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

Implementing Sorting on a Pipelined Processor

Azkaa Nasir, Fatima Faisal, Afifah Uzair

26th April 2024

Introduction:

- To build a 5-stage pipelined processor capable of executing any one array sorting algorithm. The aim is to upgrade the current single-cycle processor into a highly optimized pipelined version by incorporating specific instructions, such as bgt or blt, to facilitate seamless value swapping during sorting. This will require adaptations to various modules to enhance the processor's capabilities and performance, ultimately establishing new benchmarks in processor design.

Methodology

Task 1: In Task 1 we implemented the bubble sort algorithm on a single cycle processor

- We used the following modules in our entire code:

  ALU Control
  ALU_64 Bit
  Data Memory
  Instruction Memory
  Adder
  Mux2x1
  Instruction Parser
  Data extractor
  Program counter
  Control Unit
  Register File

- **Changes made in the code:**

  Mostly we took our old lab codes whichever ones worked best but we made changes to 4 modules especially Data Memory, Instruction memory, ALU 64 bit and ALU Control

Data Memory: we added 5 different output ports which help us in showing the working and output of our bubble sort algorithm. Here is the code:

```
module Data_Memory (
    input clk, MemWrite, MemRead,
    input [63:0] Mem_Address, Write_Data,
    output reg [63:0] Read_Data,
    output [63:0] element1, element2, element3, element4, element5
);
    reg [7:0] DataMemory [512:0];

    // Initialize DataMemory
      integer i;
    initial
    begin

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

    // Assign element outputs
    assign element1 = DataMemory[256];
          // assign element1 = {DataMemory[7], DataMemory[6], DataMemory[5],
DataMemory[4], DataMemory[3], DataMemory[2], DataMemory[1], DataMemory[0]};
    assign element2 = DataMemory[264];
    assign element3 = DataMemory[272];
```

```
    assign element4 = DataMemory[280];
    assign element5 = DataMemory[288];

          //{DataMemory[15], DataMemory[14], DataMemory[13], DataMemory[12], DataMemory[11], DataMemory[10], DataMemory[9], DataMemory[8]};
        //assign element3 = {DataMemory[23], DataMemory[22], DataMemory[21], DataMemory[20], DataMemory[19], DataMemory[18], DataMemory[17], DataMemory[16]};
        //assign element4 = {DataMemory[31], DataMemory[30], DataMemory[29], DataMemory[28], DataMemory[27], DataMemory[26], DataMemory[25], DataMemory[24]};
        //assign element5 = {DataMemory[39], DataMemory[38], DataMemory[37], DataMemory[36], DataMemory[35], DataMemory[34], DataMemory[33], DataMemory[32]};

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

    // Read operation
    always @* begin
        if (MemRead) begin
```

```
            Read_Data = {DataMemory[Mem_Address + 7], DataMemory[Mem_Address +
        6], DataMemory[Mem_Address + 5], DataMemory[Mem_Address + 4],
        DataMemory[Mem_Address + 3], DataMemory[Mem_Address + 2],
        DataMemory[Mem_Address + 1], DataMemory[Mem_Address]};
          end
       end
    endmodule
```

