CE-321L/CS-330L: Computer Architecture Lab
RISC-V Pipelined Processor

Sara Baloch 08586,
Maria Adnan 08439,

27th April'24.

TABLE OF CONTENTS:

1. Introduction

In this project, our objective was to design and implement a 5-stage pipelined processor capable of executing any one array sorting algorithm. The main challenge was to convert a single-cycle processor architecture into a pipelined one while ensuring correct functionality and performance. The key steps involved selecting a suitable sorting algorithm, converting it into RISC-V assembly, modifying the single-cycle processor architecture, pipelining the processor, handling hazards, and evaluating the performance against the single-cycle processor.

2. Methodology

Part 1: Algorithm Selection and Assembly Conversion

We began by selecting the Bubble Sort algorithm due to its simplicity and ease of implementation. We converted the pseudocode of Bubble Sort into RISC-V assembly language instructions. After verifying the correctness of the assembly code on a simulator, we proceeded to integrate it with the single-cycle processor architecture developed in Lab 11. This involved minor modifications to the processor to support the sorting algorithm instructions like introducing a branch unit and using load and store for words instead of doublewords, thus making it a 32-bit processor.

```
#BUBBLE SORT RISC CODE:
li x5, 2
li x6, 47
li x7, 6
li x8, 19
li x9, 9
li x10, 0x100 #base address of array
li x11,5 #size of array
sw x5, 0(x10)
sw x6, 4(x10)
sw x7, 8(x10)
sw x8, 12(x10)
sw x9, 16(x10)
addi x18, x0, 0 #j=0
forloop2:
bge x18, x11, exit #if outer loop shud continue
addi x18, x18, 1 # j = j+1
li x12, 0 #LOAD i=0, inner loop initialising right before it starts
forloop1:
bge x12, x11, exitt
add x13, x12, x12 #x13=i*4 for offset
add x13, x13, x12
add x13, x13, x12
add x13, x13, x10 #offset calculation
lw x14, 0(x13) #i
lw x15, 4(x13) #i+1
bge x14, x15, no_swap #direct increment
#swapping here: (pehle x15 was 4 k saath and x14 was 0 ke saath)
sw x15, 0(x13)
sw x14, 4(x13)
no_swap:
addi x12, x12, 1 #direct increment
beq x0, x0, forloop1
exitt: #only loop 1 ended, now we go back to loop 2
beq x0, x0, forloop2
exit:
```

Fig.1: Bubble Sort Algorithm RISC V Code.

File Edit View

```
#BUBBLE SORT BASIC CODE:
addi x5, x0, 2
addi x6, x0, 47
addi x7, x0, 6
addi x8, x0, 19
addi x9, x0, 9
addi x10, x0, 256
addi x11, x0, 5

sw x5, 0(x10)
sw x6, 4(x10)
sw x7, 8(x10)
sw x8, 12(x10)
sw x9, 16(x10)

addi x18,x0,0
bge x18 x11 64
addi x18 x18 1
addi x12 x0 0
bge x12 x11 48
add x13 x12 x12
add x13 x13 x12
add x13 x13 x12
add x13 x13 x10
lw x14 0(x13)
lw x15 4(x13)
bge x14 x15 12
sw x15 0(x13)
sw x14 4(x13)
addi x12 x12 1
beq x0 x0 -44
beq x0 x0 -60
```

Fig.2: Bubble Sort Algorithm BASIC Code.

