CE-321L/CS-330L: Computer Architecture
Final Lab Project

Namel Shahid (08327), Ammar Murtaza (08721), Arqam Nakhoda (07195)

Report Structure:

Introduction

Project: Pipelined Processor for array sorting using previously developed modules;

This project aims to develop a 5-stage pipelined processor using Verilog HDL, focusing on executing a specific array sorting algorithm written in RISCV assembly language. Our objective is to enhance efficiency compared to a traditional single-cycle processor by implementing pipelining.

The project structure involves three main phases:

1. Single-cycle implementation: Initially, we establish the groundwork by constructing a basic single-cycle processor.

2. Pipelining integration: Subsequently, we enhance the design by incorporating a 5-stage pipeline, aiming to optimize performance for accelerated execution.

3. Detailed report: Throughout the project, each stage will be meticulously documented in accordance with the provided rubrics, ensuring comprehensive coverage of the development process and outcomes.

2. Methodology

Task-1 – (Sorting Algorithm on Single Cycle)

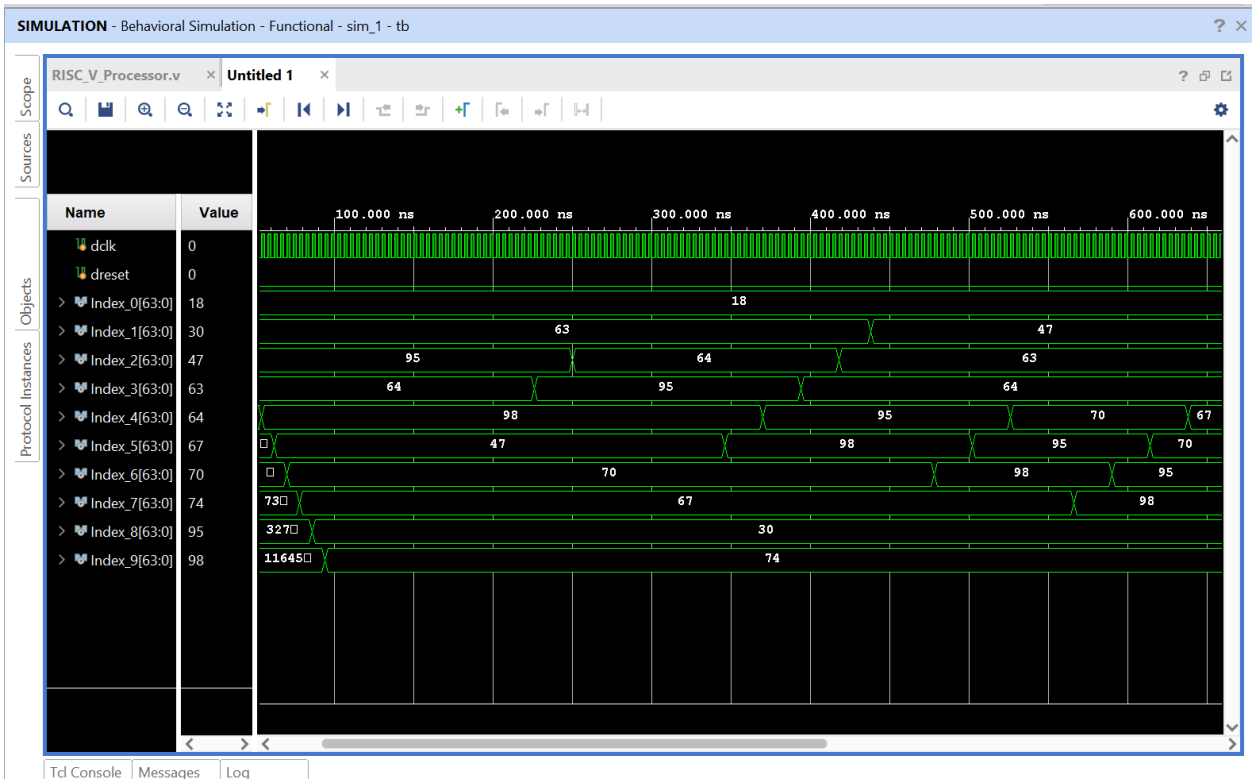
Insertion sort implemented within instruction memory.

