April 22, 2024

CE-321L/CS-330L
Computer Architecture
5-Stage Pipelined Processor To Execute A Single Array Sorting Algorithm

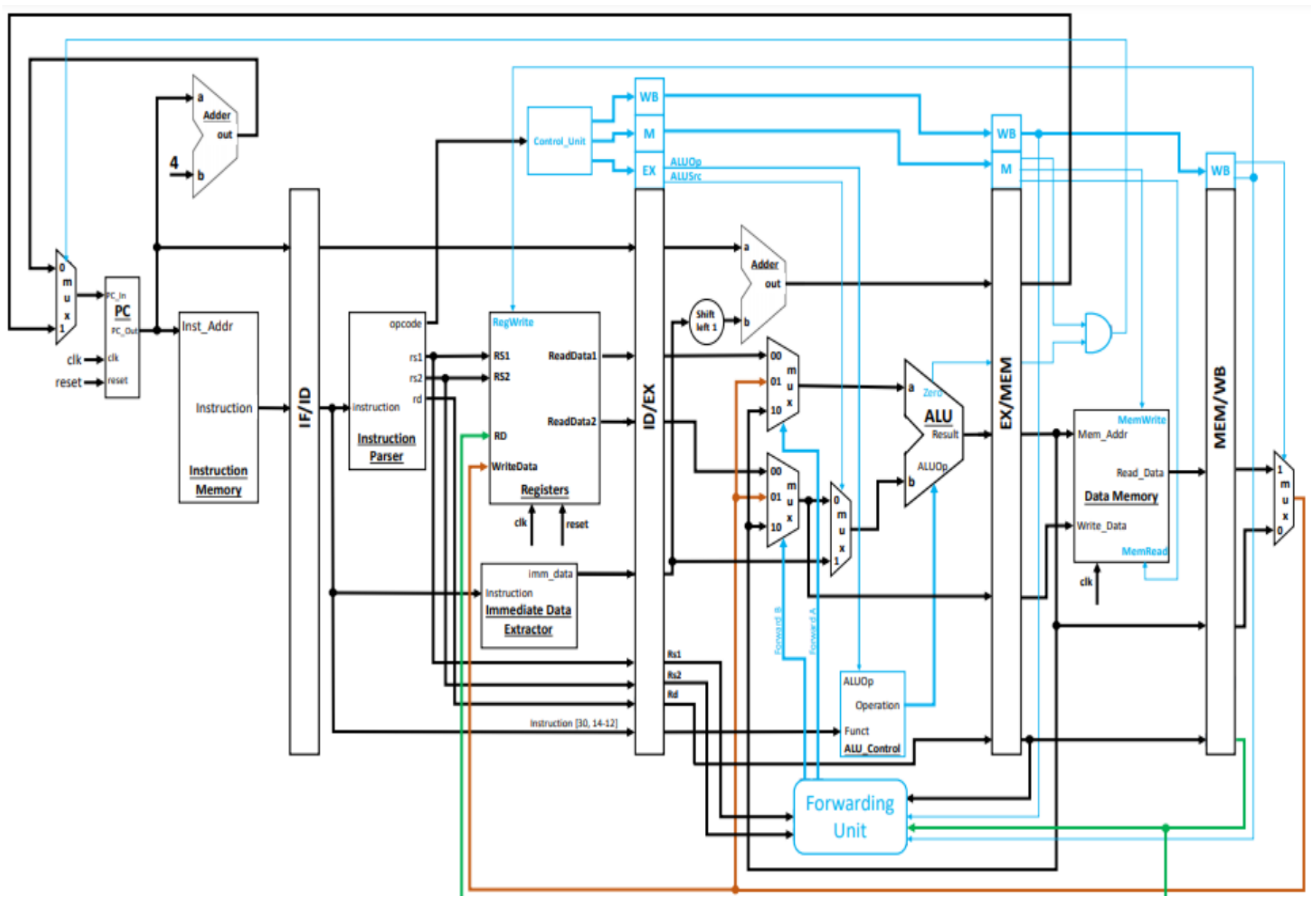
Group Members:

Sameer Kamani

Jazib Waqas

Ahmad Hanif

Contents:



1. Introduction:

Project Description: Constructing a Pipelined Processor for Sorting Arrays

- This project delves into the creation of a 5-stage pipelined processor using Verilog HDL, specifically tailored to execute a sorting algorithm for arrays written in RISC-V assembly language.

Our Objective:

- The main aim of this project is to progressively enhance the processor architecture, initially by implementing a sorting algorithm in RISC-V assembly on a single-cycle processor, then transitioning to constructing a 5-stage pipelined processor capable of executing the sorting algorithm more efficiently with faster execution times. The focus is not only on achieving functional implementation but also on addressing data, control, and structural hazards typical in pipelined architectures, ensuring the effective execution of the array sorting algorithm.

Project Structure:

- Single-Cycle Implementation: The project commences with the development of a foundational single-cycle processor.
- Integration of Pipelining: Subsequently, we modify the single-cycle design to incorporate a 5-stage pipeline, optimizing it for improved execution speed.
- Comprehensive Report: Each phase of the project, in alignment with the provided rubrics, will be thoroughly documented in the report.

