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

RISC-V Pipelined Processor

Hiba Aiman Ali (09269)
Iffat Fatima (09270)
Javeria Nadeem (09271)
Bushra Sadaf (09267)

Dr. Waseem Hasan, Ms. Maham Tabassum

29th November 2024

Contents

Introduction

1.1 Objective

The main goal of this project was to make a 5-stage pipelined processor capable of running an array sorting algorithm. To achieve this, we will need to convert the single-cycle processor into a pipelined processor. The purpose is to improve the performance of a single-cycle processor by implementing pipelining and mitigating hazards through stalls and forwarding. Finally, we will compare the performance of both processors by calculating the speed-up in terms of execution time of both processors.

Methodology

1.1.1 Sorting Algorithm

We used the following Bubble Sort code to test our processor functionality:

```
li x10  0x100    # Base address of array, pointing to *a
li x11  5         # Length of the array
# Initializing array a=[1,3,5,4,2] and storing at base address x10
li x25 1
sw x25 0(x10)
li x26 3
sw x26 4(x10)
li x27 5
sw x27 8(x10)
li x28 4
sw x28 12(x10)
li x29 2
sw x29 16(x10)
bubble:
beq x10 x0 exit    # If *a==null, exit
beq x11 x0 exit    # If length==0, exit
li x12 0           # i=0 initialization
loop_1:
bge x12 x11 exit   # If i>= length, exit outer-loop
add x13 x12 x0     # j=i
loop_2:
bge x13 x11 exit_inner  # If j>=length, exit inner loop
slli x7 x12 2      # x7 = x12*4 for 4 byte int
add x7 x10 x7      # Add the offset(x7) and base address stored in x10 and store in x7
lw x5 0(x7)        # x5 = a[i]
slli x16 x13 2     # x16 = x13*4 for 4 byte int
add x16 x10  x16   # Add the offset(x16) and base address stored in x10 and store in x16
lw x24 0(x16)      # x24=a[j]
bge x5 x24 not_swap  # If a[i]>=a[j], skip swapping
mv x14 x5           # Temporary storage: tmp = x14 = a[i]
sw x24 0(x7)        # a[i] = a[j]
sw x14 0(x16)       # a[j] = temp
not_swap:
addi x13,x13,1      # j++ (incrementing j)
j loop_2            # Jump to inner loop
exit_inner:         # Exit the inner loop
addi x12, x12, 1    # i++ (incrementing i)
j loop_1            # Jump to outer loop
exit:
```

Figure 1.1: Bubble Sort in Assembly

0x00000110	02	00	00	00
0x0000010c	04	00	00	00
0x00000108	05	00	00	00
0x00000104	03	00	00	00
0x00000100	01	00	00	00

Figure 1.2: Before Sorting

0x00000110	01	00	00	00
0x0000010c	02	00	00	00
0x00000108	03	00	00	00
0x00000104	04	00	00	00
0x00000100	05	00	00	00

Figure 1.3: After Sorting

Tasks

2.1 Task 1

2.1.1 Single-Cycled RISC V Implementation

We used the code of Single-Cycle Processor that we wrote in Lab 11 and modified it for the new instructions in our bubble sort code. We also had to fix it because it was not working properly at the time of submission, but they were fixed by tracing back the modules and fixing them.

2.1.2 Changes made

For slli instructions, we changed our existing ALU64 and ALU_Control with new cases for slli instructions. We also created a new Branch_Unit which supports beq and blt instructions. This new module handles the Zero control bit according to the type of branch instruction.