One of the most commonly used sorting algorithms is Quicksort due to its average-case time complexity of O(n log n) and its relatively low overhead. However, for simplicity, we chose insertion Sort for this exercise since its pseudocode is straightforward and easy to convert to assembly language.

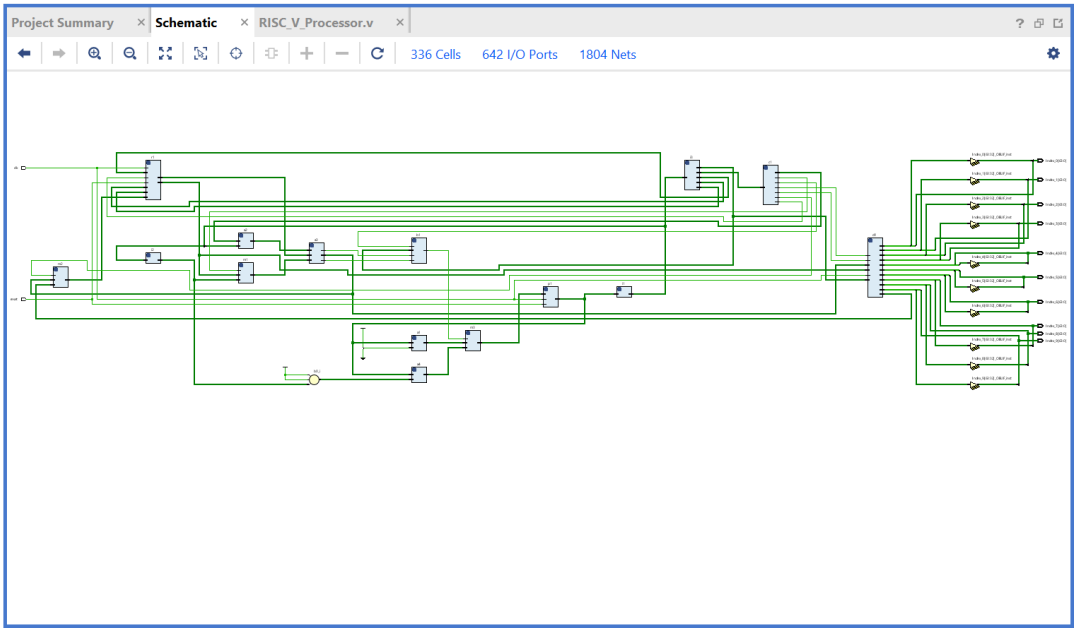
C language psuedocode was converted to Assembly language and then tested on venus (online simulator). For the algorithm (converted in Assembly language). We tested for different cases.

STIMULATION FOR TASK 1

Output of values stored in MemoryArray (Sorted)



SCHEMATIC FOR TASK 1



3. **Changes Implemented in Single Cycle Processor**

The changes were made in the ALU, and the last MUX module and a new Branch module was introduced to make our sorting algorithm work**.**

To enhance our ALU, we introduced a new bit that holds the negation of the last bit of the ALU Result. Then, we introduced another module called the Branch module. We know that if the Zero from the ALU Result is 1, then the branch statement is a beq, and if it is 0, then the statement is a bne. By introducing another variable, we can OR it with the Zero to add more functionality. So, if both variables are 1, then we have the bgt statement, and if both are 0, then we have the blt statement. Next, we AND these statements to make our branch variable 1, as we have to control the MUX with this variable. By doing that, we finally enable our processor to work with bgt and blt statements.

ALU_64_bit:

```
module ALU_64_bit(a, b, ALUOp, Zero, Result, Pos);

  input [63:0] a;
  input [63:0] b;
  input [3:0]ALUOp;
  output reg Zero;
  output reg [63:0] Result;
  output reg Pos;


  always @(*) begin
    if (ALUOp == 4'b0000) begin
     Result = a & b;
  end
    else if (ALUOp == 4'b0001) begin
     Result = a | b;
  end
    else if (ALUOp == 4'b0010) begin
     Result = a + b;
  end
    else if (ALUOp == 4'b0110) begin
     Result = a - b;
  end
    else if (ALUOp == 4'b1100) begin
```

```
    Result = ~(a|b);
  end
   if (Result == 0)
    Zero = 1;
   else
    Zero = 0;
   Pos <= ~Result[63];
  end
endmodule
```