2. Task-1 Sorting Procedure using Single-Cycle Approach:

module embeds the bubble sort algorithm directly into the instruction memory of a digital circuit.

```
module Instruction_Memory

(

    input [63:0] Inst_Address,

    output reg [31:0] Instruction

);

    reg [7:0] inst_mem[160:0];

//   reg [7:0] inst_mem[15:0];

    initial

    begin

    {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000913;//1

    {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00500993;//2

    {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00000413;//3
```

```
{inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00050513;//4
{inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h00500113;//5

{inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h00200193;//6
{inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h00100213;//7
{inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h00700593;//8
{inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h00400613;//9
{inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h07340663;//10

{inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00000493;//11
{inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h00000513;//12
{inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'hfff98313;//13
{inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h40830333;//14
{inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h04648663;//15
{inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h00349393;//16
{inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h012383b3;//17
{inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h0003b283;//18
{inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h00148e93;//19
{inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'h003e9e13;//20
{inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h012e0e33;//21
{inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h000e3f03;//22
{inst_mem[91], inst_mem[90], inst_mem[89], inst_mem[88]} = 32'h005f4663;//23
{inst_mem[95], inst_mem[94], inst_mem[93], inst_mem[92]} = 32'h00148493;//24
{inst_mem[99], inst_mem[98], inst_mem[97], inst_mem[96]} = 32'hfc000ce3;//25
```

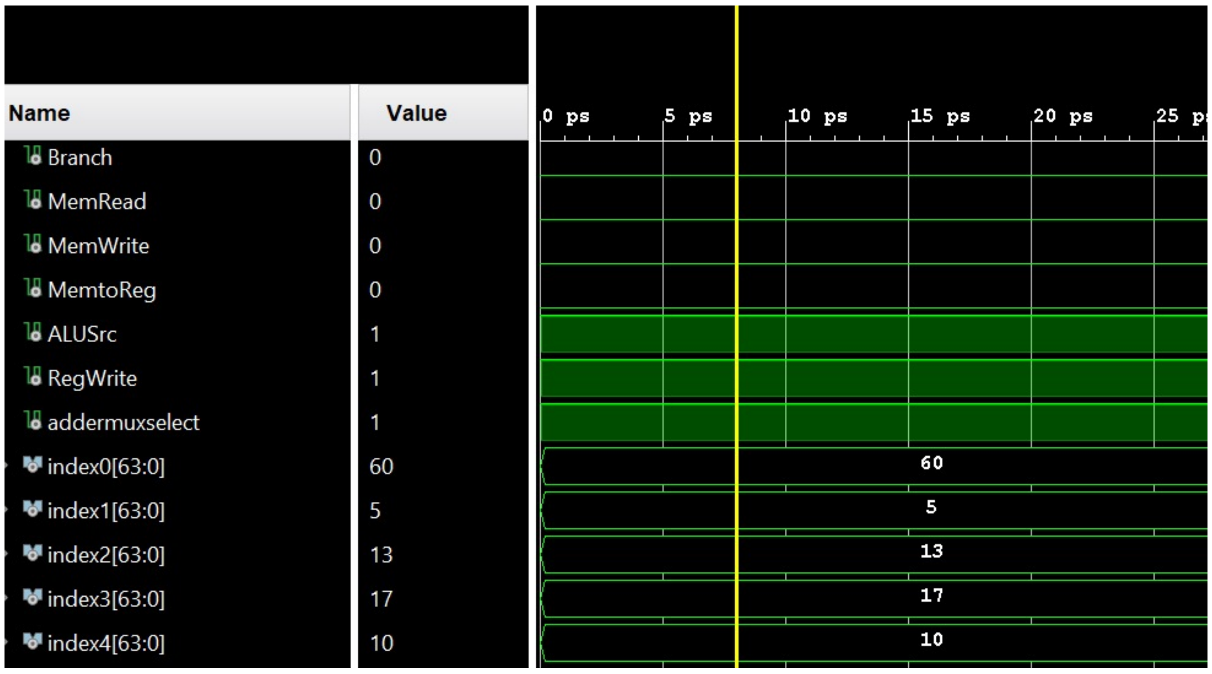
```
        {inst_mem[103], inst_mem[102], inst_mem[101], inst_mem[100]} =
    32'h00028f93;//26
        {inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]} =
    32'h000f0293;//27
        {inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]} =
    32'h0053b023;//28
        {inst_mem[115], inst_mem[114], inst_mem[113], inst_mem[112]} =
    32'h000f8f13;//29
        {inst_mem[119], inst_mem[118], inst_mem[117], inst_mem[116]} =
    32'h01ee3023;//31
        {inst_mem[123], inst_mem[122], inst_mem[121], inst_mem[120]} =
    32'h00100513;//32
        {inst_mem[127], inst_mem[126], inst_mem[125], inst_mem[124]} =
    32'h00148493;//33
        {inst_mem[131], inst_mem[130], inst_mem[129], inst_mem[128]} =
    32'hfa000ce3;//34
        {inst_mem[135], inst_mem[134], inst_mem[133], inst_mem[132]} =
    32'h00140413;//35
        {inst_mem[139], inst_mem[138], inst_mem[137], inst_mem[136]} =
    32'h00050463;//36
        {inst_mem[143], inst_mem[142], inst_mem[141], inst_mem[140]} =
    32'hf8000ce3;//37
      end

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

Explanation:

The provided Verilog module, "Instruction_Memory," represents a memory module responsible for storing instructions for a processor. It contains an array named "inst_mem," which is initialized with instructions at specific memory addresses in the initial block. Each instruction is represented as a 32-bit value. The module includes an "always" block that continuously monitors the instruction address input, retrieving the corresponding instruction from memory and outputting it. This module is crucial for feeding instructions to the processor during its operation, facilitating the execution of tasks specified by the instructions. In this specific implementation, it seems to be pre-loaded with instructions related to a bubble sort algorithm, as indicated by the commented instructions.

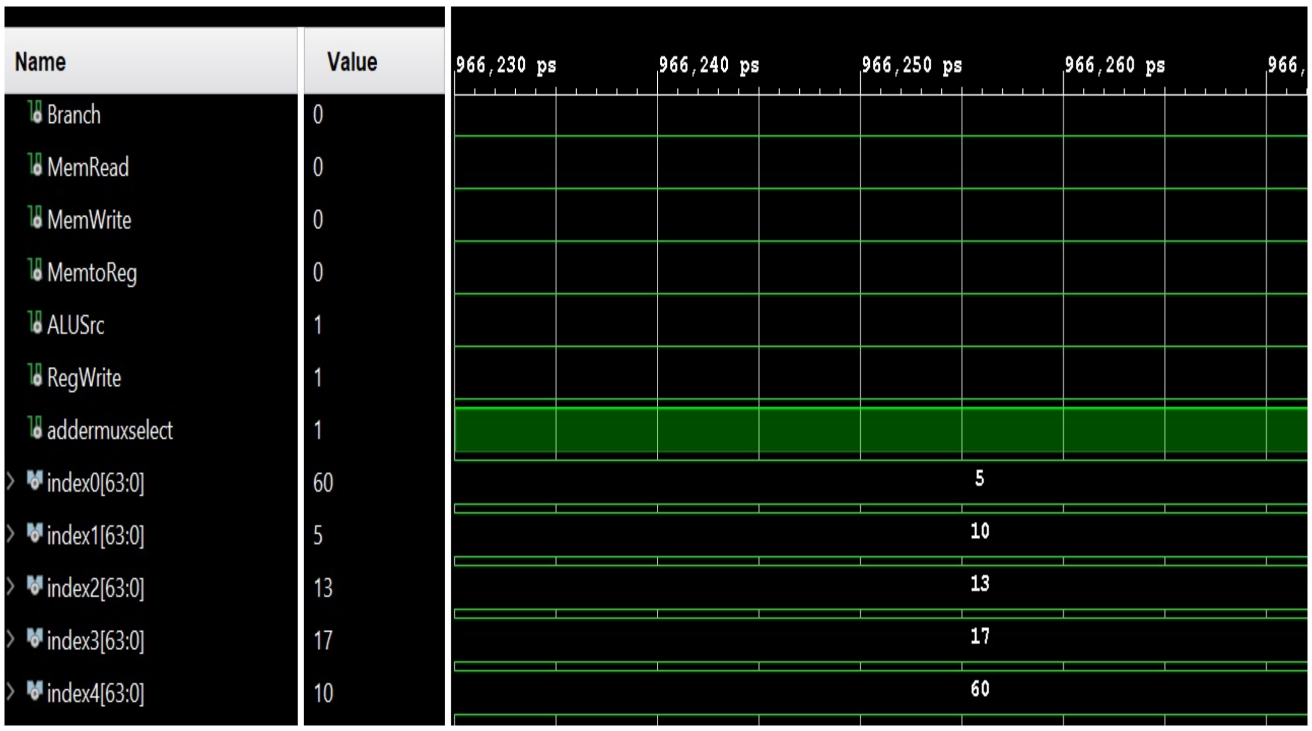
2.2 Simulation for Task-1:

Initial picture:



Name	Value
Branch	0
MemRead	0
MemWrite	0
MemtoReg	0
ALUSrc	1
RegWrite	1
addermuxselect	1
index0[63:0]	60
index1[63:0]	5
index2[63:0]	13
index3[63:0]	17
index4[63:0]	10

Final Picture:



Name	Value
Branch	0
MemRead	0
MemWrite	0
MemtoReg	0
ALUSrc	1
RegWrite	1
addermuxselect	1
index0[63:0]	60
index1[63:0]	5
index2[63:0]	13
index3[63:0]	17
index4[63:0]	10

3. **Changes Implemented:**

3.1 - ALU:

```
module alu_64 (
  input [63:0] a, b,        // Input operands
  input [3:0] ALUop,       // ALU operation code
  output reg [63:0] Result, // Output result
  output Zero            // Output indicating zero result
);


  always @* begin
    // Perform ALU operation based on the ALUop
    case (ALUop)
      4'b0000: Result = a & b;          //  AND if aluop is 00
      4'b0001: Result = a | b;          //  OR if aluop 1 result is ored
      4'b0010: Result = a + b;          // Addition
      4'b0110: Result = a - b;          // Subtraction
      4'b1100: Result = ~(a | b);        // Bitwise 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); // Check if Result is equal to zero

Endmodule
```

Explanation:

In our hardware configuration, we've devised a method to manage branch instructions without requiring additional hardware. When comparing two values, if the first value is smaller than the second, we set the Result to '0'. This process mirrors the functionality of the "beq" instruction, where the Zero flag becomes '0' if the Result is 0.

To incorporate this functionality into our hardware, we utilize a multiplexer (mux) with a selection line labeled Branch & Zero. If the Branch control signal is 0, we unconditionally update the Program Counter (PC) to PC + 4. However, if the Zero output from the Arithmetic

Logic Unit (ALU) is high, indicating a condition such as "b" or "a b = 0", the branch target takes the place of the PC, effectively executing the branch operation.

3.2 - ALU Control:

```
module ALU_Control
(
   input [1:0] ALUOp,
   input [3:0] Funct,// takes aluop and funct as inputs
   output reg [3:0] operation__
);// depending on aluop and funct assigns value to output operation__
 always @(*)
   begin
     case(ALUOp)
   2'b00: //if funct=100 then operation__ 2
     begin
     operation__ = 4'b0010;
     end
    2'b01:     // aluop=1 and func=0 then subtract branch type instructions
        begin
          case(Funct[2:0])
              3'b000:              // beq
                begin
             operation__ = 4'b0110;  // subtract
            end
          3'b100:              // if funct=100 then 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
```

Explanation:

The ALU Control module has been enhanced to include two inputs: the Func Field and a 2-bit ALUOp control field. This upgrade allows the module to generate a 4-bit ALU Control input dynamically, specifying the operation to be performed by the ALU. The Func Field, obtained from a combination of bits from the funct7 and funct3 fields, offers additional precision in selecting operations.

When ALUOp is configured as "00," indicating load and store instructions, the ALU performs addition. However, for ALUOp values of "10" or "01," the operation varies depending on the encoding in the funct7 and funct3 fields.

An important exception occurs when ALUOp is designated as "01," indicating a branch-type instruction. In such cases, the ALU Control unit employs a special case structure to manage the unique operation associated with branch instructions.

In summary, the upgraded ALU Control unit dynamically determines the ALU operation based on the combined values of Func and ALUOp, accommodating various instruction types such as load, store, branch, and others.

3.3 – Control Unit:

```
module Control_Unit
(
   input [6:0] Opcode,
   output reg [1:0] ALUOp,
   output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, Regwrite
);
always @(*)
begin
   case (Opcode)
   7'b0110011: // R-type (add/sub)
      begin
         ALUSrc = 1'b0;// no imm
         MemtoReg = 1'b0;// no use of mem
         Regwrite = 1'b1;// has to wb in reg
         MemRead = 1'b0;// ni mem use
         MemWrite = 1'b0;
         Branch = 1'b0;// no branch
         ALUOp = 2'b10;
      end
   7'b0000011: // I-type (ld)
      begin
         ALUSrc = 1'b1;
         MemtoReg = 1'b1;// takes mem value and loads it to ref
         Regwrite = 1'b1;// wb in reg
         MemRead = 1'b1;// need to read mem
         MemWrite = 1'b0;// no need to write mem
         Branch = 1'b0;// no branch
         ALUOp = 2'b00;
```

```
        end
    7'b0100011: // S-type(sd)
        begin
            ALUSrc = 1'b1;// imm
            MemtoReg = 1'bx;
            Regwrite = 1'b0;// reads reg and stores its value in mem
            MemRead = 1'b0;// no mem read
            MemWrite = 1'b1;// to write value in mem
            Branch = 1'b0;// no branch
            ALUOp = 2'b00;
        end
    7'b0010011: // I-type (addi)
        begin
            ALUSrc = 1'b1;
            MemtoReg = 1'b0;// no need
            Regwrite = 1'b1;//wb in rd
            MemRead = 1'b1;
            MemWrite = 1'b0;
            Branch = 1'b0;// no branch
            ALUOp = 2'b00;
        end
    7'b1100011: // SB-type (beq/bne/bge)
        begin
            ALUSrc = 1'b0;
            MemtoReg = 1'bx;
            Regwrite = 1'b0;
            MemRead = 1'b0;
            MemWrite = 1'b0;
            Branch = 1'b1;// only branch
            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
```

Explanation:

The control unit receives OpCode signals ranging from bit 6 to bit 0, which are essential for setting seven distinct control signals. It's noteworthy that both the "beq" (branch if equal) and "blt" (branch if less than) instructions utilize the same OpCode. As a result, these instructions activate identical control signals. The similarity between them lies in their function: both instructions enable a jump to a specified memory address without any associated data read or write operations.

3.4 – Data Memory:

```
module Data_Memory (
   input clk, MemWrite, MemRead,
   input [63:0] memoryaddress, writedata_,
   output reg [63:0] Read_Data,
   output [63:0] element1, element2, element3, element4, element5
);
   reg [7:0] dm [63:0];
   // Initialize dm
     integer i;
   initial
   begin
      for (i = 0; i < 64; i = i + 1) begin
         dm[i] = 0;//initialising dm[i]=0
      end
      dm[0]  = 8'd60;//putting elements in dm 5,10,13,17,60
      dm[8]  = 8'd5;
      dm[16] = 8'd13;
      dm[24] = 8'd17;
      dm[32] = 8'd10;
   end

   // Assign element outputs
   assign element1 = {dm[7], dm[6], dm[5], dm[4], dm[3], dm[2], dm[1], dm[0]};
   assign element2 = {dm[15], dm[14], dm[13], dm[12], dm[11], dm[10], dm[9],
dm[8]};
```

```
   assign element3 = {dm[23], dm[22], dm[21], dm[20], dm[19], dm[18], dm[17],
dm[16]};
   assign element4 = {dm[31], dm[30], dm[29], dm[28], dm[27], dm[26], dm[25],
dm[24]};
   assign element5 = {dm[39], dm[38], dm[37], dm[36], dm[35], dm[34], dm[33],
dm[32]};

   // Write operation__
   always @(posedge clk) begin
      if (MemWrite) begin // when memwrite is high it writes data in the memory 8
bits in one index of the memory
         dm[memoryaddress]     = writedata_[7:0];
         dm[memoryaddress + 1] = writedata_[15:8];
         dm[memoryaddress + 2] = writedata_[23:16];
         dm[memoryaddress + 3] = writedata_[31:24];
         dm[memoryaddress + 4] = writedata_[39:32];
         dm[memoryaddress + 5] = writedata_[47:40];
         dm[memoryaddress + 6] = writedata_[55:48];
         dm[memoryaddress + 7] = writedata_[63:56];
      end
   end