Furthermore, we also initialized some values in our Data_Memory to test our sorting algorithm. Finally, we called the Branch_Unit in our Top, connected all the appropriate wires and ran the Single Cycle to sort the array.

```
addi x11 x0  5         # Length of the array

beq x11 x0 exit    # If length==0, exit
addi x12 x0 0           # i=0 initialization

loop_1:
bgt x12 x11 exit   # If i>= length, exit outer-loop
beq x12, x11, exit
add x13 x12 x0     # j=i

loop_2:
bgt x13 x11 exit_inner  # If j>=length, exit inner loop
beq x13 x11 exit_inner

slli x7 x12 3      # x7 = x12*8 for 8 byte int
ld x5, 0(x7)         # x5 = a[i]

slli x16 x13 3     # x16 = x13*8 for 8 byte int
ld x24, 0(x16)       # x24=a[j]
blt x5 x24 not_swap  # If a[i]<=a[j], skip swapping
beq x5 x24 not_swap

addi x14 x5 0           # Temporary storage: tmp = x14 = a[i]
# Swapping:
sd x24 0(x7)         # a[i] = a[j]
sd x14 0(x16)        # a[j] = temp

not_swap:
addi x13,x13,1      # j++ (incrementing j)
beq x0 x0 loop_2           # Jump to inner loop
exit_inner:         # Exit the inner loop
addi x12, x12, 1    # i++ (incrementing i)
beq x0 x0 loop_1           # Jump to outer loop
exit:
```

Figure 2.1: Bubble sort that we used in our instruction memory (ld and sd giving errors since it is not supported by Venus Simulator)

2.1.3 Code and Waveform (Single Cycle Waveform)

Code can be found in the GitHub link attached in the Appendix.