Instruction Memory:

```
     module Instruction_Memory
    (
     input [63:0] Inst_Address,
     output reg [31:0] Instruction
    );
     reg [7:0] inst_mem [147:0];

    initial
     begin
            inst_mem[0]  = 8'h93;
            inst_mem[1]  = 8'h05;
            inst_mem[2]  = 8'h60;
            inst_mem[3]  = 8'h00;
            inst_mem[4]  = 8'h93;
            inst_mem[5]  = 8'h0E;
            inst_mem[6]  = 8'h60;
            inst_mem[7]  = 8'h00;
            inst_mem[8]  = 8'h13;
            inst_mem[9]  = 8'h0F;
            inst_mem[10] = 8'h00;
            inst_mem[11] = 8'h00;
            inst_mem[12] = 8'h13;
            inst_mem[13] = 8'h0F;
```

```
inst_mem[14] = 8'h00;
inst_mem[15] = 8'h00;
inst_mem[16] = 8'h13;
inst_mem[17] = 8'h0E;
inst_mem[18] = 8'h60;
inst_mem[19] = 8'h00;
inst_mem[20] = 8'h23;
inst_mem[21] = 8'h20;
inst_mem[22] = 8'hBF;
inst_mem[23] = 8'h10;
inst_mem[24] = 8'h93;
inst_mem[25] = 8'h8F;
inst_mem[26] = 8'h1F;
inst_mem[27] = 8'h00;
inst_mem[28] = 8'h13;
inst_mem[29] = 8'h0F;
inst_mem[30] = 8'h8F;
inst_mem[31] = 8'h00;
inst_mem[32] = 8'h93;
inst_mem[33] = 8'h85;
inst_mem[34] = 8'hF5;
inst_mem[35] = 8'hFF;
inst_mem[36] = 8'h63;
inst_mem[37] = 8'h04;
inst_mem[38] = 8'hFE;
inst_mem[39] = 8'h01;
inst_mem[40] = 8'hE3;
inst_mem[41] = 8'h06;
inst_mem[42] = 8'h00;
inst_mem[43] = 8'hFE;
inst_mem[44] = 8'h13;
inst_mem[45] = 8'h0F;
```

```
inst_mem[46] = 8'h00;
inst_mem[47] = 8'h00;
inst_mem[48] = 8'h93;
inst_mem[49] = 8'h0F;
inst_mem[50] = 8'h0F;
inst_mem[51] = 8'h00;
inst_mem[52] = 8'h93;
inst_mem[53] = 8'h0E;
inst_mem[54] = 8'h00;
inst_mem[55] = 8'h00;
inst_mem[56] = 8'h93;
inst_mem[57] = 8'h05;
inst_mem[58] = 8'h60;
inst_mem[59] = 8'h00;
inst_mem[60] = 8'h63;
inst_mem[61] = 8'h8C;
inst_mem[62] = 8'hE5;
inst_mem[63] = 8'h05;
inst_mem[64] = 8'h33;
inst_mem[65] = 8'h85;
inst_mem[66] = 8'h0E;
inst_mem[67] = 8'h00;
inst_mem[68] = 8'h93;
inst_mem[69] = 8'h0F;
inst_mem[70] = 8'h1F;
inst_mem[71] = 8'h00;
inst_mem[72] = 8'h13;
inst_mem[73] = 8'h8E;
inst_mem[74] = 8'h8E;
inst_mem[75] = 8'h00;
inst_mem[76] = 8'h63;
inst_mem[77] = 8'h88;
```

```
inst_mem[78] = 8'hBF;
inst_mem[79] = 8'h02;
inst_mem[80] = 8'h83;
inst_mem[81] = 8'h27;
inst_mem[82] = 8'h0E;
inst_mem[83] = 8'h10;
inst_mem[84] = 8'h03;
inst_mem[85] = 8'h28;
inst_mem[86] = 8'h05;
inst_mem[87] = 8'h10;
inst_mem[88] = 8'h63;
inst_mem[89] = 8'hCE;
inst_mem[90] = 8'h07;
inst_mem[91] = 8'h01;
inst_mem[92] = 8'h93;
inst_mem[93] = 8'h8F;
inst_mem[94] = 8'h1F;
inst_mem[95] = 8'h00;
inst_mem[96] = 8'h13;
inst_mem[97] = 8'h0E;
inst_mem[98] = 8'h8E;
inst_mem[99] = 8'h00;
inst_mem[100] = 8'hE3;
inst_mem[101] = 8'h04;
inst_mem[102] = 8'h00;
inst_mem[103] = 8'hFC;
inst_mem[104] = 8'h13;
inst_mem[105] = 8'h0F;
inst_mem[106] = 8'h1F;
inst_mem[107] = 8'h00;
inst_mem[108] = 8'h13;
inst_mem[109] = 8'h0E;
```

```
inst_mem[110] = 8'h8E;
inst_mem[111] = 8'h00;
inst_mem[112] = 8'hE3;
inst_mem[113] = 8'h06;
inst_mem[114] = 8'h00;
inst_mem[115] = 8'hFC;
inst_mem[116] = 8'h13;
inst_mem[117] = 8'h05;
inst_mem[118] = 8'h0E;
inst_mem[119] = 8'h00;
inst_mem[120] = 8'hE3;
inst_mem[121] = 8'h02;
inst_mem[122] = 8'h00;
inst_mem[123] = 8'hFE;
inst_mem[124] = 8'h83;
inst_mem[125] = 8'h26;
inst_mem[126] = 8'h05;
inst_mem[127] = 8'h10;
inst_mem[128] = 8'h03;
inst_mem[129] = 8'hA7;
inst_mem[130] = 8'h0E;
inst_mem[131] = 8'h10;
inst_mem[132] = 8'h23;
inst_mem[133] = 8'hA0;
inst_mem[134] = 8'hDE;
inst_mem[135] = 8'h10;
inst_mem[136] = 8'h23;
inst_mem[137] = 8'h20;
inst_mem[138] = 8'hE5;
inst_mem[139] = 8'h10;
inst_mem[140] = 8'h93;
inst_mem[141] = 8'h8E;
```

```
        inst_mem[142] = 8'h8E;
        inst_mem[143] = 8'h00;
        inst_mem[144] = 8'hE3;
        inst_mem[145] = 8'h06;
        inst_mem[146] = 8'h00;
        inst_mem[147] = 8'hFC;
 end

 always @(Inst_Address)
 begin

Instruction={inst_mem[Inst_Address+3],inst_mem[Inst_Address+2],inst_mem[Inst_Addr
ess+1],inst_mem[Inst_Address]};
 end
endmodule
module Instruction_Parser(
   input [31:0] instruction,
   output [6:0] opcode, funct7,
   output [4:0] rd , rs1 , rs2,
   output [2:0] funct3

);

assign opcode = instruction[6:0];
assign rd = instruction[11:7];
assign funct3 = instruction[14:12];
assign rs1 = instruction[19:15];
assign rs2 = instruction[24:20];
assign funct7 = instruction[31:25];

endmodule
```

