
);
always @(*)
begin
//this is for R-type instruction
if(Opcode == 7'b0110011)
begin
Branch = 0;
MemRead = 0;
MemtoReg = 0;
ALUOp = 10;
MemWrite = 0;
ALUSrc = 0;
RegWrite = 1;
end
//this is for I-type instruction(ld)
else if(Opcode == 7'b0000011)
begin
```

```

Branch = 0;
MemRead = 1;
MemtoReg = 0;
ALUOp = 00;
MemWrite = 1;
ALUSrc = 1;
RegWrite = 1;
end
//this is for S-type instruction
else if(Opcode == 7'b0100011)
begin
Branch = 0;
MemRead = 0;
MemtoReg = 1'bX;
ALUOp = 00;
MemWrite = 1;
ALUSrc = 1;
RegWrite = 0;
end
//this is for SB-type instruction
else if(Opcode == 7'b1100011)
begin
Branch = 1;
MemRead = 0;
MemtoReg = 1'bX;
ALUOp = 01;
MemWrite = 0;
ALUSrc = 0;
RegWrite = 0;
end
//this is an “addi” instruction
else if(Opcode == 7'b0010011)
begin
Branch = 0;
MemRead = 0;
MemtoReg = 1'bX;
ALUOp = 00;
MemWrite = 0;
ALUSrc = 1;
RegWrite = 1;
end
// Default case
else
begin
Branch = 1'b0;
MemRead = 1'b0;

```

```

MemtoReg = 1'b0;
ALUOp = 2'b00;
MemWrite = 1'b0;
ALUSrc = 1'b0;
RegWrite = 1'b0;
end
end
endmodule

```

Explanation:

Mostly remains unchanged, the control unit itself is taking opcodes which are important for signalling the separate control signals. The default case was adjusted as well alongside the fact that the BLT and BEQ instructions are identical.

3.4 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'd5;
    DataMemory[8] = 8'd2;
    DataMemory[16] = 8'd1;
    DataMemory[24] = 8'd7;

```

```

    DataMemory[32] = 8'd4;
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
    end
end

endmodule

```


3.5. Instruction Memory:

Updated Code already mentioned in task 1.

4. Task 2 – Pipelining

In the field of processor architecture, the implementation of a single-cycle approach poses a challenge. This method, which processes one instruction at a time, results in significant periods of inactivity across various components. To address this inefficiency, pipelining is introduced as a solution. Pipelining aims to improve processing capabilities and optimize resource utilization by dividing instruction execution into distinct stages. In our Risc-V processor framework, we introduce a five-stage pipeline that allows for the simultaneous processing of five instructions. These pipeline stages define specific phases of instruction execution, leading to improved overall efficiency and throughput.

Pipeline registers are essential components in our processor's architecture, allowing for the simultaneous processing of multiple instructions while tracking their progress through distinct stages. These registers greatly enhance processor performance by facilitating parallel instruction execution.

In addition to these registers, we incorporate a control line and a forwarding unit to further enhance the functionality of the pipeline. These components enable the smooth transfer of control signals between pipeline stages, synchronized with clock pulses. As the clock cycles, these registers either forward stored data for further processing or clear contents at each rising clock edge.

While the pipeline aims for continuous progression, practical considerations arise, such as choosing between the incremented program counter (PC) and the branch address from the MEM stage.

The adaptation of the single-cycle processor to incorporate pipelining involves delineating each pipeline stage individually, emphasizing their respective roles and importance. This meticulous approach optimizes processor efficiency by enabling the concurrent execution of multiple instructions.

IF (Instruction Fetch): Fetches the instruction.

ID (Instruction Decode): Decodes the instruction.

EX (Execution or Address Calculation): Executes the instruction or calculates addresses.

MEM (Data Memory Access): Accesses data memory.

WB (Write Back): Writes back the result.

To facilitate pipelining, four new registers are introduced:

IF/ID register: Stores the fetched instruction for use in the ID stage.

ID/EX register: Stores the decoded instruction for use in the EX stage.

EX/MEM register: Stores the result of the execution stage.

MEM/WB register: Stores the result of the memory access stage.