This Verilog module implements a 64-bit Arithmetic Logic Unit (ALU) capable of performing various arithmetic and logical operations based on the specified ALU operation code (`ALUOp`). The module takes two 64-bit inputs `a` and `b`, and based on the value of `ALUOp`, it performs one of the following operations: bitwise AND (`&`), bitwise OR (`|`), addition (`+`), subtraction (`-`), or bitwise NOR (`~(a|b)`). The result of the operation is stored in the 64-bit `Result` output. Additionally, the module computes whether the result is zero (`Zero`) and whether it's positive (`Pos`) by checking the most significant bit (MSB) of the result. If the result is zero, `Zero` is set to 1; otherwise, it's set to 0. Similarly, `Pos` is set to 1 if the MSB of the result is 0, indicating a positive value.

MUX:

```
module MUX(X, Y, S, O);

  input [63:0] X;
  input [63:0] Y;
  input S;
  output [63:0] O;

  assign O = S ? Y : X;
endmodule
```

This is a multiplexer that selects between two input signals `X` and `Y` based on the value of a select signal `S`. The inputs `X` and `Y` are 64 bits wide, allowing for multiplexing of large data buses. The select signal `S` determines which input is passed through to the output `O`, where `O` is also a 64-bit wide signal. When `S` is 0, the output `O` mirrors the value of `X`, and when `S` is 1, the output `O` mirrors the value of `Y`. This behavior is implemented using a conditional assignment statement, where `O` is assigned the value of `Y` when `S` is 1, and it is assigned the value of `X` when `S` is 0, effectively realizing the functionality of a multiplexer.

Branch Module:

```
module branch_module(zero, pos, branch, funct3, bne, beq, bge, blt, to_branch);

  input zero;
  input pos;
  input branch;
  input [2:0] funct3;
  output reg bne;
  output reg beq;
  output reg bge;
  output reg blt;
  output reg to_branch;


  always @(*)
    begin
      if (branch) begin
        if (zero && funct3 == 3'b000) begin
            beq <= 1'b1;
            bne <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if (~zero && funct3 == 3'b001) begin
            bne <= 1'b1;
            beq <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if ((pos || zero) && funct3 ==3'b101) begin
            bne <= 1'b0;
            beq <= 1'b0;
            bge <= 1'b1;
            blt <= 1'b0;
        end
        else if ((~pos && ~zero) && funct3 ==3'b100) begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b1;
            bge <= 1'b0;
        end
        else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
```

```
            bge <= 1'b0;
        end
    end
    else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
            bge <= 1'b0;
    end
    to_branch <= branch && (bne || beq || blt || bge);
  end
endmodule
```

branch_module is designed to generate control signals for branching operations based on input conditions. It takes several inputs including zero, pos, branch, and funct3, and produces control signals bne, beq, bge, and blt, as well as to_branch. The module uses conditional statements to determine the appropriate control signals based on the input conditions. If the branch signal is asserted, the module checks various conditions based on the values of zero, pos, and funct3. Depending on these conditions, specific control signals such as bne, beq, bge, and blt are set accordingly to facilitate branching operations. If the branch signal is not asserted, all control signals are cleared to 0. Finally, the to_branch signal is set to 1 if any of the branching control signals are active, allowing it to signal when a branch operation is to be executed. Overall, this module efficiently handles branching logic in a digital system.

4. **Task-2 – (Introducing Pipeline Stages)**

Implementing a single-cycle processor presents a challenge due to its sequential execution of instructions, resulting in significant idle time for various components. To address this inefficiency, pipelining is introduced to enhance processing power and resource utilization.

In our RISC-V processor, a five-stage pipeline is adopted, allowing the concurrent handling of five instructions.

These pipeline stages include:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execution or Address Calculation (EX)
4. Data Memory Access (MEM)
5. Write Back (WB)

To support pipelining, four new registers are introduced:

1. IF/ID
2. ID/EX
3. EX/MEM
4. MEM/WB

These registers enable the simultaneous handling of multiple instructions and track their progress through the pipeline, enhancing processor performance by enabling parallel instruction processing. Additionally, a control line and a forwarding unit are incorporated to facilitate communication between pipeline stages. Timed to the clock, these components either pass stored contents for further processing or flush on each positive clock edge. Despite the benefits of pipelining, considerations such as handling branch instructions and managing the program counter (PC) incrementation must be addressed to ensure smooth pipeline operation. Overall, the introduction of pipelining optimizes processor efficiency by enabling concurrent instruction execution and improving resource utilization.