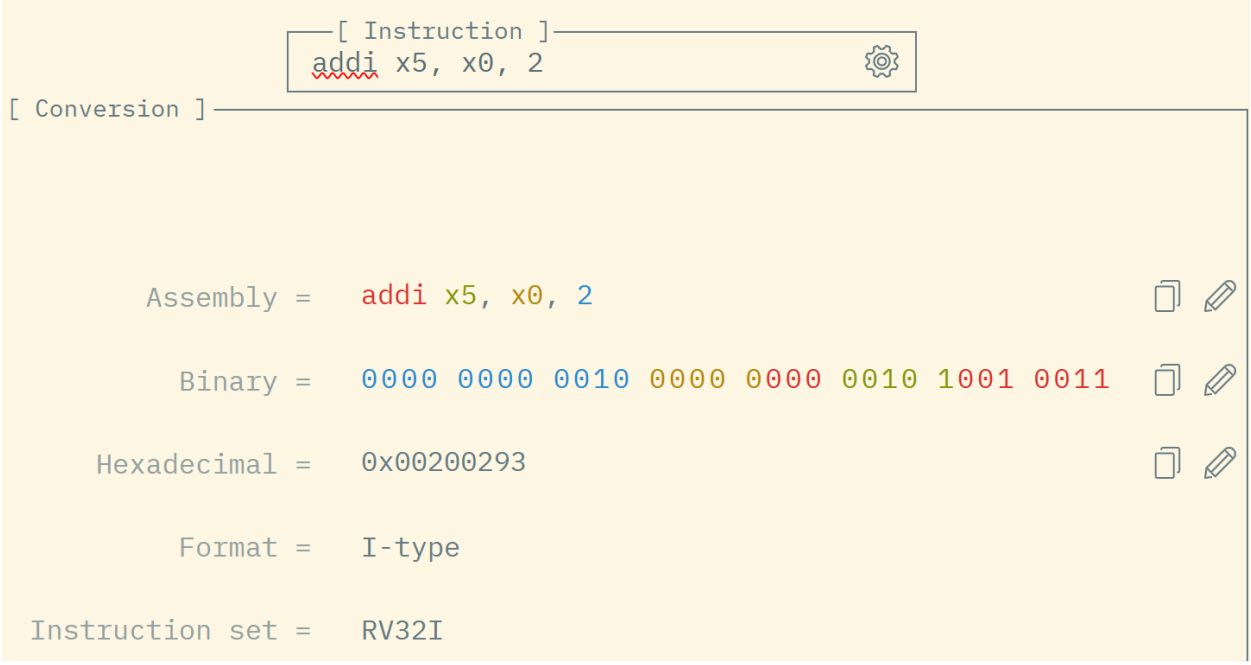

Fig.3: An Example of Instruction Decoder from BASIC to Machine Code.

Part 2: Pipelined Processor Design

The next step was to convert the single-cycle processor into a pipelined one with 5 stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). We modified the processor accordingly, ensuring proper data and control flow between stages. Each stage was tested separately to verify correct execution.

Part 3: Hazard Detection and Handling

To ensure correct pipelining, we introduced circuitry to detect and handle hazards, including data hazards, control hazards, and structural hazards. We implemented a Data Hazard Detection unit, which stalls any instruction causing a hazard. This was implemented to our fullest capability, but Task3 still had some errors in the Load and Store instructions, and hence the sorted list wasn't being stored back into the data memory properly.

Part 4: Performance Comparison

Finally, we compared the performance of the pipelined processor with the single-cycle processor in terms of execution time for sorting large arrays. We conducted simulations with varying array sizes to assess the impact of pipelining on performance.

3. **Results**

a. Task1: This task was mainly based on Lab11, which integrated all the modules made throughout the labs. We only introduced a Branch Unit, which would do the register comparisons and determine the next instruction to be executed based on the comparison.

```
module beq_bge(
   input [2:0] funct3,
   input [63:0] rd1,
   input [63:0] rd2,
   output reg a
  );

  initial
    begin
      a = 0;
    end

  always @(*)
    begin
      case (funct3)
        3'b000:
          begin
            if (rd1 == rd2)
              a = 1;
            else
              a = 0;
            end
         3'b100:
            begin
              if (rd1 < rd2)
              a = 1;
            else
              a = 0;
```

Fig.4: Branch Unit.

```
always @(*)
  begin
    case (funct3)
      3'b000:
        begin
          if (rd1 == rd2)
            a = 1;
          else
            a = 0;
          end
       3'b100:
          begin
            if (rd1 < rd2)
            a = 1;
          else
            a = 0;
          end
      3'b101:
        begin
          if (rd1 > rd2)
          a = 1;
         else
            a = 0;
        end
     endcase
    end
endmodule
```

Fig.5: Branch Unit (cont'd).