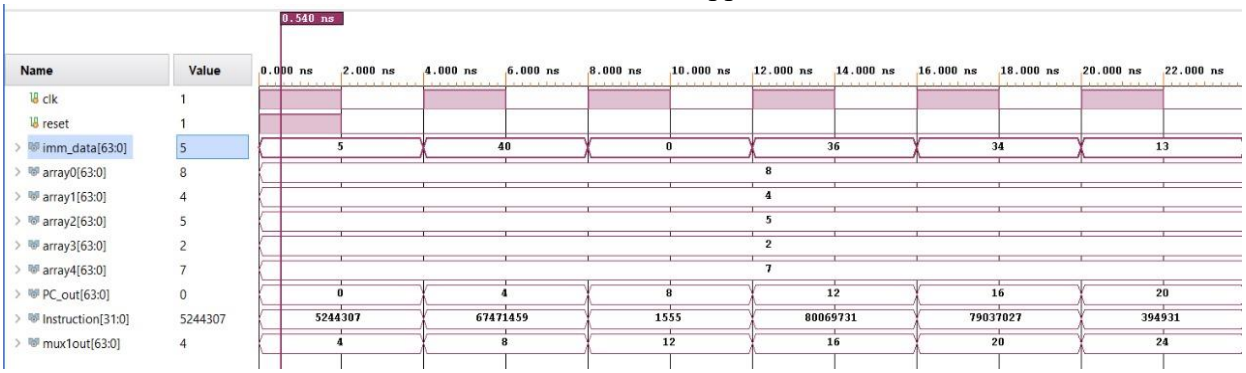

Figure 2.2: Before Sorting

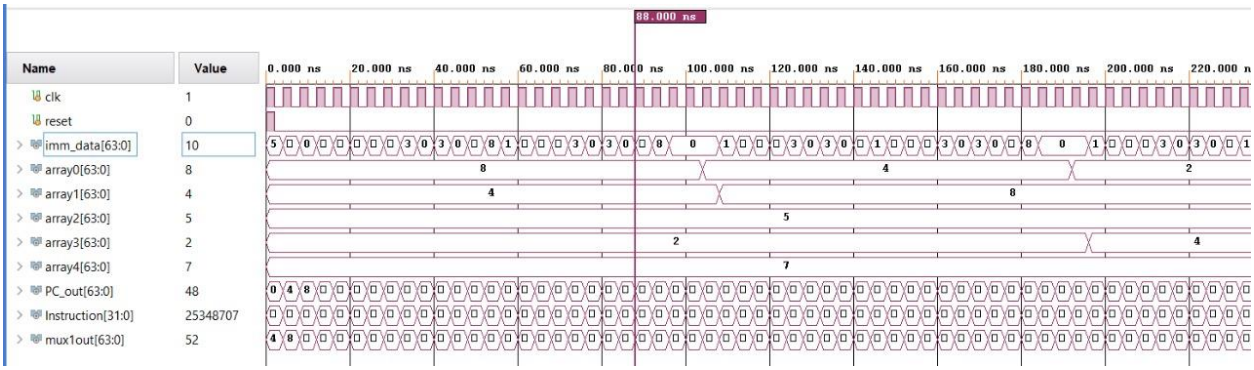

Figure 2.3: Process of Sorting (shows each iteration of the loop and how each value is being swapped until the whole array is swapped at 716 ns. This completes the bubble sort code in our Single-Cycle Processor)

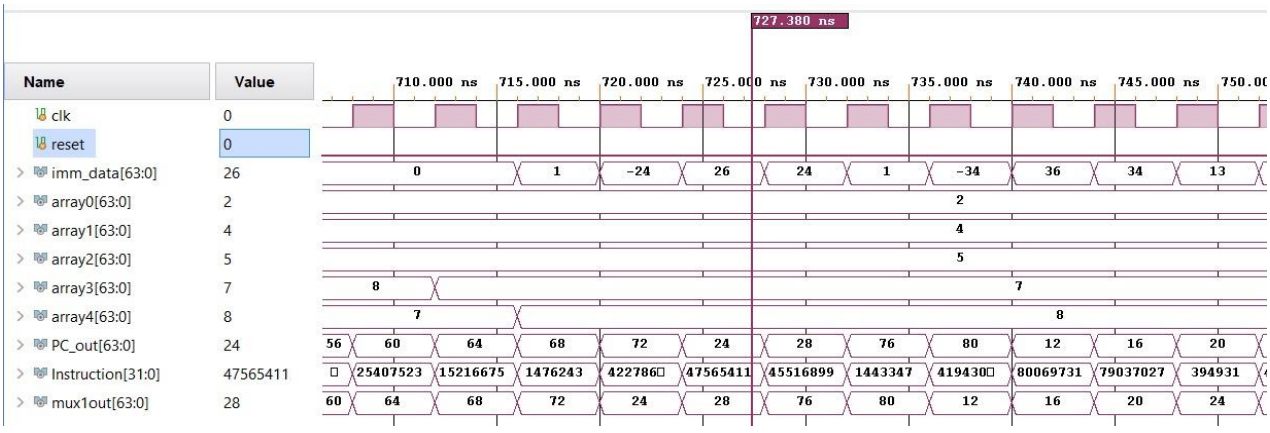

Figure 2.4: After Sorting

2. 2 Task 2

2.2.1 Testing 5-Stage/Pipelined RISC V Processor for a Single Instruction

For task 2 our goal was to shift from a Single-Cycled Processor to a Pipelined one. We know the Pipelined Processor has five stages and utilizes four pipeline registers. To implement this, we created four new modules, one for each register.

2.2.2 Changes made

The following modules were created for implementing pipelining:

- IF_ID
- ID_EX
- EX_MEM
- MEM_WB

After creating the four pipeline registers, we called the modules in the Top module and added appropriate connections for these registers.

2.2.3 Code and Testing

Code can be found in the GitHub link attached in the Appendix.

We tested all different types of instructions to validate whether the pipelined processor is working as expected or not.