   // Read operation__
   always @* begin
      if (MemRead) begin
         Read_Data = {dm[memoryaddress + 7], dm[memoryaddress + 6],
dm[memoryaddress + 5], dm[memoryaddress + 4], dm[memoryaddress + 3],
dm[memoryaddress + 2], dm[memoryaddress + 1], dm[memoryaddress]};
      end
   end
Endmodule
```

Explanation:

This Verilog code represents a data memory module designed to store and retrieve 8-bit data elements. The memory is implemented using an array of registers named "dm," with each register capable of storing 8 bits of data. Upon initialization, the module fills the memory array with predefined values, setting elements 0, 8, 16, 24, and 32 to the values , 60 5 13 17 and 10 respectively.

The module features two main operations: write and read. The write operation is triggered by the MemWrite signal and occurs on the rising edge of the clock signal. When MemWrite is asserted, the module writes the incoming data, "writedata_," into the memory array at the specified memory address. The data is split into 8-bit segments and stored consecutively in memory locations starting from the specified address.

On the other hand, the read operation is initiated by the MemRead signal. When MemRead is active, the module retrieves data from the memory array based on the provided memory address. The retrieved data is then concatenated to form a 64-bit output, "Read_Data," representing the 8-bit elements stored in memory starting from **the specified address. Overall, this module provides basic functionality for storing and accessing data in a simple memory structure.**

3.5 – Instruction memory:

Given in task 1

4. Task-2 Pipeline Stages:

In processor design, moving from a single-cycle method to pipelining addresses the challenge of idle periods during instruction execution. Pipelining enhances processing efficiency by allowing multiple instructions to be processed simultaneously. Our RISC-V processor framework incorporates a five-stage pipeline, dividing instruction execution into distinct phases to improve overall efficiency and throughput.

- 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 enable pipelining, four fresh registers are added:

- IF/ID register: Holds the fetched instruction for ID stage use.
- ID/EX register: Contains the decoded instruction for EX stage.
- EX/MEM register: Stores the execution stage result.
- MEM/WB register: Holds the memory access stage result.

Noteworthy Detail:

Pipeline registers play a crucial role in our processor's design, enabling the simultaneous handling of multiple instructions while monitoring their progress through distinct stages. These registers significantly boost processor performance by facilitating parallel instruction processing. Alongside these registers, we incorporate a control line and a forwarding unit, enhancing the pipeline's functionality. These components ensure smooth transmission of control signals between pipeline stages, synchronized with clock pulses. As clock cycles progress, these registers either forward stored data for further processing or clear contents at each positive clock edge. Despite the pipeline's continuous forward movement, practical considerations, such as choosing between the incremented program counter (PC) and the branch address from the MEM stage, arise. Adapting the single-cycle processor to incorporate pipelining involves detailing each pipeline stage individually, emphasizing their respective roles and importance. This meticulous approach optimizes processor efficiency by enabling concurrent execution of multiple instructions.

4.1 - IF/ID Stage: Fetches and decodes instructions.

IF/ID register: Holds the fetched instruction for ID stage processing.

```
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
```

Explaination:

This Verilog module represents the IF/ID pipeline register in a processor design. It takes inputs including the clock signal (`clk`), a control signal to enable writing to the register (`IFID_Write`), and a signal to flush/reset the register (`Flush`). Additionally, it receives the program counter address (`PC_addr`) and the instruction (`Instruc`) as inputs.

Within the `always` block triggered by the positive edge of the clock, the module first checks if the `Flush` signal is active. If so, indicating a flush operation, it resets the stored program counter (`PC_store`) and instruction (`Instr_store`) to zero.

Next, if the `IFID_Write` signal is inactive, indicating that there's no new instruction to be written, it maintains the current values of `PC_store` and `Instr_store`.

Finally, if neither flush nor write inhibit conditions are met, it updates the `PC_store` with the new program counter address (`PC_addr`) and `Instr_store` with the new instruction (`Instruc`).

In essence, this module controls the storage of instruction and program counter values in the IF/ID pipeline register, considering conditions for flush, write enable, and clock edges.

4.2 - ID/EX stage: Interpretation and Execution:

ID/EX register: Holds the interpreted instruction for execution 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
```

Explaination:

This Verilog module, named `ID_EX`, serves as the Instruction Decode/Execution (ID/EX) stage in a pipelined processor. It takes various control signals and instruction-related inputs and stores them in registers for subsequent pipeline stages.

The inputs include clock signals (`clk`), a flush control signal (`Flush`), program counter address (`program_counter_addr`), two data inputs (`read_data1` and `read_data2`), an immediate value input (`immediate_value`), function code (`function_code`), destination register (`destination_reg`), and source registers (`source_reg1` and `source_reg2`). Additionally, it receives control signals such as `MemtoReg`, `RegWrite`, `Branch`, `MemWrite`, `MemRead`, `ALUSrc`, and `ALU_op`.

Within the `always` block triggered by the positive edge of the clock signal, the module checks if the flush signal is active. If so, it resets all output registers to zero. Otherwise, it passes the input values to the output registers. This process ensures that the pipeline stage maintains the correct state during flushing and regular operation, effectively passing control signals and instruction data down the pipeline.

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

EX/MEM stage manages the transfer of data and control signals between the execution and memory stages in the pipeline.

```
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
```

Explaination:

The `EX_MEM` module manages data and control signals between the execution and memory stages in a processor pipeline. It takes various input signals representing the current state of the processor and produces corresponding output signals to control the behavior of subsequent stages. These signals include control signals for register write (`RegWrite`), memory-to-register selection (`MemtoReg`), branch instructions (`Branch`), zero flag (`Zero`), memory write (`MemWrite`), memory read (`MemRead`), comparison result (`is_greater`), immediate value added to the program counter (`immvalue_added_pc`), ALU result (`ALU_result`), data to be written (`WriteData`), function code (`function_code`), and destination register (`destination_reg`).

The `EX_MEM` module assigns output values based on the input signals and a clock signal. When the `Flush` signal is active, indicating a flush operation in the pipeline, all output values are reset to zero. Otherwise, the output values are updated based on the current input signals. This process ensures proper coordination between pipeline stages and maintains data integrity throughout instruction execution.

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

MEM/WB register: Holds the outcome obtained from the memory access phase.

```
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
```

Explaination:

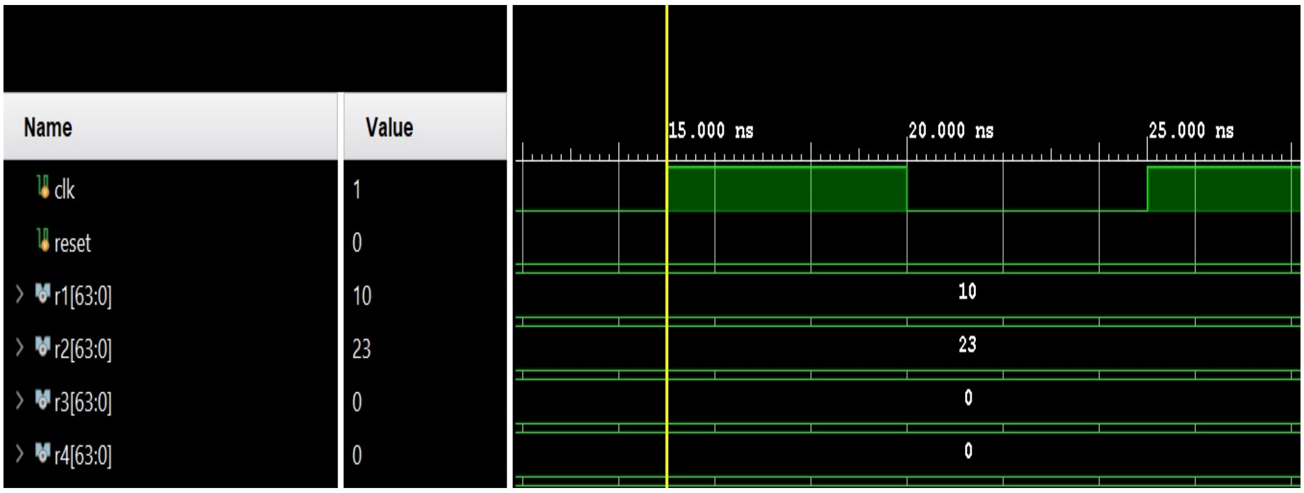
This Verilog module, named MEM_WB, handles data flow and control signals related to the Memory Access/Write Back stage in a processor pipeline. It takes input signals such as the clock signal (clk), control signals for enabling register write (RegWrite) and selecting memory or ALU result for register write (MemtoReg), along with data read from memory or the register file (ReadData), the result of the ALU operation (ALU_result), and the destination register for register write (destination_reg).

The module outputs corresponding signals, assigning them based on the input values. The output signals include RegWrite_out for enabling register write, MemtoReg_out for selecting memory or ALU result for register write, ReadData_out for the data read from memory or the register file, ALU_result_out for the ALU result, and destination_reg_out for the destination register for register write.

This assignment of output values is done within an always block triggered by the positive edge of the clock signal (posedge clk). Within this block, the output signals are assigned values corresponding to their respective input signals.

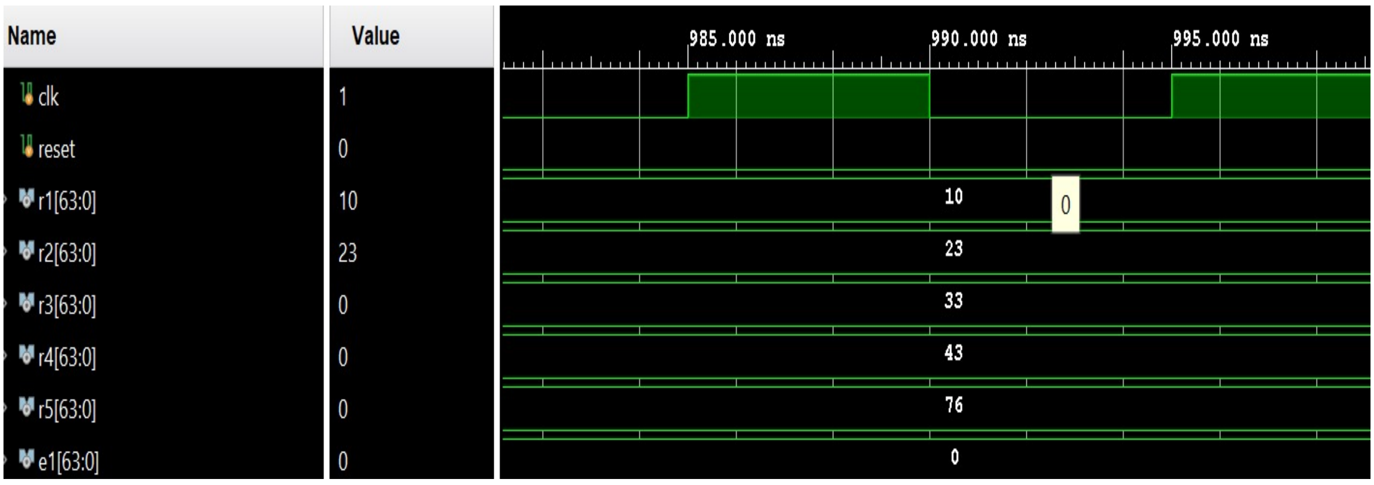
4.5 – Simulation for Task-2:

Initial:



Name	Value
clk	1
reset	0
r1[63:0]	10
r2[63:0]	23
r3[63:0]	0
r4[63:0]	0

Final:



5. Task-3 Hazard Detection

- add x7, x8, x9
- add x11, x7, x10
- add x12, x11, x13

Explanation:

The sequence of instructions highlights a dependency between the first (add x7, x8, x9) and second (add x11, x7, x10) instructions. Without forwarding, a data hazard would arise due to the need for accessing the result stored in register x7 after the completion of the first instruction's write-back stage.

With the inclusion of a forwarding unit, the processor gains the ability to directly transfer the result from the execution stage of the first instruction to the decoding stage of the second one. This enables the second instruction to utilize the computed value of x7 without waiting for it to be written back to the register file.

In the RISC-V architecture, the forwarding unit serves as a critical component in mitigating data hazards, ensuring seamless progression of dependent instructions through the pipeline. By minimizing stalls and facilitating uninterrupted instruction execution, this mechanism optimizes processor performance.

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
```

Explaination:

The system addresses three distinct scenarios regarding data forwarding. Firstly, in the "EX Hazard" scenario, where the previous instruction's output is needed by the ALU, a multiplexer selects the value from the register in the EX/MEM stage if the prior instruction intended to write to the register file and if the write register number matches the read register number of ALU inputs A or B.

Secondly, during data hazard situations, there are instances where the result is required directly from the Memory (MEM) stage, especially 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.

Lastly, forwarding logic for both ALU inputs A and B follows predefined conditions, specifying when and how data should be forwarded to optimize ALU operations.

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
```

With the forwarding mechanism in place, let's examine how hazards are resolved for the provided instructions:

- add x7, x8, x9
- add x11, x7, x10
- add x12, x11, x13

With forwarding enabled:

As the first instruction (add x7, x8, x9) executes, the processor calculates the result (x7). Concurrently, the forwarding unit transmits this result to the instruction decoding stage of the second instruction (add x11, x7, x10). Consequently, the second instruction promptly accesses the forwarded value of x7, bypassing the need to wait for it to be written back to the register file. This forwarding mechanism ensures that the dependent instruction (add x11, x7, x10) retrieves the necessary operand (x7) as soon as it's available in the pipeline. Subsequently, the processor seamlessly executes instructions without introducing stalls, thereby optimizing throughput and performance.