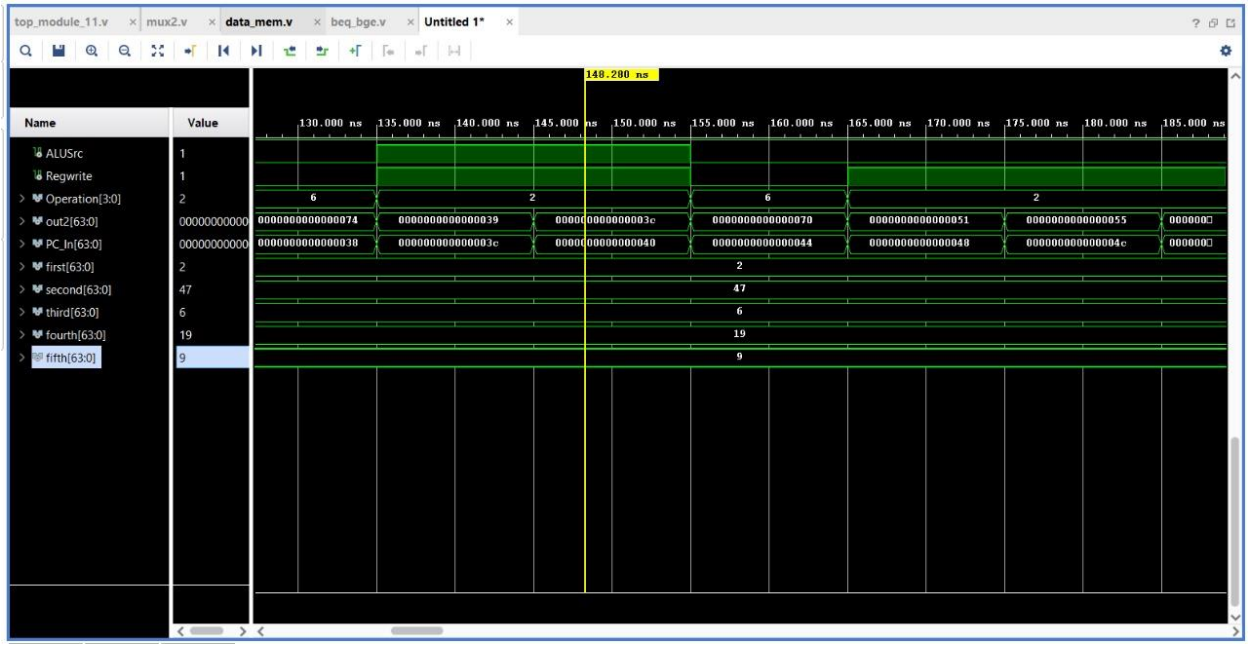

Fig.6: Unsorted Array via Non-pipelined Processor.