ALU CONTROL: we added an extra operation for the blt instruction which was used in our code:

```
module ALU_Control
(
    input [1:0] ALUOp,
    input [3:0] Funct,
    output reg [3:0] Operation
);
    always @(*) begin
    if (ALUOp == 2'b00) begin
        Operation = 4'b0010; // Same operation for '2'b00'
    end
    else if (ALUOp == 2'b01) begin // Branch type instructions
        if (Funct[2:0] == 3'b000) begin // beq
            Operation = 4'b0110; // subtract
        end
        else if (Funct[2:0] == 3'b100) begin // blt
            Operation = 4'b0100; // less than operation
        end
    end
    else if (ALUOp == 2'b10) begin
        if (Funct == 4'b0000) begin Operation = 4'b0010; // Operation for '4'b0000'
        end
        else if (Funct == 4'b1000) begin Operation = 4'b0110; // Operation for '4'b1000'
        end
        else if (Funct == 4'b0111) begin Operation = 4'b0000; // Operation for '4'b0111'
        end
        else if (Funct == 4'b0110) begin Operation = 4'b0001; // Operation for '4'b0110'
        end
    end
```

```
        end
        endmodule
```

ALU 64 Bit: we added an extra operation for the blt instruction which was used in our code:

```
module ALU_64Bit (
  input [63:0] a, b,         // Input operands
  input [3:0] ALUop,         // ALU operation code
  output reg [63:0] Result, // Output result
  output reg Zero           // Output indicating zero result
);
    always@(*) begin
        if (ALUop == 4'b0000) begin Result = a&b; end //bitwise and
        else if (ALUop == 4'b0001) begin Result = a|b; end //bitwise or
        else if (ALUop == 4'b0010) begin Result = a+b; end //addition
        else if (ALUop == 4'b0110) begin Result = a-b; end // subtraction
        else if (ALUop == 4'b1100) begin Result = ~(a|b); end //bitwise NOR
        else if (ALUop == 4'b0100) begin Result = (a < b) ? 0 : 1; end // blt
        else begin assign Zero = (Result == 0);end // Check if Result is equal to zero
    end
endmodule
```

Results

The output ports integrated within the data memory function helps us to assign random values which can be seen being sorted in the waveform.

Task 2: In Task 2 we introduced four new modules to implement a pipelines single cycle processor. We tested three add instructions to test out the working. We used the following modules in our entire code:

IF_ID
ID_EX
EX_MEM
MEM_WB

- **Changes made in the code:**
    - Besides the four new modules, while most of our code remained the same we implemented a few changes in our instruction module and data memory to test our add instructions.