Instruction Fetch/Instruction Decode (IF/ID):

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

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

  always @ (posedge clk or posedge reset)
    begin
    if (reset == 1'b1)
      begin
        PC_Out <= 0;
        Inst_output <= 0;
      end
     else
       begin
         PC_Out = PC_In;
         Inst_output <= Inst_input;
       end
    end
endmodule
```

This Verilog module represents the Instruction Fetch/Decode stage in a processor, contains registers for storing the Program Counter (PC) and the fetched instruction. It operates on a clock signal (`clk`) and a reset signal (`reset`). When the reset signal is asserted (`reset == 1'b1`), indicating a reset condition, both the PC and instruction output are set to zero. Otherwise, during normal operation, the PC output is

updated with the value of `PC_In`, representing the next PC value fetched from the previous stage. Similarly, the instruction output (`Inst_output`) is updated with the value of `Inst_input`, which represents the instruction fetched from memory or elsewhere. The module's behavior is synchronized with the positive edge of the clock signal, ensuring stable operation within a processor pipeline.

Instruction Decode/Execution (ID/EX):

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

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

```
  always @ (posedge clk or posedge reset)
    begin
      if (reset == 1'b1)
        begin
          PC_Out<= 0;
          Funct_out <= 0;
          ALUOp_out <= 0;
                    MemtoReg__out <= 0;
                    RegWrite_out <= 0;
                    Branch_out <= 0;
                    MemWrite_out <= 0;
                    MemRead_out <= 0;
                    ALUSrc_out <= 0;
                    ReadData1_out <= 0;
                    ReadData2_out <= 0;
                    rs1_out <= 0;
                    rs2_out <= 0;
                    rd_out <= 0;
                    imm_data_out <= 0;
        end
      else
        begin
           PC_Out<= PC_In;
          Funct_out <= Funct_inp ;
          ALUOp_out <= ALUOp_inp;
                    MemtoReg__out <= MemtoReg_inp;
                    RegWrite_out <= RegWrite_inp;
                    Branch_out <= Branch_inp;
                    MemWrite_out <= MemWrite_inp;
                    MemRead_out <= MemRead_inp;
                    ALUSrc_out <= ALUSrc_inp;
                    ReadData1_out <=  ReadData1_inp;
                    ReadData2_out <=  ReadData2_inp;
                    rs1_out <= rs1_in;
                    rs2_out <= rs2_in;
                    rd_out <= rd_inp;
                    imm_data_out <= imm_data_inp;
        end
    end
endmodule
```

This Verilog module models the Instruction Decode/Execute stage in a processor pipeline. It takes inputs representing various control signals (`Funct_inp`, `ALUOp_inp`, `MemtoReg_inp`, `RegWrite_inp`, `Branch_inp`, `MemWrite_inp`, `MemRead_inp`, `ALUSrc_inp`) along with data inputs such as register values (`ReadData1_inp`, `ReadData2_inp`), source and destination register addresses (`rs1_in`, `rs2_in`, `rd_inp`), and immediate data (`imm_data_inp`). On the positive edge of the clock (`clk`) or a positive edge of the reset signal (`reset`), the module updates its outputs based on these inputs. During a reset

condition (`reset == 1'b1`), all outputs are set to zero. Otherwise, the outputs are updated with the values of the corresponding inputs. These outputs represent the control signals and data forwarded to the subsequent stages of the processor pipeline for further processing and execution. The module ensures synchronization and proper functioning of the pipeline stage by updating its outputs only on the positive edge of the clock or reset signal.

Execution/Memory (EX/MEM):

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

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


  always @ (posedge clk or posedge reset)
    begin
      if (reset == 1'b1)
        begin
          PC_Out<= 0;
          Result_out <=0;
          ZERO_out <= 0;
                    MemtoReg_out <= 0;
                    RegWrite_out <= 0;
```

```
                Branch_out <= 0;
                MemWrite_out <= 0;
                MemRead_out <= 0;
                rd_out <= 0;
                data_out <= 0;
      end
     else
      begin
       PC_Out<= PC_In;
       Result_out <= Result_inp;
       ZERO_out <= ZERO_inp ;
                MemtoReg_out <= MemtoReg_inp;
                RegWrite_out <= RegWrite_inp;
                Branch_out <= Branch_inp;
                MemWrite_out <= MemWrite_inp;
                MemRead_out <= MemRead_inp;
                rd_out <= rd_inp;
                data_out <= data_inp;
      end
   end
endmodule
```