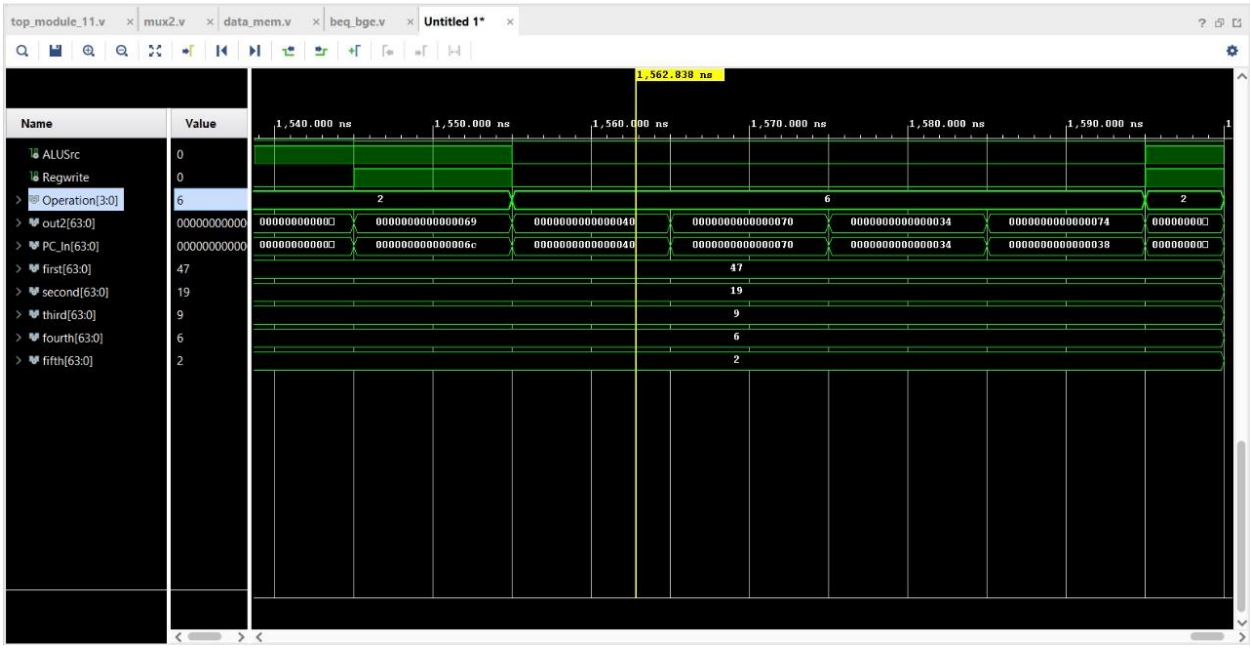

Fig.7: Sorted Array via Non-pipelined Processor.

b. Task2: For this task, we introduced 4 different modules (Fig. 8-11) initializing registers essential for pipelining, as indicated in Fig.13: IF/ID, ID/EX, EX/MEM, MEM/WRB. As expected, it executed a single instruction correctly, but not more, due to hazards. However,

one roadblock was that only the first instruction was executing and the second wasn't even going into the desired cycle. But after reaching out to our RA(Maham) and debugging ourselves, we figured out the problem was the absence of clock's positive edge in one of our newly created modules. Precisely, this task was also successfully running by the end.

```
module IF_ID(
    input clk,
    input [63:0] PC_Out,
    input [31:0] Instruction,
    output reg [63:0] IFID_PC_Out,
    output reg [31:0] IFID_Instruction
    );
always @ (posedge clk) begin
        IFID_Instruction = Instruction;
        IFID_PC_Out = PC_Out;
    end
endmodule
```

Fig.8: IF/ID Module.

```
module ID_EX(
    input clk, reset,
    input Branch, MemRead, MemWrite, MemtoReg, RegWrite, ALUSrc,
    input [1:0] ALUOp,
    input [3:0] Funct,
    input [4:0] RS1, RS2, RD,
    input [63:0] IFID_PC_Out, ReadData1, ReadData2, Imm,
    output reg IDEX_Branch, IDEX_MemRead, IDEX_MemWrite, IDEX_MemtoReg, IDEX_RegWrite, IDEX_ALUSrc,
    output reg [1:0] IDEX_ALUOp,
    output reg [3:0] IDEX_Funct,
    output reg [4:0] IDEX_RS1, IDEX_RS2, IDEX_RD,
    output reg [63:0] IDEX_PC_Out, IDEX_ReadData1, IDEX_ReadData2, IDEX_Imm
    );
always @(posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
        IDEX_Branch = 0; IDEX_MemRead = 0; IDEX_MemWrite = 0;
        IDEX_MemtoReg = 0; IDEX_RegWrite = 0; IDEX_ALUSrc = 0;
        IDEX_ALUOp = 0; IDEX_Funct = 0;
        IDEX_RS1 = 0; IDEX_RS2 = 0; IDEX_RD = 0;
        IDEX_PC_Out = 0; IDEX_ReadData1 = 0; IDEX_ReadData2 = 0; IDEX_Imm = 0;
    end
    else begin
        IDEX_Branch = Branch; IDEX_MemRead = MemRead; IDEX_MemWrite = MemWrite;
        IDEX_MemtoReg = MemtoReg; IDEX_RegWrite = RegWrite; IDEX_ALUSrc = ALUSrc;
        IDEX_ALUOp = ALUOp; IDEX_Funct = Funct;
        IDEX_RS1 = RS1; IDEX_RS2 = RS2; IDEX_RD = RD;
        IDEX_PC_Out = IFID_PC_Out; IDEX_ReadData1 = ReadData1; IDEX_ReadData2 = ReadData2; IDEX_Imm = Imm;
    end
end
endmodule
```