Tested instructions: add, sub, ld, sd, slli, beq, addi and blt.

```
integer i;
initial begin
for (i = 0 ; i < 31 ; i = i + 1)
    Registers[i] = 0; //initializing values as 0

Registers[12]=64'd12;
Registers[13]=64'd13;
Registers[14]=64'd14;
Registers[15]=64'd5;

end
assign r2=Registers[12];
assign r3=Registers[13];
assign r4=Registers[14];
assign r5=Registers[15];
```

Figure 2.5: Initialized the following values in our Register_File for the following test cases

Test Case:

add x12 x13 x14
sub x12 x13 x14
slli x12 x14 3
addi x12 x0 1
ld x14 0(x0)
sd x13 0(x0)
beq x15 x15 exit
addi x15 x0 2
exit:
blt x13 x15 exit1
addi x13 x0 4
exit1:

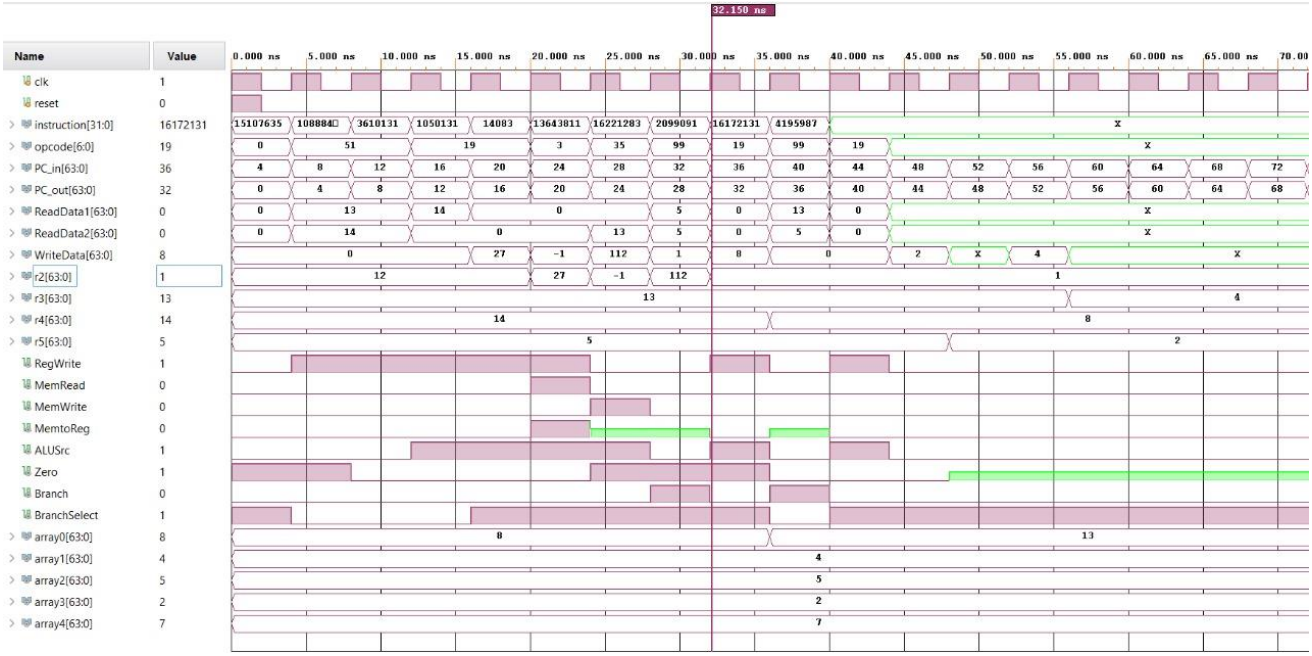

Figure 2.6: Testcases waveform

2.3 Task 3

2.3.1 Forwarding

After Pipelining the processor, we started mitigating the different types of hazards, starting from data hazards. We took the help of Figure 4.55 in our Textbook and made the appropriate connections for the new inputs of our ALU64. Forwarding required two new modules, Mux_3x1 and Forwarding_Unit which included all the conditions to output Forward_A and Forward_B. These outputs became the select lines for our 3x1 Muxes.

While working on this, we encountered a third stall problem. The first two instructions after a given instruction were being mitigated and dealt with correctly through Forwarding but the third instruction was also causing a data hazard which should not have been the case. For solving this problem, we changed the Register_File by letting it write at a negedge clk instead of posedge. This new addition ensured write before read functionality. This update altogether eliminated the third instruction stall problem.