IF_ID: We start by defining our input signals, clk, IFID_Write, Flush. The clk signal ensures the synchronous execution of the module whereas the IFID_Write signal is used to determine whether new values should be written and the program counter and the instruction count should be updated. The flush signal is used to determine whether the calculated stored values should be reset or flushed. Then we define two input registers called PC_addr and Instruc each of 64 and 32 bits respectively. Moreover, two out put registers called PC_Store and Instr_store are defined each of 64 and 32 bits. These registers are used to pass on the correct value of the instruction address calculated during the instruction fetch stage.

```
module IF_ID(
```

```
//Declare the input signals
    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
    //If the Flush signal is flush the calculated value of the PC address.
    if (Flush) begin
            PC_store <= 0;
        Instr_store <= 0;
    end else if (!IFID_Write) begin
        // If the IFID_Write signal is 0, do not change the value of the calculated pc address.
        PC_store <= PC_store;
        Instr_store <= Instr_store;
    end else begin
        // If the IFID_WRITE signal is 1, overwrite the new calculated address.
        PC_store <= PC_addr;
        Instr_store <= Instruc;
    end
end

endmodule
```

ID_EX: This module primarily acts as a latch to hold the data and control signals during the ID/EX stage of the pipeline. It passes the data and control signals from the ID stage to the EX stage in a single-cycle pipeline processor. In case of a flush, it ensures that the pipeline registers are cleared to maintain correctness in the pipeline operation. It doesn't perform any computation or manipulation of the data itself; it merely passes the data and control signals through to the next stage. We start by defining all the input signals, clk for synchronization, and the flush signal to

clear the pipeline registers in case of a flush. We define the following input registers:

- **program_counter_addr:** Program counter address of the instruction.
- **read_data1 and read_data2:** Data fetched from registers for operand reading.
- **immediate_value:** Immediate value extracted from the instruction.
- **function_code:** Function code extracted from the instruction.
- **destination_reg, source_reg1, source_reg2:** Register numbers for destination and source operands.

  Following control signals are also defined to determine the transfer of data from one stage to another:

- **MemtoReg, RegWrite:** Memory-to-register and register write control signals.
- **Branch:** Branch control signal.
- **MemWrite, MemRead:** Memory write and read control signals.
- **ALUSrc, ALU_op:** ALU source and operation control signals.

  Following output are also defined for the transfer of data from one pipeline stage to the next:

- **program_counter_addr_out:** Stored program counter address.
- **read_data1_out, read_data2_out:** Stored data fetched from registers.
- **immediate_value_out, function_code_out:** Stored immediate value and function code.
- **destination_reg_out, source_reg1_out, source_reg2_out:** Stored register numbers.

More control signals are defined to be used in the next stage:

MemtoReg_out, RegWrite_out, Branch_out, MemWrite_out, MemRead_out, ALUSrc_out, ALU_op_out.

On each rising edge of the clock (clk), the module checks if the Flush signal is active.

- If Flush is active, all output registers are reset to zero.
- If Flush is inactive, input values are passed to output registers to propagate them to the next pipeline stage.

```
module ID_EX(
    input       clk,                        // Clock signal
    input       Flush,                      // Flush signal
    input [63:0] program_counter_addr,    // Program counter address input
    input [63:0] read_data1,              // Input for data q
    input [63:0] read_data2,              // input for data 2
    input [63:0] immediate_value,         // Immediate value
    input [3:0]  function_code,           // Function code
    input [4:0]  destination_reg,         // Destination register
    input [4:0]  source_reg1,             // Source register 1
    input [4:0]  source_reg2,             // Source register 2
    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
    // If the flush signal is 1, we reset the values of all the registers.
    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
```

```
    // If flush signal is 0, we continue the transfer of data from ID stage to EX stage.
   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
```

EX_MEM: Similar to the previous pipeline stage this module primarily acts as a latch to hold the data and control signals during the EX/MEM stage of the pipeline. It passes the data and control signals from the EX stage to the MEM stage in a single-cycle pipeline processor. In case of a flush, it ensures that the pipeline registers are cleared to maintain correctness in the pipeline operation. We start by defining our input signals clk and flush signals with the same purpose as before.

The following control signals are used:

- **RegWrite:** Control signal for enabling register write.
- **MemtoReg:** Control signal for selecting memory or ALU result for register write.

- **Branch:** Control signal for branch instruction.
- **Zero:** Control signal indicating the ALU result is zero.
- **MemWrite:** Control signal for memory write.
- **MemRead:** Control signal for memory read.
- **is_greater:** Control signal indicating the comparison result of the ALU operation.