Fig.9: ID/EX Module.

```
module EX_MEM(
    input clk, reset,
    input Branch, MemRead, MemWrite, MemtoReg, RegWrite,
    input Zero,
    input [4:0] RD,
    input [63:0] Adder2Out, Result, WriteData,
    output reg EM_Branch, EM_MemRead, EM_MemWrite, EM_MemtoReg, EM_RegWrite,
    output reg EM_Zero,
    output reg [4:0] EM_RD,
    output reg [63:0] EM_Adder2Out, EM_Result, EM_WriteData
    );
always @(posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
        EM_Branch = 0; EM_MemRead = 0; EM_MemWrite = 0; EM_MemtoReg = 0;
        EM_RegWrite = 0; EM_Zero = 0; EM_RD = 0; EM_Adder2Out = 0;
        EM_Result = 0; EM_WriteData = 0;
    end
    else begin
        EM_Branch = Branch; EM_MemRead = MemRead; EM_MemWrite = MemWrite;
        EM_MemtoReg = MemtoReg; EM_RegWrite = RegWrite; EM_Zero = Zero;
        EM_RD = RD; EM_Adder2Out = Adder2Out; EM_Result = Result;
        EM_WriteData = WriteData;
    end
end
endmodule
```

Fig.10: EX/MEM Module.

```
module MEM_WRB(
    input clk, reset,
    input MemtoReg, RegWrite,
    input [4:0] RD,
    input [63:0] EM_Result, Read_Data,
    output reg MW_MemtoReg, MW_RegWrite,
    output reg [4:0] MW_RD,
    output reg [63:0] MW_Result, MW_Read_Data
    );
always @(posedge clk or posedge reset) begin
    if (reset == 1'b1) begin
        MW_MemtoReg = 0; MW_RegWrite = 0; MW_RD = 0;
        MW_Result = 0; MW_Read_Data = 0;
    end
    else begin
        MW_MemtoReg = MemtoReg; MW_RegWrite = RegWrite; MW_RD = RD;
        MW_Result = EM_Result; MW_Read_Data = Read_Data;
    end
end
endmodule
```

Fig.11: MEM/WB Module.