EXMEM represents the Execute/Memory stage in a processor pipeline. It takes inputs such as register destination address (`rd_inp`), branch control signal (`Branch_inp`), memory write signal (`MemWrite_inp`), memory read signal (`MemRead_inp`), memory-to-register signal (`MemtoReg_inp`), register write signal (`RegWrite_inp`), program counter value (`PC_In`), result of the ALU operation (`Result_inp`), and a flag indicating if the result is zero (`ZERO_inp`). On the positive edge of the clock (`clk`) or a positive edge of the reset signal (`reset`), the module updates its outputs based on these inputs. During a reset condition (`reset == 1'b1`), all outputs are set to zero. Otherwise, the outputs are updated with the values of the corresponding inputs. These outputs include the program counter value for the next instruction (`PC_Out`), the result of the ALU operation (`Result_out`), a flag indicating if the result is zero (`ZERO_out`), and control signals and data forwarded to the subsequent stages of the processor pipeline for further processing and execution.

Memory/Write back (MEM/WB):

```
module MEMWB(clk, reset, Result_inp, Read_Data_inp, rd_inp, MemtoReg_inp,
RegWrite_inp, MemtoReg_out, RegWrite_out, Result_out, Read_Data_out, rd_out);

  input clk;
  input reset;
  input wire [63:0] Result_inp;
  input [63:0]Read_Data_inp;
  input [4:0] rd_inp;
  input wire MemtoReg_inp;
```

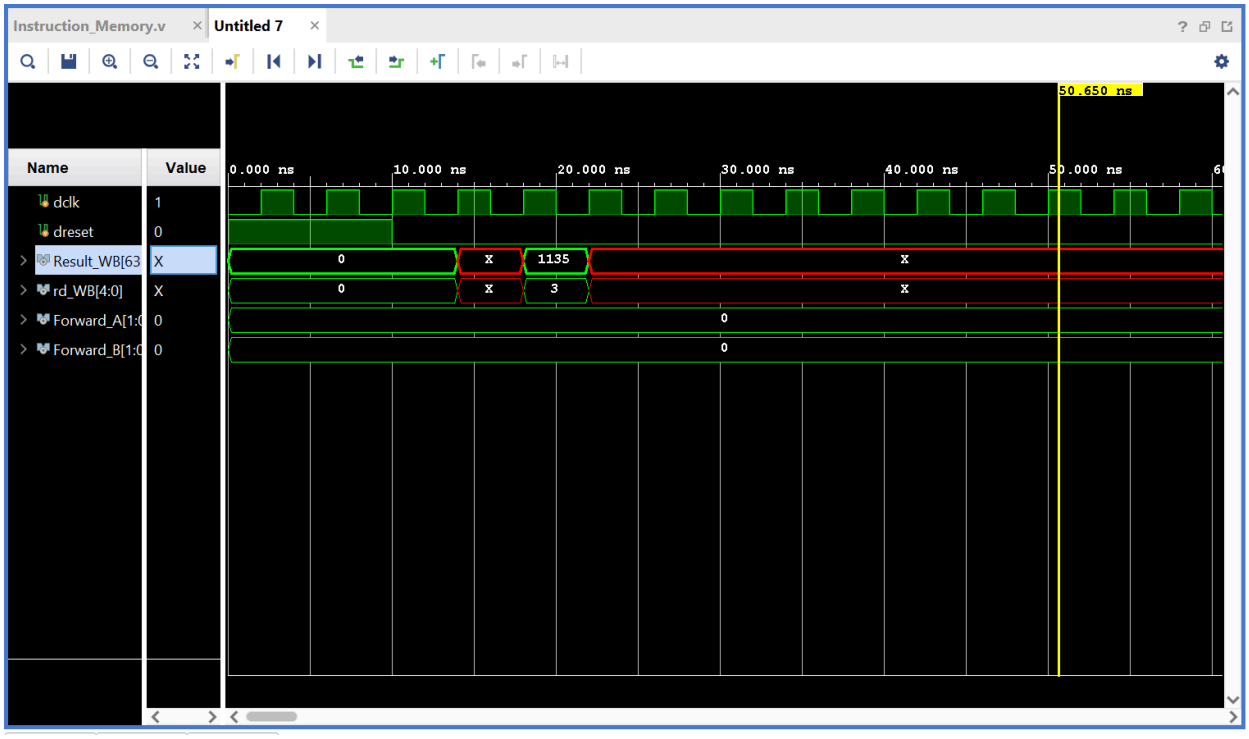
```
  input RegWrite_inp;
  output reg MemtoReg_out;
  output reg RegWrite_out;
  output reg [63:0] Result_out;
  output reg [63:0]Read_Data_out;
  output reg [4:0] rd_out;


  always @ (posedge clk or posedge reset)
    begin
    if (reset == 1'b1)
      begin
        Result_out <= 0;
        Read_Data_out <= 0;
        rd_out <= 5'b0;
        MemtoReg_out <= 0;
        RegWrite_out <= 0;
      end
     else
       begin
        Result_out <= Result_inp;
        Read_Data_out <= Read_Data_inp;
        rd_out <= rd_inp;
        MemtoReg_out <= MemtoReg_inp;
        RegWrite_out <= RegWrite_inp;
       end
    end
endmodule
```