The following input registers of 64 bits are used:

- **immvalue_added_pc:** Immediate value added to the program counter.
- **ALU_result:** Result of the ALU operation.
- **WriteData:** Data to be written to memory or register file.

  4-bit register function_Code and 5 bit register destination_reg is also defined which hold the values for the function code and destination register for reg write.

The following output registers are defined:

RegWrite_out: This signal indicates whether the result produced in the current pipeline stage should be written back to a register in the Register File.

- **MemtoReg_out:** This signal selects whether the data to be written back to a register in the Register File should come from the memory (e.g., in the case of a load instruction) or from the ALU result (e.g., in the case of arithmetic or logical operations).
- **Branch_out:** This signal indicates whether the current instruction is a branch instruction. It is typically used to control the behavior of the branch predictor and the program counter.
- **Zero_out:** This signal indicates whether the result of the ALU operation is zero. It is used in conditional branches and comparisons.
- **MemWrite_out:** This signal indicates whether the current instruction is a memory write operation. It controls whether the data from the ALU should be written to memory.
- **MemRead_out**: This signal indicates whether the current instruction is a memory read operation. It controls whether data should be read from memory into a register.
- **is_greater_out**: This signal indicates the result of a comparison operation performed by the ALU. It is typically used in conditional branches and comparisons.
- **immvalue_added_pc_out:** This signal represents the immediate value added to the program counter. It's often used in branch and jump instructions to calculate the target address.

- **ALU_result_out:** This signal represents the result produced by the ALU (Arithmetic Logic Unit). It could be the result of arithmetic, logical, or shift operations.
- **WriteData_out**: This signal represents the data to be written to memory or to a register in the Register File.
- **function_code_out:** This signal represents the function code for the ALU operation. It determines the specific operation to be performed by the ALU, such as addition, subtraction, bitwise AND, etc.
- **destination_reg_out:** This signal represents the destination register for register write operations. It specifies which register in the Register File should receive the result of the computation.

On each rising edge of the clock (clk), the module checks if the Flush signal is active.

- If Flush is active, all output registers are reset to zero.
- If Flush is inactive, input values are passed to output registers to propagate them to the next pipeline stage.

```
module EX_MEM(
    input clk,                  // Clock signal
    input Flush,                // Flush signal
    input RegWrite,             // Control signal for determining register write
     input MemtoReg,                    // Control signal for choosing between memory or ALU result for register write
    input Branch,               // Control signal for branch
    input Zero,                 // Control signal to determine whether the result of the ALU is 0
    input MemWrite,             // Control signal for memory write
    input MemRead,              // Control signal for memory read
     input is_greater,          // Control signal to determine the comparison result of the ALU operation
    input [63:0] immvalue_added_pc, // Immediate value added to PC to calculate address
    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 to determine the alu operation
    input [4:0] destination_reg,   // Destination register for register write

    output reg RegWrite_out,       // Output signal for determining 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 register write
);

    // Determine output values based on control signals
    always @(posedge clk) begin
        if (Flush) begin
            // Reset output values when flush signal is 1
            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
            // Determine 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
```

MEM_WB: This is the final stage of the pipelined processor. This module also acts as a latch to hold the data and control signals during the MEM/WB stage of the pipeline then it passes the data and control signals from the MEM stage to the WB stage in a single-cycle pipeline processor.
The following input signals are defined:

- **clk:** Clock signal used for synchronization.
- **RegWrite:** Control signal for enabling register write.

- **MemtoReg:** Control signal for selecting memory or ALU result for register write.
- **ReadData:** Data read from memory or the Register File.
- **ALU_result:** Result of the ALU operation.
- **destination_reg:** Destination register for register write.

The following output signals are defined

- **RegWrite_out:** Output signal indicating whether the result produced in the current pipeline stage should be written back to a register in the Register File.
- **MemtoReg_out:** Output signal selecting whether the data to be written back to a register in the Register File should come from the memory (e.g., in the case of a load instruction) or from the ALU result (e.g., in the case of arithmetic or logical operations).
- **ReadData_out:** Output signal representing the data read from memory or the Register File.
- **ALU_result_out:** Output signal representing the result produced by the ALU (Arithmetic Logic Unit).
- **destination_reg_out:** Output signal representing the destination register for register write operations.

On each rising edge of the clock (clk), the module assigns the input values directly to the corresponding output registers.