Test cases for checking forwarding:

- add x12, x13, x14
- addi x14, x12, 4
- sub x13, x14, x12

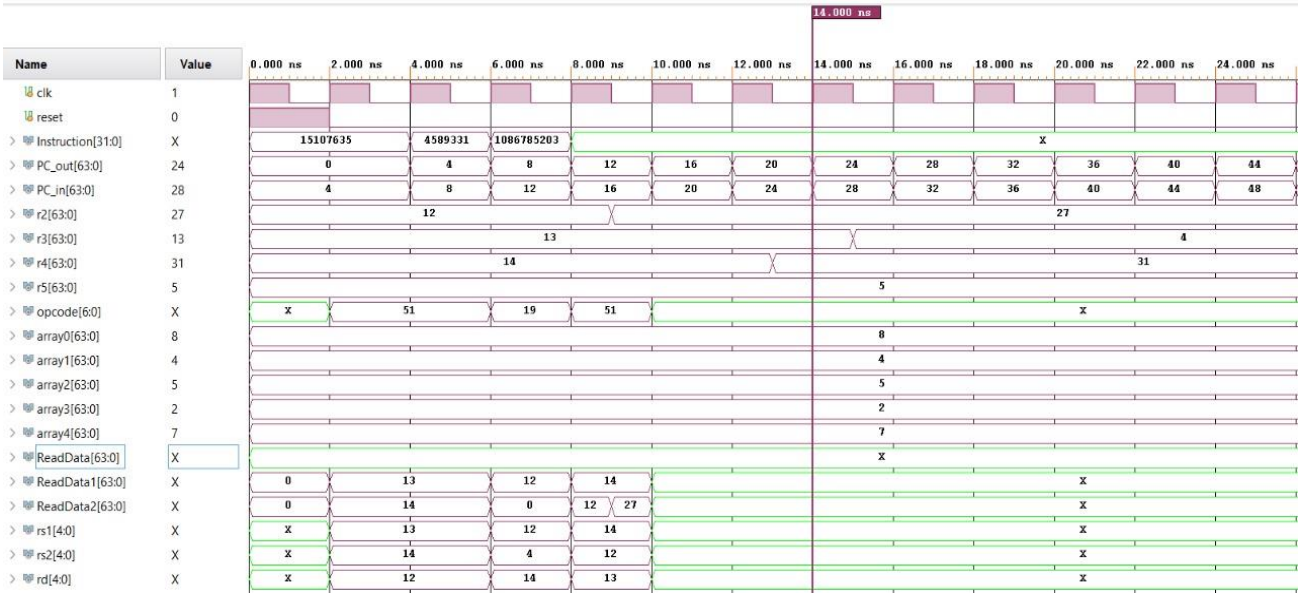

Figure 2.9: No stalls required by forwarding the values

2.3.2 Handling Data Hazards

The main goal of task 3 was to eliminate all hazards into our processor and run the final bubble sort code in the completed Pipelined Processor.

2.3.3 Changes made

Moving onto mitigating Control Hazard and Load Use Data Hazard, we implemented a Hazard_Detection Unit for stalling the Processor when required. Hazard_Detection_Unit delays the clock cycle by preserving/ holding the same values of Program_Counter and IF_ID pipeline registers when it detects a load use data hazard. It also sets all control lines to 0 which takes care of all the pipeline registers above, basically performing a NOP (no operation).

Lastly, after a few tries we were also successful in implementing Flushing in our Processor which eliminates the Control Hazard in cases of Branch instructions being taken. We calculated the Branch_selection bit through Funct[2] and gave that input to the first three pipeline registers. Finally, we tested our bubble sort code and were successful in sorting the array in the Pipelined Processor.

2.3.4 Code and Waveform

Code can be found in the GitHub link
attached in the Appendix.