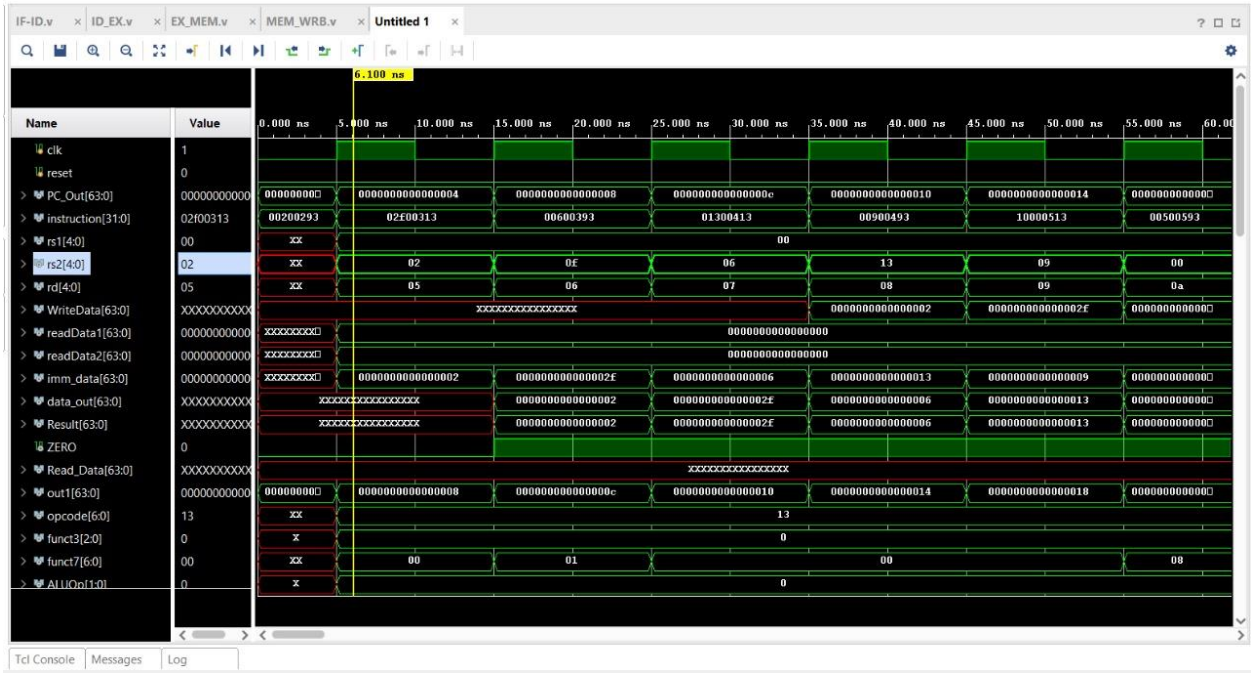

Fig.12: Sample Instructions/ Testcases Executing on Pipelined Processor Confirming Correct Implementation.

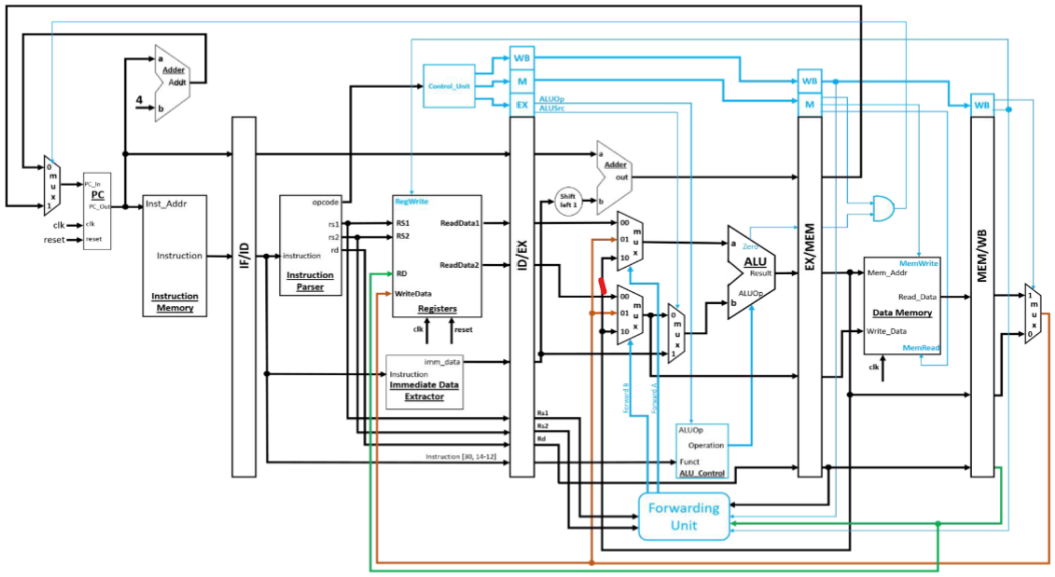

Figure 1: A subset of Pipeline RISC-V processor with forwarding functionality only. Some wires are shown in color just to help a user to distinguish wires easily (specifically between IF/ID and ID/EX stages).

3

Fig.13: Datapath for Pipelining.

c. Task3: This task was probably the most time-taking one, as we did it a bit differently than the datapath given above. Rather than introducing a whole Forwarding Unit as a new module, we found it easier to modify the ALU_64_bit module to meet the requirements, which we successfully did. Moreover, we added a Hazard Detection Module, which introduces stalls wherever needed. However, the task did not run successfully with the intended output but did follow most of the datapath. The reason we figured it wasn't working was because the load and store instructions weren't executing properly, which could be due to some overlooking of the code. We're confident that we could've figured it out if we had more time to debug.

```
module hazards(
    input [4:0] RS1, RS2, IDEX_RD,
    input IDEX_MemRead,
    output reg stall // IFID_Write, PC_Write, IDEX_MuxOut,
    );

always @(*) begin
    if (IDEX_MemRead && (IDEX_RD == RS1 || IDEX_RD == RS2)) begin
        stall = 1; // IDEX_MuxOut = 1; IFID_Write = 0; PC_Write = 0;
    end
    else begin
        stall = 0; // IDEX_MuxOut = 0; IFID_Write = 1; PC_Write = 1;
    end
end
endmodule
```

Fig.14: Hazard Detection Unit.