4.1 - Instruction Fetch/Instruction Decode (IF/ID) Stage:

IF/ID register: Stores the fetched instruction for use in the ID stage.

```
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
```

4.2 - Instruction Decode/Execution (ID/EX) Stage:

ID/EX register: Stores the decoded instruction for use in the EX stage.

```

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
);

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
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

```

4.3 - Execution/Memory (EX/MEM) Stage:

EX/MEM register: Stores the result of the execution stage.

```

module EX_MEM(
input clk, // Clock signal
input Flush, // Flush control signal
input RegWrite, // Control signal for enabling register write

```

input MemtoReg, // Control signal for selecting memory or ALU result for register write

input Branch, // Control signal for branch instruction

input Zero, // Control signal indicating the ALU result is zero

input MemWrite, // Control signal for memory write

input MemRead, // Control signal for memory read

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, // Output signal for enabling register write

output reg MemtoReg_out, // Output signal for selecting memory or ALU result for register write

output reg Branch_out, // Output signal for branch instruction

output reg Zero_out, // Output signal indicating the ALU result is zero

output reg MemWrite_out, // Output signal for memory write

output reg MemRead_out, // Output signal for memory read

output reg is_greater_out, // Output signal indicating the comparison result of the ALU operation

output reg [63:0] immvalue_added_pc_out, // Output signal for immediate value added to the program counter

output reg [63:0] ALU_result_out, // Output signal for the ALU result

output reg [63:0] WriteData_out, // Output signal for data to be written to memory or register file

output reg [3:0] function_code_out, // Output signal for function code for ALU operation

```
output reg [4:0] destination_reg_out // Output signal for destination register for register
write
);
// Assign output values based on control signals
always @(posedge clk) begin
if (Flush) begin
// Reset output values when flush signal is active
RegWrite_out = 0;
MemtoReg_out = 0;
Branch_out = 0;
Zero_out = 0;
is_greater_out = 0;
MemWrite_out = 0;
MemRead_out = 0;
immvalue_added_pc_out = 0;
ALU_result_out = 0;
WriteData_out = 0;
function_code_out = 0;
destination_reg_out = 0;
end
else begin
// Assign output values based on input signals
RegWrite_out = RegWrite;
MemtoReg_out = MemtoReg;
Branch_out = Branch;
Zero_out = Zero;
is_greater_out = is_greater;
```



```

MemWrite_out = MemWrite;
MemRead_out = MemRead;
immvalue_added_pc_out = immvalue_added_pc;
ALU_result_out = ALU_result;
WriteData_out = WriteData;
function_code_out = function_code;
destination_reg_out = destination_reg;
end
end
endmodule

```

4.4 - Memory/Write Back (MEM/WB) Stage:

MEM/WB register: Stores the result of the memory access stage.

```

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

```

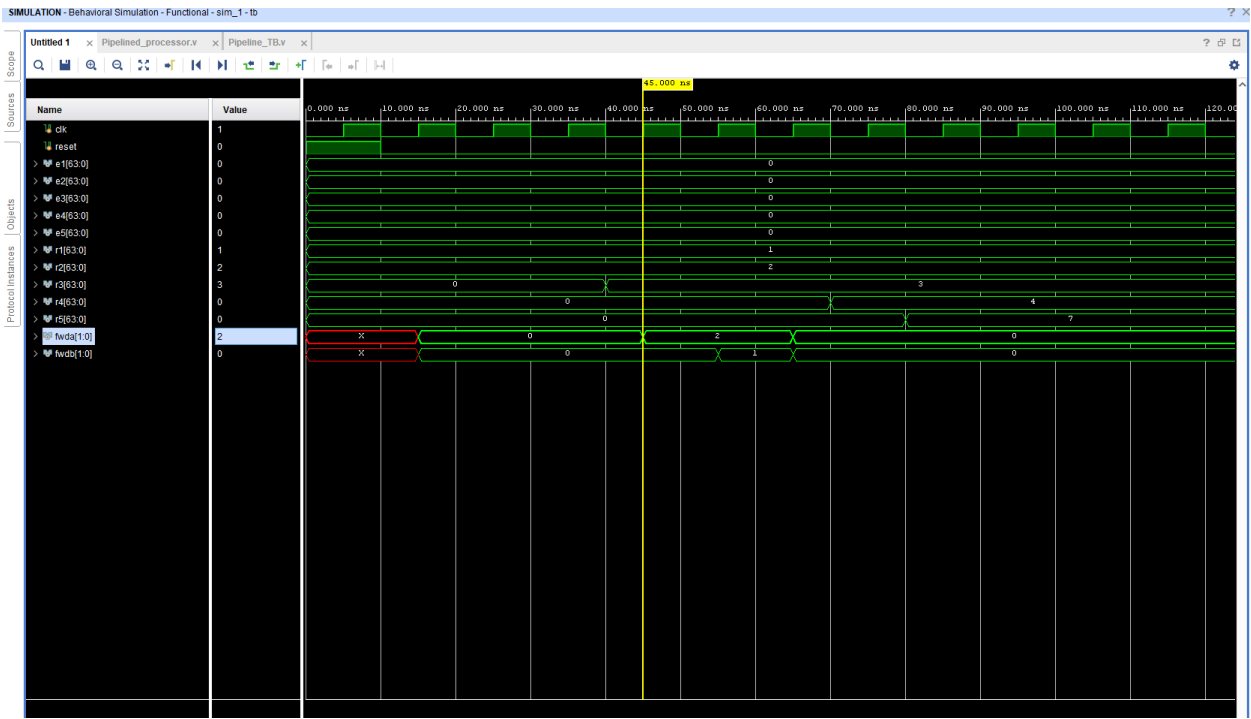
4.5 – Simulation for Task-2:

Initial:

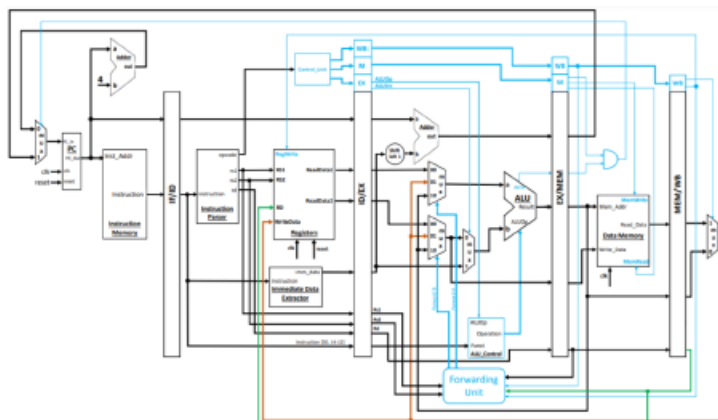
Name	Value	0 ps	1 ps	2 ps	3 ps	4 ps	5 ps
 clk	0						
 reset	0						
>  r1[63:0]	1				1		
>  r2[63:0]	2				2		
>  r3[63:0]	3				0		
>  r4[63:0]	4				0		
>  r5[63:0]	7				0		

Final:

Name		Value	999,990 ps	999,991 ps	999,992 ps	999,993 ps	999,994 ps	999,995 ps
clk		0						
reset		0						
r1[63:0]		1			1			
r2[63:0]		2			2			
r3[63:0]		3			3			
r4[63:0]		4			4			
r5[63:0]		7			7			



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.



5. Task 3 – Hazard Detection

- add x1, x2, x3
- sub x4, x1, x5
- add x6, x4, x7

Explanation:

In the given sequence, the second step (subtraction) depends on the result of the first step (addition). Without forwarding, a data hazard would occur because the processor would not have access to the value stored in register x1 (the result of the first add operation) until the write-back stage of the initial instruction is completed.

With the inclusion of a forwarding unit, the processor can directly transfer the result from the execution stage of the first instruction to the decoding stage of the second instruction. This allows the second instruction to use the computed value of x1 without waiting for it to be written back to the register file.

In the RISC-V architecture, the forwarding unit plays a crucial role in mitigating data hazards, ensuring smooth progress of dependent instructions through the pipeline. By reducing stalls and enabling a continuous flow of instructions, this mechanism enhances the efficiency of the processor.