Forwarding Unit: This module represents a Forwarding Unit in a single pipelined RISC-V processor. Its primary purpose is to resolve data hazards by forwarding data from the output of one pipeline stage to the input of a previous stage without waiting for the data to be written back to the register file.

- **`EXMEM_rd`, `MEMWB_rd`:** These inputs represent the destination register numbers from the EX/MEM and MEM/WB pipeline stages, respectively.
- **`IDEX_rs1`, `IDEX_rs2`**: These inputs represent the source register numbers from the IDEX pipeline stage.
- **`EXMEM_RegWrite`, `EXMEM_MemtoReg`, `MEMWB_RegWrite`:** These inputs represent

control signals indicating whether register writes are enabled and whether the data to be written back comes from memory in the EX/MEM and MEM/WB pipeline stages.

- **`fwd_A`, `fwd_B`:** These outputs represent control signals indicating the source of the operands A and B for the current instruction being processed.

Changes made in the instruction memory and register file:

The following changes were made in the instruction memory:

Sinc we are testing three add instructions:

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

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

Here Type 1a hazard occurs.

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

Here type 2a hazard occurs.

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

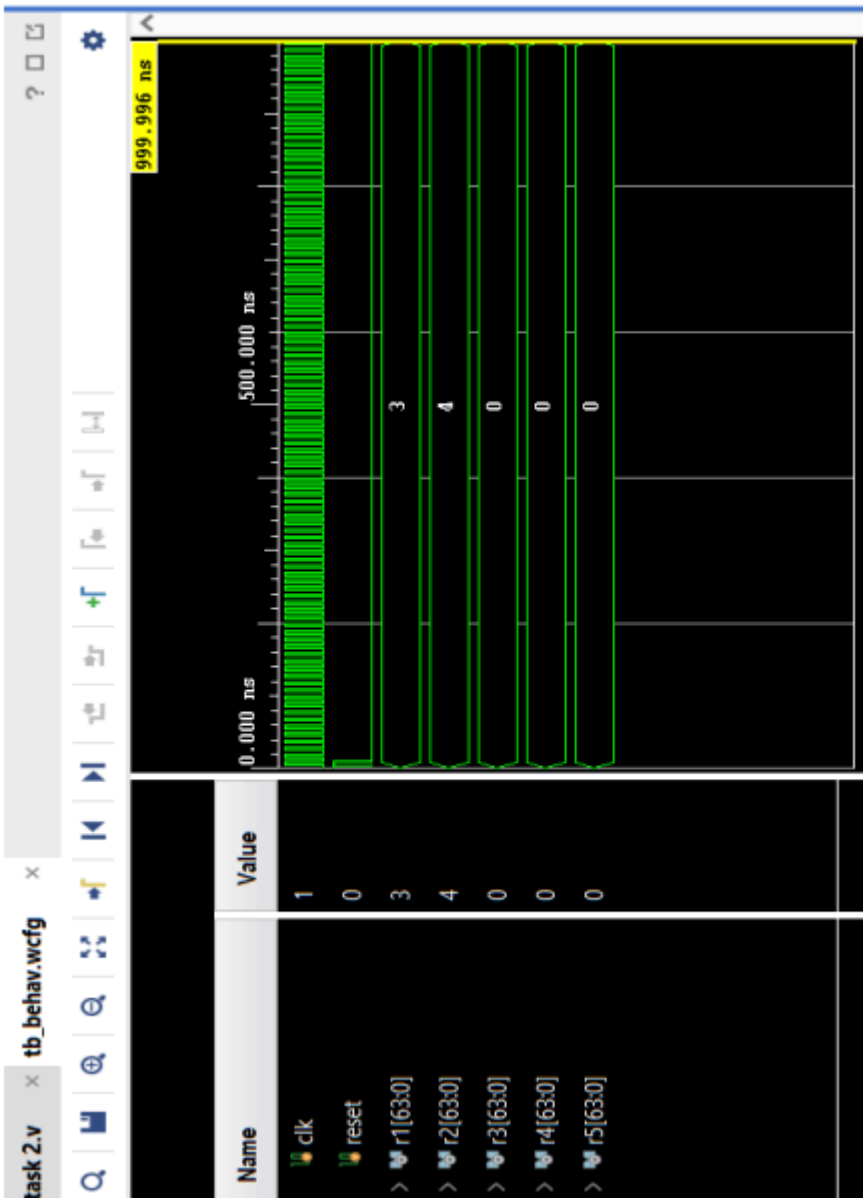
```
    inst_mem[10] = 8'b10100100;
    inst_mem[11] = 8'b00000000;




            end
```