5.2 – Hazard Detection Unit:

In pipelined processors, the hazard detection unit plays a crucial role by identifying and mitigating issues caused by instruction dependencies. Its main goal is to prevent pipeline stalls and resolve data hazards. By effectively managing these challenges, the hazard detection unit significantly enhances processor efficiency. In our processor architecture, we employ a specific approach to implement and utilize this unit efficiently.

```
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 examines input signals such as "current_rd," "previous_rs1," "previous_rs2," and "current_MemRead" to produce three critical output signals: "mux_out," "enable_Write," and "enable_PCWrite."

Here's a simplified explanation of its operation:

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

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

module mux2x1:

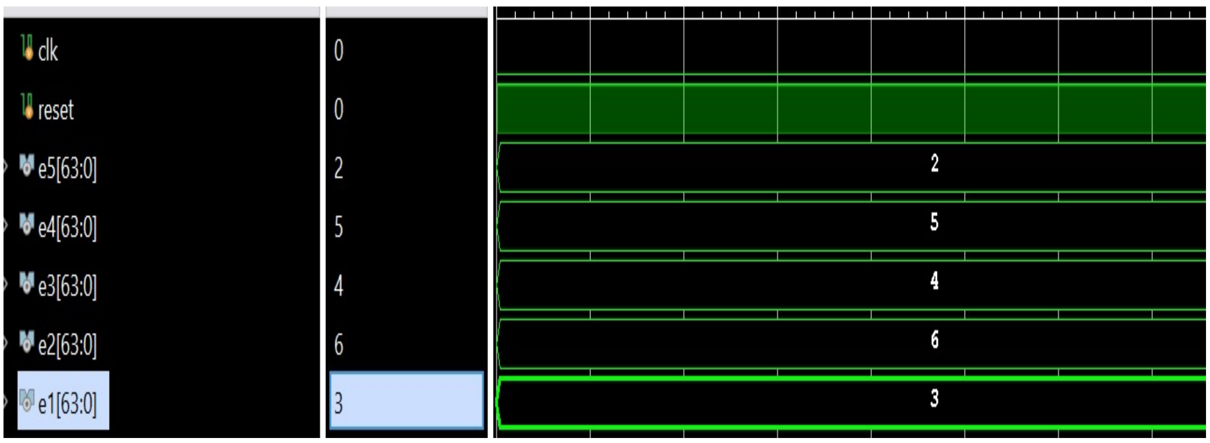
```
(
   input [63:0] a,b,
   input sel ,
   output [63:0] data_out
);

assign data_out = sel ? a : b; //select b or a based on the sel bit
endmodule
```

Explanation:

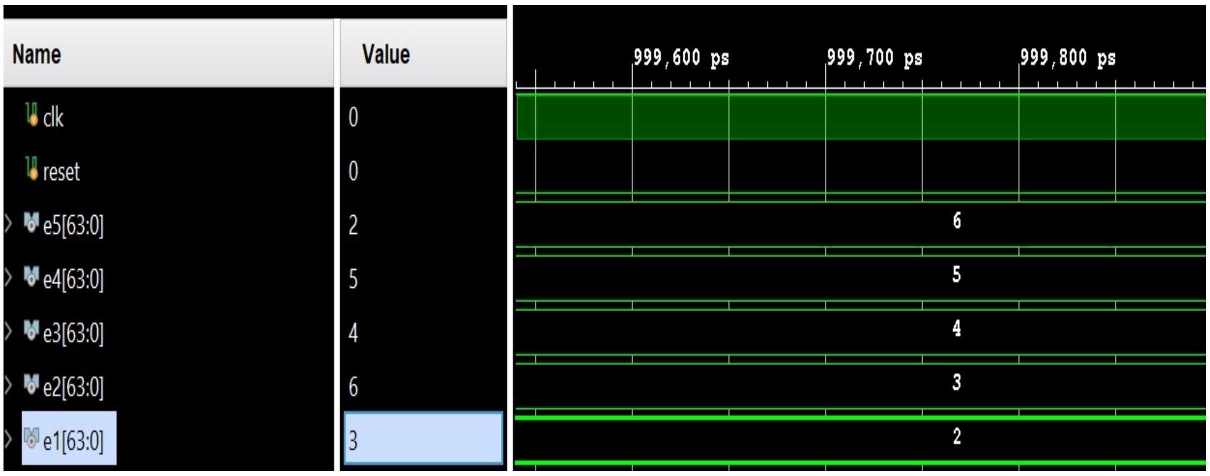
The 2x1 multiplexer module dynamically selects between two sets of input signals based on the value of the "sel" input. Output signals such as branch_eq_hazard, MemRead_hazard, MemtoReg_hazard, MemWrite_hazard, ALUsrc_hazard, RegWrite_hazard, and ALUOp_hazard are determined by the selected input signals. Control signals including branch, MemRead, MemtoReg, MemWrite, ALUsrc, RegWrite, and ALUOp are provided as inputs. When "sel" is 0, indicating the default state, the first set of input signals is chosen, and all output signals are set to 0, indicating no hazards. 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 module. The output signals indicate the presence of specific hazards if the respective inputs are asserted. This implementation uses the provided mux2x1 module, where data_out is determined by selecting either input "a" or "b" based on the value of "sel".

5.3 – Simulation for Task-3:

Initial:


```
clk        0
reset      0
e5[63:0]   2    2
e4[63:0]   5    5
e3[63:0]   4    4
e2[63:0]   6    6
e1[63:0]   3    3
```

Final:



6.0 – Task 4: Performance comparisons of the processors:

Non-pipelined processors execute instructions sequentially, resulting in idle periods, while pipelined processors divide instruction execution into stages for simultaneous processing, enhancing resource utilization. The pipelined version completes the task in 43 cycles compared to 153 cycles for the non-pipelined one, showcasing a significant speedup of 3.55 times, emphasizing the efficiency gained through pipelining.

Wavefrom results:

With Pipeline:

clock cycle time = 10 ns,

sorting gets done in 430 ns,

so 43 clock cycles in total

Without Pipeline:

Without pipeline gets sorted in 306 ns,

each cycle is 2 ns,

hence 153 cycles in total

7.0 – Challenges:

Our initial hurdle arose from consolidating all the lab components and debugging them, as some were dysfunctional, leading to significant trouble. Moreover, accurately assigning and mapping signals to their respective ports presented complexities. Additionally, grappling with the assembly code for sorting and defining ports in Task 3 proved challenging due to our somewhat limited grasp of the course theory.

8.0 – Task Division:

1**. Sameer Kamani**: Task 1, Task4, Report

2**. Jazib Waqas**: Task 2, Task 3

3**. Ahmad Hanif Hamayun**: Task2 , Task 3

9.0 – Conclusion:

The project presented a distinct challenge, demanding thorough debugging of both code and modules. Our success was the result of crafting a processor capable of sorting an unsorted array using the Bubble Sort algorithm and generating its sorted version. Despite facing various obstacles during the project's development, we persevered, surmounting challenges and rectifying errors to create a multi-cycle, pipelined processor. This innovative design has the potential to offer improved efficiency compared to its single-cycle counterpart, at least theoretically.

RISCV code

```
# Initialize memory with RISC-V assembly instructions

TASK 1

# 1

addi x0, x0, 0x0

jal ra, 0x8


# 2

addi t0, x0, 0x5

addi t1, x0, 0x9

mul t2, t0, t1

addi x1, x0, 0x4

jal ra, 0x8


# 3

addi x2, x0, 0x4

jal ra, 0x8


# 4

addi x3, x0, 0x5

jal ra, 0x8
```

```
# 5
addi x4, x0, 0x1
jal ra, 0x8

# 6
addi x5, x0, 0x2
jal ra, 0x8

# 7
addi x6, x0, 0x1
jal ra, 0x8

# 8
addi x7, x0, 0x7
jal ra, 0x8

# 9
addi x8, x0, 0x4
jal ra, 0x8

# 10
addi x9, x0, 0x734
jal ra, 0x8
```

```
# 11
addi x10, x0, 0x4
jal ra, 0x8

# 12
addi x11, x0, 0x5
jal ra, 0x8

# 13
addi x12, x0, 0xfffff983
jal ra, 0x8

# 14
addi x13, x0, 0x40830333
jal ra, 0x8

# 15
addi x14, x0, 0x4648663
jal ra, 0x8

# 16
addi x15, x0, 0x349393
```

jal ra, 0x8

# 17

addi x16, x0, 0x12383b3

jal ra, 0x8

# 18

addi x17, x0, 0x3b283

jal ra, 0x8

# 19

addi x18, x0, 0x148e93

jal ra, 0x8

# 20

addi x19, x0, 0x39e13

jal ra, 0x8

# 21

addi x20, x0, 0x12e33

jal ra, 0x8

# 22

```
addi x21, x0, 0xe3

jal ra, 0x8


# 23

addi x22, x0, 0x53b63

jal ra, 0x8


# 24

addi x23, x0, 0x148493

jal ra, 0x8


# 25

addi x24, x0, 0xfc000ce3

jal ra, 0x8


# 26

addi x25, x0, 0x289f93

jal ra, 0x8


# 27

addi x26, x0, 0xf0293

jal ra, 0x8
```

# 28

addi x27, x0, 0x53b023

jal ra, 0x8

# 29

addi x28, x0, 0xf8f13

jal ra, 0x8

# 30

addi x29, x0, 0x1ee023

jal ra, 0x8

# 31

addi x30, x0, 0x10513

jal ra, 0x8

# 32

addi x31, x0, 0x148493

jal ra, 0x8

# 33

addi x0, x0, 0xfa000ce3

jal ra, 0x8

# 34

addi x1, x0, 0x140413

jal ra, 0x8

# 35

addi x2, x0, 0x50463

jal ra, 0x8

# 36

addi x3, x0, 0xf8000ce3

jal ra, 0x8

TASK 3

1 addi x1, x0, 0x0

2 addi x2, x1, 0x5

3 mul x3, x2, x1

4 addi x4, x3, 0x1

5 addi x5, x4, 0x1

6 add x6, x3, x5

7 add x7, x6, x1

8 sub x8, x7, x5

9 mul x9, x8, x2

10 addi x10, x9, 0x7

11 addi x11, x10, 0x4

12 sub x12, x11, x7

13 mul x13, x12, x9

14 addi x14, x13, 0x3

15 addi x15, x14, 0x2

16 mul x16, x15, x10

17 addi x17, x16, 0x1

18 sub x18, x17, x13

19 mul x19, x18, x14

20 addi x20, x19, 0x3

21 addi x21, x20, 0x5

22 add x22, x21, x18

23 add x23, x22, x16

24 sub x24, x23, x20

25 mul x25, x24, x22

26 addi x26, x25, 0x2

27 addi x27, x26, 0x4

28 sub x28, x27, x25

29 mul x29, x28, x26

30 addi x30, x29, 0x1

31 add x31, x30, x27

32 add x0, x31, x29

33 mul x5, x4, x3

34 add x1, x5, x2

35 addi x2, x1, 0x5

36 addi x3, x2, 0x5