MEMWB represents the Memory Writeback stage in a processor pipeline. It takes inputs such as the result of memory operation (`Result_inp`), data read from memory (`Read_Data_inp`), and the destination register address (`rd_inp`). Additionally, it receives control signals indicating whether to write the result to a register (`RegWrite_inp`) and whether the data comes from memory (`MemtoReg_inp`). On the positive edge of the clock (`clk`) or a positive edge of the reset signal (`reset`), the module updates its outputs based on these inputs. During a reset condition (`reset == 1'b1`), all outputs are set to zero. Otherwise, the outputs are updated with the values of the corresponding inputs. These outputs include the result to be written back to the register file (`Result_out`), the data read from memory (`Read_Data_out`), the destination register address (`rd_out`), and the control signals indicating whether to write back to the register file (`MemtoReg_out`) and whether to perform the write operation (`RegWrite_out`). This module ensures proper synchronization and handling of data and control signals in the processor pipeline's Memory Writeback stage.

Forwarding Unit:

Before implementing forwarding, the value stored in register x3 was arbitrary or garbage because forwarding wasn't utilized. However, after implementing forwarding, the value in register x3 became the same as the value in register x6. This change occurred because now the value is being forwarded from the previous instruction to the subsequent one, ensuring that the correct data is available for the subsequent operations.

WB value is garbage as seen below:
(not using forwarding)