The following registers were defined in the register file:

```
initial
begin
    for (i = 0; i < 32; i = i + 1)
            Registers[i] = 64'd0;
    Registers[12] = 64'd3;
    Registers[13] = 64'd4;
end
 assign r1 = Registers[12];
  assign r2 = Registers[13];
  assign r3 = Registers[10];
  assign r4 = Registers[8];
  assign r5 = Registers[1];
```

Without the hazard detection module, the actual result comes out to be

However if we add the hazard detection the follwoing will result:



Task 3: In Task 3 we introduced 3 new modules to detect hazards. We used the following modules in our entire code:

Forwarding Unit

Hazard Detection

Branch Control

The Forwarding_Unit module handles data forwarding in a pipelined processor. It detects and

resolves data hazards by forwarding data from previous pipeline stages to the current stage.
The module takes input signals representing register destinations and control signals from the EX/MEM and MEM/WB pipeline stages, as well as source register values from the ID/EX pipeline stage. Based on these inputs, it determines whether data needs to be forwarded for operands A and B in the current stage.
For each operand, it checks if the register value is produced by a previous instruction that has been written to the register. If so, it forwards the value accordingly. The forwarding logic prioritizes the most recent data available, checking first for forwarding from the EX/MEM stage and then from the MEM/WB stage.
The module outputs signals indicating the forwarding status for operands A and B, facilitating the proper execution of instructions in the pipeline while avoiding stalls due to data hazards.

The Hazard_Detection module takes inputs representing the current instruction's destination register (current_rd), and the source registers of the previous instruction (previous_rs1 and previous_rs2). Additionally, it receives a signal indicating whether the current instruction is a memory read operation (current_MemRead).

Hazard condition is where a current memory read instruction depends on the result of a previous instruction's memory write to the same register. If such a hazard is detected, it disables the multiplexer output (mux_out), prevents writing to the next pipeline stage (enable_Write), and disables PC write (enable_PCWrite) to stall the pipeline and resolve the hazard.

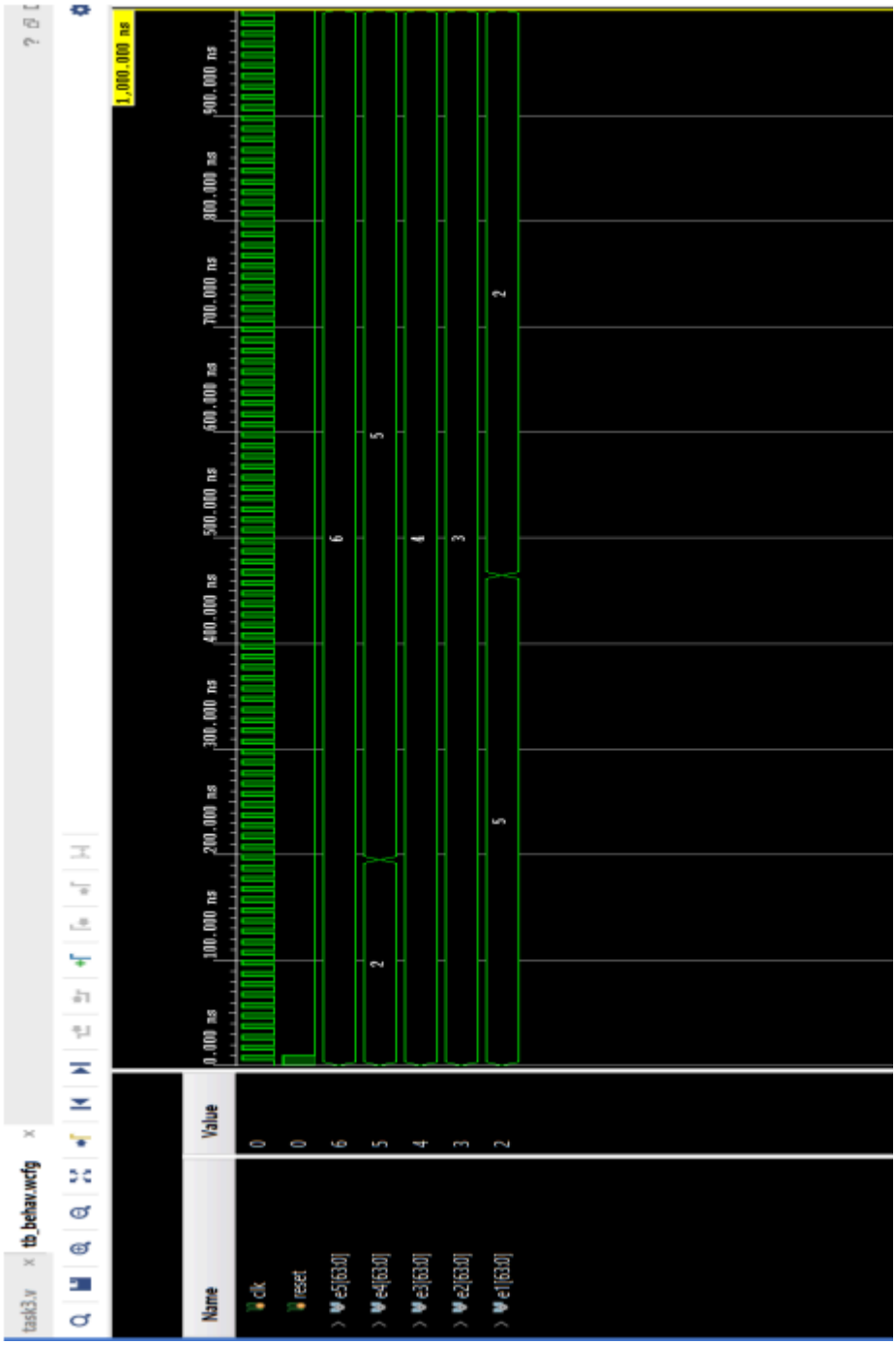
If no hazard is detected, the module enables the multiplexer output and allows writing to the next pipeline stage and PC write to proceed normally.