5.1. 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
```

```
// 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

```

Explanation:

In the system described, forwarding considerations encompass three distinct scenarios. First, in the "EX Hazard" scenario, the output of the previous instruction can be sent to either input of the Arithmetic Logic Unit (ALU). To address this, a multiplexer selects the value from the register in the EX/MEM stage if the previous instruction aimed to write to the register file and if the write register number matches the read register number of ALU inputs A or B.

During data hazard situations, there are instances where the result is needed directly from the Memory (MEM) stage. This need arises when the result may be stored multiple times in a single register. To ensure the retrieval of the most recent result, it is directly obtained from the MEM stage.

The forwarding logic for both forwardA and forwardB operates according to a predetermined table of forwarding conditions, determining when and how data should be forwarded to the ALU inputs A and B.

5.2 Module Mux3x1:

```
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
```

Let's go over how the hazard is resolved for the following instructions after implementing the forwarding mechanism:

- add x1, x2, x3
- sub x4, x1, x5
- add x6, x4, x7

Now, assuming that forwarding is in place:

During the execution stage of the first instruction (add x1, x2, x3), the processor computes the result (x1). Simultaneously, the forwarding unit sends this result to the instruction decoding stage of the second instruction (sub x4, x1, x5). As a result, the second instruction can immediately use the forwarded value of x1 without waiting for it to be written back to the register file, resolving the data hazard.

This forwarding strategy ensures that the dependent instruction (sub x4, x1, x5) accesses the necessary operand (x1) as soon as it becomes available in the pipeline. This allows the processor to smoothly execute instructions without introducing stalls, thus optimizing throughput and overall performance.

5.3. Hazard Detection Unit:

In pipelined processors, the hazard detection unit plays a crucial role in identifying and resolving issues related to instruction pipelining. Its main goal is to detect and address instruction dependencies to prevent pipeline stalls and mitigate data hazards. By effectively managing these challenges, the hazard detection unit significantly improves the efficiency of the processor. In our processor architecture, we follow a specific methodology to implement and deploy this hazard detection unit effectively.

```
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: Set control signals accordingly
mux_out = 0; // Disable the multiplexer output
enable_Write = 0; // Disable write to the next pipeline stage
enable_PCWrite = 0; // Disable PC write
end else begin
// No hazard detected: Set control signals accordingly
mux_out = 1; // Enable the multiplexer output
enable_Write = 1; // Enable write to the next pipeline stage
enable_PCWrite = 1; // Enable PC write
end
end
endmodule
```


Explanation:

The hazard detection unit analyzes input signals, including "current_rd," "previous_rs1," "previous_rs2," and "current_MemRead," to generate three key output signals: "mux_out," "enable_Write," and "enable_PCWrite."

Here's a simplified breakdown of its functionality:

The unit checks if the destination register of the current instruction ("current_rd") matches either of the source registers of the previous instruction ("previous_rs1" or "previous_rs2"). It also checks if the current instruction involves a memory read operation ("current_MemRead"). If both conditions are met, indicating a potential hazard, the unit adjusts the output signals accordingly. The "mux_out" signal controls the multiplexer output, potentially favoring the result from the MEM/WB pipeline stage. Additionally, "enable_Write" and "enable_PCWrite" are set to 0, indicating that the current instruction should not update the register file or the program counter.

In hazard-free scenarios, the unit sets the output signals to 1, allowing the smooth progress of the current instruction. The "mux_out" signal is adjusted to prioritize the result from the EX/MEM pipeline stage. Furthermore, both "enable_Write" and "enable_PCWrite" are enabled (set to 1), indicating that the current instruction can update the register file and the program counter.

5.4 Module mux2x1