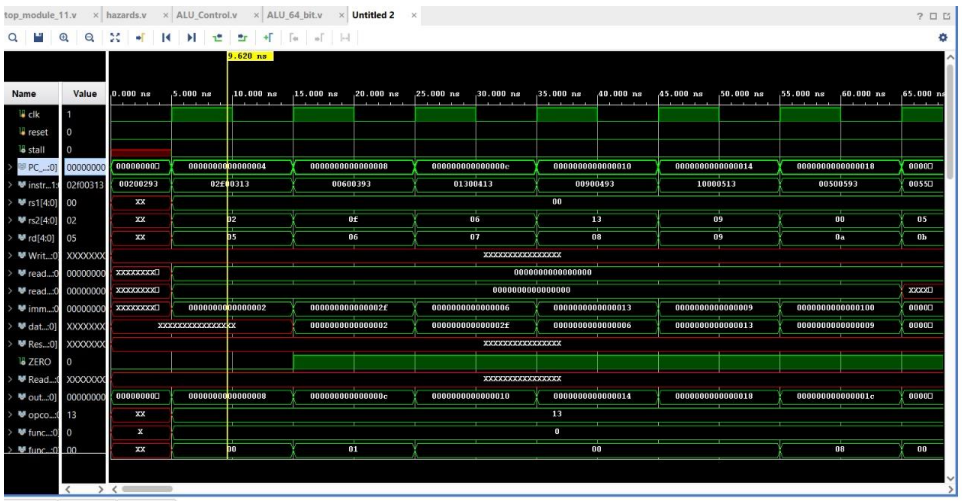

Fig.15: Simulation of Task3.

d. Task4: Even though our task3 was not working to the fullest, we were still able to analyze some performance time theoretically. The non-pipelined processor took 4280 nanoseconds in total, each cycle running for 10 nanoseconds. Hence, the whole algorithm of Bubble Sort took 428 Cycles on Non-pipelined processor.

Each cycle takes 10 nanoseconds.

The longest stage latency (MEM) is 3 cycles.

Average number of stall cycles per instruction is 1 cycle.

Number of pipeline stages: 5

Number of instructions for Bubble Sort: 29 instructions

Total cycles for pipelined processor = (Number of instructions) * (Number of stages) + (Number of stall cycles)
Total cycles for pipelined processor = 29 * 5 + 29 * 1
= 174
So, the estimated total number of cycles for the Bubble Sort algorithm on the pipelined processor with 5 stages is approximately 174 cycles theoretically.

After completing the steps, we conducted final simulations to evaluate the functionality and performance of the pipelined processor executing the Bubble Sort algorithm. The results for Task1 and 2 demonstrated that the pipelined processor successfully sorted arrays and executed instructions. Task3, however, was still simulating the end of clock cycles, but not storing values. Still, it exhibited improved performance compared to the single-cycle processor.

4. **Challenges**

Throughout the project, we encountered several challenges, including:

- Introducing and debugging the branch unit's functioning correctly.
- Designing the pipelined processor architecture while ensuring proper synchronization and hazard handling.
- Debugging issues related to data hazards and pipeline stalls.

Some of these challenges were solved through repeated testing and some help from the RAs, but one or two remain.

5. **Conclusion**

Overall, our project was (almost) successful in designing and implementing a pipelined processor capable of executing a sorting algorithm efficiently. By converting the single-cycle processor into a pipelined architecture and implementing hazard detection and handling techniques, we achieved improved performance compared to the single-cycle counterpart. The project provided valuable insights into processor design, pipelining, and performance optimization.

6. **References**

- Lab 11
- CA Textbook
- Project Guidelines Document