The Branch_Control module decides whether to take a branch instruction based on various conditions and generates a control signal (switch) accordingly. It generates a Flush signal to indicate whether the pipeline needs to be flushed due to a branch instruction being taken.
inputs:

Branch: Indicates whether a branch instruction is being executed.
Zero: Indicates whether the result of the ALU operation is zero.
Is_Greater_Than: Indicates whether the result of the ALU operation is greater than zero.
funct[3:0]: Specifies the function code of the instruction.
The module uses a case statement based on the lower three bits of the function code (funct[2:0]). It then checks various conditions depending on the function code and input signals to determine whether to take the branch. Based on the conditions, it sets the switch signal to either 1 (indicating the branch should be taken) or 0 (indicating the branch should not be taken).

The Flush signal is set based on the value of the switch. If switch is 1, indicating that the branch is taken, Flush is set to 1 to indicate that the pipeline needs to be flushed.

If no branch instruction is being executed (Branch is inactive), switch is set to 0, and Flush is also set to 0.


task3.v
tb_behav.wcfg
Name
Value
clk
reset
e5[63:0]
e4[63:0]
e3[63:0]
e2[63:0]
e1[63:0]
0
0
6
5
4
3
2
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns
800.000 ns
900.000 ns
1,000.000 ns

Task 4:

Previously, sorting integers with the single-cycle processor took 995ns. However, employing the RISC-V Processor reduced the sorting time to 465ns for the same integers. Despite bubble sort having a time complexity of O(n^2), its performance can be enhanced on a RISC-V Processor.

Challenges

- The bubble sort code was proving to be a challenge and we spent two three days just testing out codes that worked on our modules and were able to achieve it in the end
- Implementing our top module especially with our modules being used in multiple tasks was a big challenge
- In task 2 our code is not working properly with just forwarding it is only working if the hazard detection module is working.
- The project in itself was challenging as we had to code the entire pipelined processor

Task Division

- Our task division is very rough as we mostly sat together to handle one task at a time so we all managed to work on each and every one of the tasks
- Task 1 - Azkaa
- Task 2 - Fatima
- Task 3 - Afifah

Conclusions

- How did your project turn out?

The project turned out to be incredibly helpful in deepening my understanding of how the RISC-V processor works. It not only enhanced my theoretical understanding but also gave me practical insights into processor design.

- Briefly state why it was or was not successful.

The project was successful as all the tasks were completely achieved!

References (If any)

We took most of our codes from the labs and modified others from our textbook to fit in our code.

Appendices

We have uploaded the relevant modules in our report. The rest of the code can be found at:

1