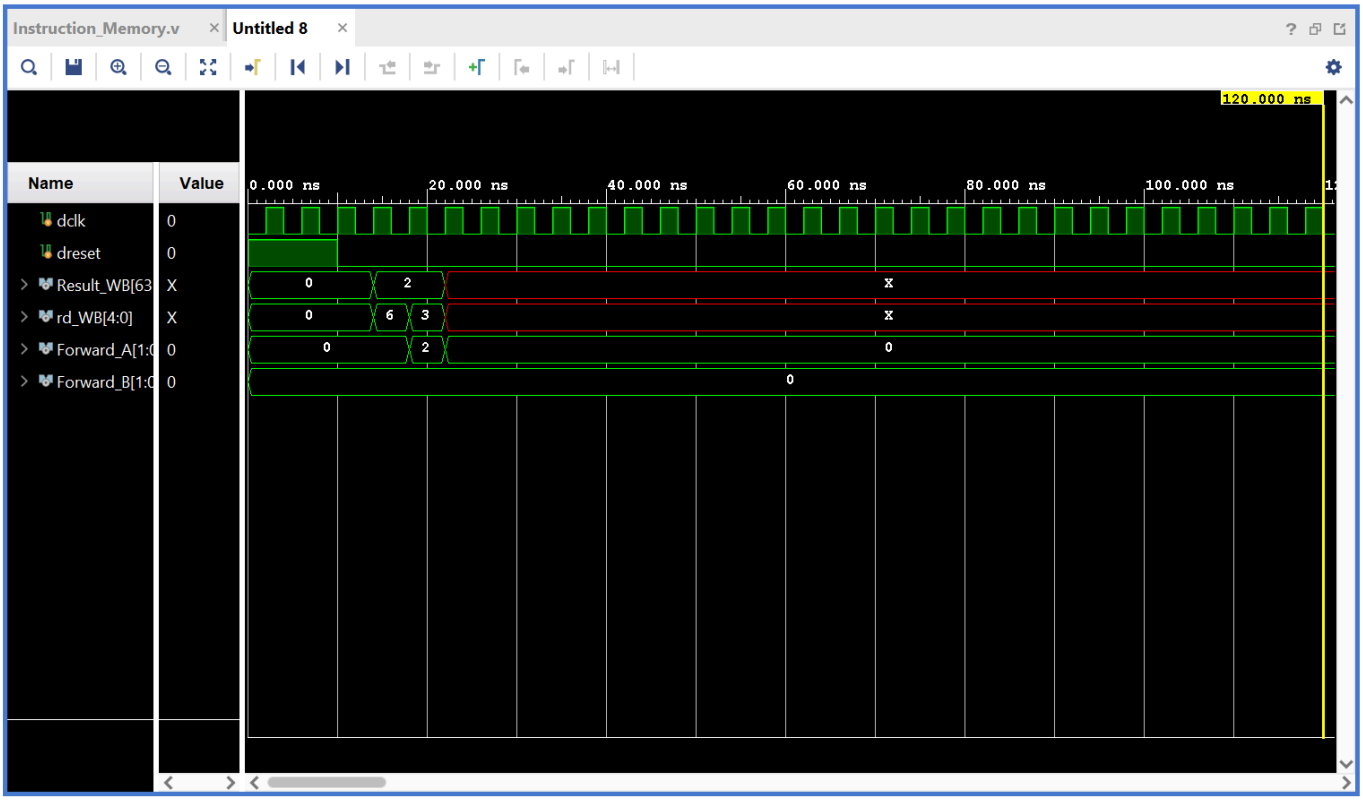
```
module Forwarding_Unit(EX_MEM_rd, MEM_WB_rd, ID_EX_rs1, ID_EX_rs2,
EX_MEM_RegWrite, MEM_WB_RegWrite, forward_A, forward_B);
  input wire[4:0] EX_MEM_rd;
  input wire [4:0] MEM_WB_rd;
  input wire [4:0] ID_EX_rs1;
  input wire [4:0] ID_EX_rs2;
  input wire EX_MEM_RegWrite;
  input wire MEM_WB_RegWrite;
  output reg [1:0] forward_A;
  output reg [1:0] forward_B;

always @(EX_MEM_rd or MEM_WB_rd or EX_MEM_RegWrite or MEM_WB_RegWrite or
ID_EX_rs1 or ID_EX_rs2)
  begin
    if ((EX_MEM_RegWrite == 1) && (EX_MEM_rd != 0) && (EX_MEM_rd == ID_EX_rs1))
          forward_A <= 2'b10;
    else if ((MEM_WB_RegWrite == 1) &&  (MEM_WB_rd != 0)  && (MEM_WB_rd ==
ID_EX_rs1)&& !(EX_MEM_RegWrite == 1 && EX_MEM_rd != 0 && EX_MEM_rd ==
ID_EX_rs1))
          forward_A <= 2'b01;
    else
          forward_A <= 2'b00;



    if((EX_MEM_rd == ID_EX_rs2) &&(EX_MEM_RegWrite == 1) && (EX_MEM_rd!=0))
       forward_B <= 2'b10;

    else if( (MEM_WB_rd==ID_EX_rs2) && (MEM_WB_RegWrite==1) &&
(MEM_WB_rd!=0) && !(EX_MEM_RegWrite == 1 && EX_MEM_rd != 0 && EX_MEM_rd
== ID_EX_rs2))
       forward_B <= 2'b01;
    else
       forward_B = 2'b00;
 end

endmodule
```

the Forwarding_Unit serves as a component that determines whether to forward data from later pipeline stages to earlier ones to resolve data hazards efficiently. It takes input signals representing the state of the pipeline stages, including destination register numbers and write signals, and produces output signals indicating whether forwarding is necessary and to which source register the data should be forwarded. Through combinatorial logic, it evaluates conditions such as register writes and register number matches to determine the appropriate forwarding action for each source register. The forward_A and forward_B outputs represent the forwarding decisions for the two source registers in the instruction decode/execution stage, enabling the processor to handle data dependencies dynamically and minimize stalls during instruction execution.