```
(  
input [63:0] a,b,  
input sel ,  
output [63:0] data_out  
);  
assign data_out = sel ? a : b; //select b or a based on the sel bit  
endmodule
```

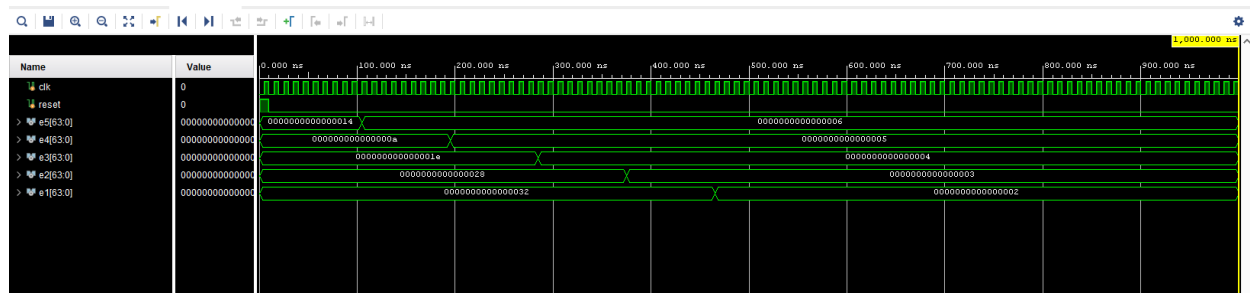
Explanation:

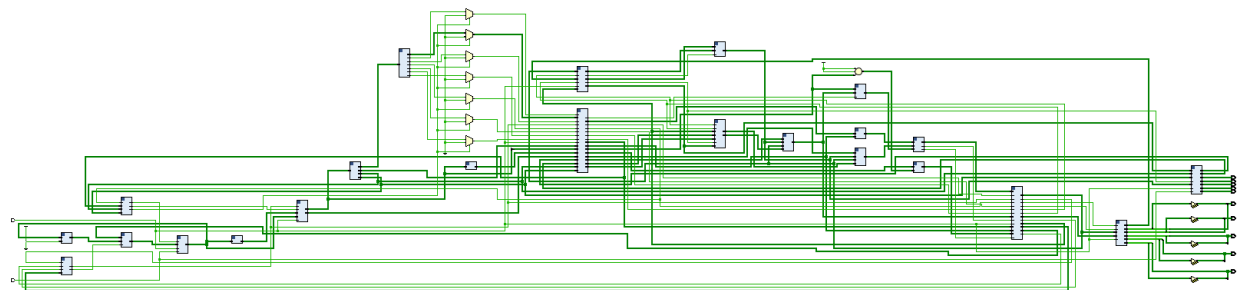
The 2x1 multiplexer dynamically selects between two sets of input signals based on the value of the "sel" input, determining the output signals—branch_eq_hazard, MemRead_hazard, MemtoReg_hazard, MemWrite_hazard, ALUsrc_hazard, RegWrite_hazard, and ALUOp_hazard.

Control signals—branch, MemRead, MemtoReg, MemWrite, ALUsrc, RegWrite, and ALUOp—are provided to the hazard detection unit as inputs. When "sel" is 0,

Conversely, when "sel" is 1, selecting the second set of input signals, the output signals are configured based on these inputs using the 2x1 multiplexer. The branch_eq_hazard output signal reflects the value of the branch input, indicating a branch hazard if the branch input is asserted. Similarly, MemRead_hazard, MemtoReg_hazard, MemWrite_hazard, ALUSrc_hazard, RegWrite_hazard, and ALUOp_hazard output signals are adjusted according to their corresponding input values, indicating the presence of specific hazards if the respective inputs are asserted.

Simulation:





6. Task 4

The way that pipelined and non-pipelined processors carry out instruction execution is the primary difference between them. Because instructions are carried out in a non-pipelined processor in a sequential fashion, there may be times when the processor sits idle while each instruction is handled separately. On the other hand, pipelined processors split up the execution of instructions into steps so that several instructions can be processed at once.

Pipelining improves processor efficiency dramatically by allowing activities to be completed in fewer cycles as compared to their non-pipelined counterparts. In one particular scenario, for example, a task executed by a pipelined processor takes 46 cycles to complete, whereas a non-pipelined counterpart requires 159 cycles. The efficiency of pipelining is demonstrated by this significant decrease in cycle count.

In particular, the pipelined method outperforms its non-pipelined counterpart by a factor of 3.45. This enhancement demonstrates the efficiency that pipelining provides, as it

7. Challenges

There were a multitude of challenges, keeping code consistent between multiple code bases proved to be a challenge even with the use of GitHub. Many changes needed to be made between previous port configurations. Figuring out the conversion from the assembly code was another issue. Between task 2 and task 3, figuring out pipelining and configuring the code was also an issue between transitioning from the original base.

8. Task Division

Ahmed Khalid – Task 1, Task 4

Salman Adnan – Task 2, 3

Ibrahim Rana – Task 2, 3

9. Conclusion

The project was particularly difficult because it required extensive debugging of the modules as well as the code. Over the course of the project, we encountered many obstacles, but we persevered because we learned from our mistakes, corrected them, and eventually created a pipelined, multi-cycle processor. Theoretically speaking, this novel design offers greater efficiency over its single-cycle equivalent, indicating the possibility of substantial improvements in CPU architecture. And was overall a fantastic learning experience that has helped shape our understanding of how processing works.

10. Github Link

https://github.com/Reckon1ng/RISC_V-Pipelined-Processor