TEST CASE 1:

addi x6, x0, 2
add x3, x6, x0

However, after implementing forwarding, the value in register x3 became the same as the value in register x6. This change occurred because now the value is being forwarded from the previous instruction to the subsequent one, ensuring that the correct data is available for the subsequent operations, as shown below.



5. **Task 3 – (Hazard Detection Circuitry)**

Hazard detection Unit:

```
module Hazard_Detection_Unit(
   input [4:0] rs1_ID, rs2_ID, rd_EX, rd_MEM,
   input MemRead_EX, Branch_ID,
   output reg stall_IF, stall_ID, flush_EX
);
   always @(*) begin
      // Load-use hazard detection
      stall_IF = MemRead_EX && ((rd_EX == rs1_ID) || (rd_EX == rs2_ID));
      stall_ID = stall_IF; // Stall ID stage when IF is stalled
      flush_EX = Branch_ID; // Flush EX on branch
   end
endmodule
```

The Verilog code defines a Hazard Detection Unit (HDU) module for a pipelined processor. The module takes input signals representing various register IDs and control signals such as MemRead_EX and Branch_ID. Inside an always block, the HDU determines if there is a load-use hazard by comparing the destination register of the execute stage (rd_EX) with the source registers of the instruction fetch stage (rs1_ID and rs2_ID). If a hazard is detected, the instruction fetch stage is stalled (stall_IF). The stall signal is propagated to the instruction decode stage (stall_ID). Additionally, the module identifies branch instructions (Branch_ID) and signals to flush the execute stage (flush_EX) to maintain correct program execution. This HDU ensures that the pipeline operates smoothly, handling hazards and branch instructions appropriately to avoid data hazards and ensure correct program flow.

6. **Task 4 – (Performance Comparison: Single Cycle vs. Pipelined RISC-V Processor)**

Single Cycle Processor:

The single-cycle processor executes each instruction in a single clock cycle, making it straightforward but potentially inefficient for complex instructions. In the context of sorting algorithms, such as bubble sort or insertion sort, the single-cycle processor would execute each step of the algorithm sequentially, without overlapping instruction execution. While simple to implement and understand, this approach might result in longer execution times for sorting large datasets due to the lack of instruction pipelining and parallelism.

Pipelined RISC-V Processor:

In contrast, the pipelined RISC-V processor divides the instruction execution into multiple stages (fetch, decode, execute, memory, write-back) and allows multiple instructions to be processed simultaneously. This pipelining increases throughput and efficiency by overlapping the execution of different instructions. For sorting algorithms, the pipelined processor can exploit instruction-level parallelism to execute multiple sorting steps concurrently, potentially resulting in shorter execution times compared to the single-cycle processor, especially for large datasets.

Comparison:

When comparing the performance of running the array sorting program on the single-cycle processor versus the pipelined RISC-V processor, several factors come into play. The single-cycle processor may struggle with the inherently sequential nature of sorting algorithms, executing each step one after the other. This approach could lead to longer execution times, particularly for larger datasets, as it cannot exploit parallelism. On the other hand, the pipelined RISC-V processor can leverage its instruction pipelining to overlap the execution of different sorting steps, achieving better performance through parallelism and higher throughput. Therefore, the pipelined processor is expected to exhibit shorter execution times compared to the single-cycle processor when running sorting algorithms, especially for larger datasets where parallelism can be fully utilized.

Challenges

we struggled with understanding the appropriate sorting algorithm to be used. The struggle between choosing bubble sort and insertion sort stemmed from the need to balance performance and simplicity within the context of the provided code. Bubble sort and insertion sort are both relatively simple sorting algorithms.

Considering the need for efficiency and simplicity in hardware description, we chose to use insertion sort. Its performance is generally better, especially for smaller datasets.

Task Division
We sat together and discussed the project thoroughly, We started task 1 and 2 in the lab, and these were completed by Namel afterwards. Task 3 was done by Arqam and Ammar.

Conclusions
The project was mostly successful.

Appendix

github repository link: https://github.com/namel20/CA_project.git