Final Waveform:

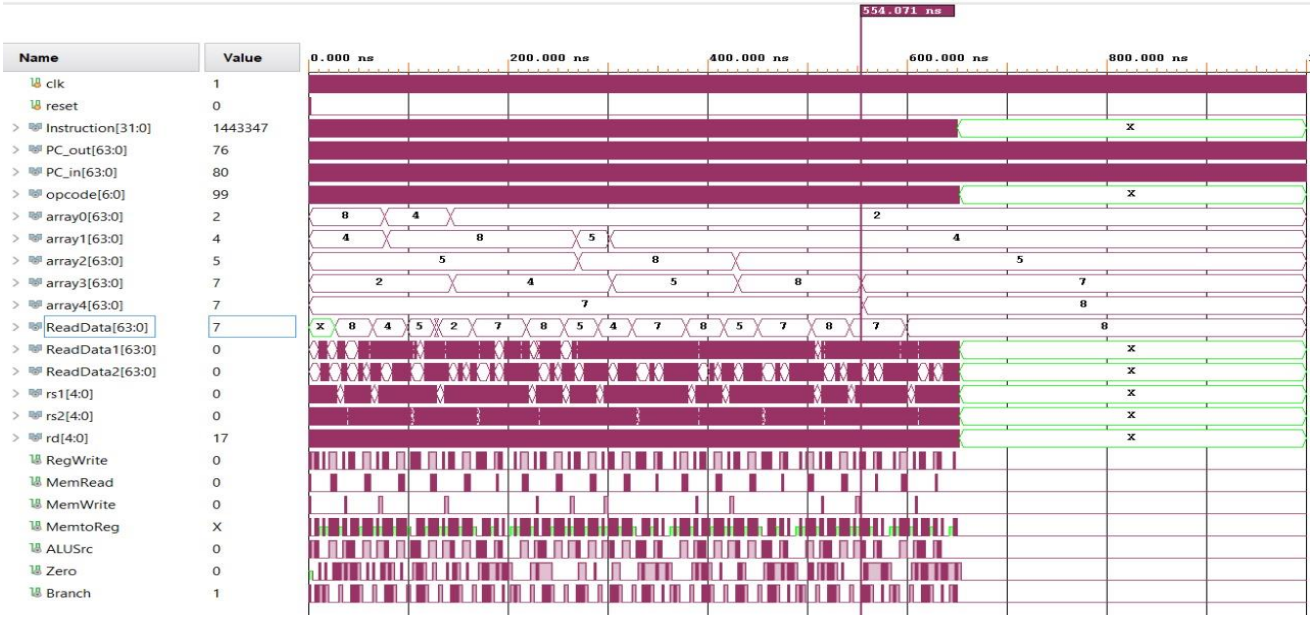

Figure 2.10: Final Waveform

2.4 Task 4

2.4.1 Performance Comparison

The final task entailed the comparison of Single Cycle and Pipeline Cycle through Clock Cycles and Execution Time. To determine the execution time for both processors, we read off the time after the data values were completely sorted which marked the end of the sorting process. The calculation we did for comparing the performance and finding out the speed up is as follows:

Single Cycle processor:
Clock Cycle Time = 2 ns
Execution Time = 716 ns
Clock Cycles = Execution time / clock cycle time
= 716 / 2
= 358 cc

Pipelined processor:
Clock Cycle Time = single cycle clock time/5
= 1 /5
= 0.2ns
Execution Time = 554ns
Clock Cycles = Execution time / clock cycle time
= 554 / 0.2
= 2770 cc

Note: (dividing by 5 since each instruction has now split up into 5 stages and takes 0.4ns each)
Speed Up:
Speed up = ExTimeSingleCycle / ExTimePipelined
= 716 / 554
= 1.29

Therefore, our Pipeline Processor is **1.29** times faster than the Single Cycle Processor.

2.4.2 Simulation Comparison

Figure 2.11: Single-Cycled execution

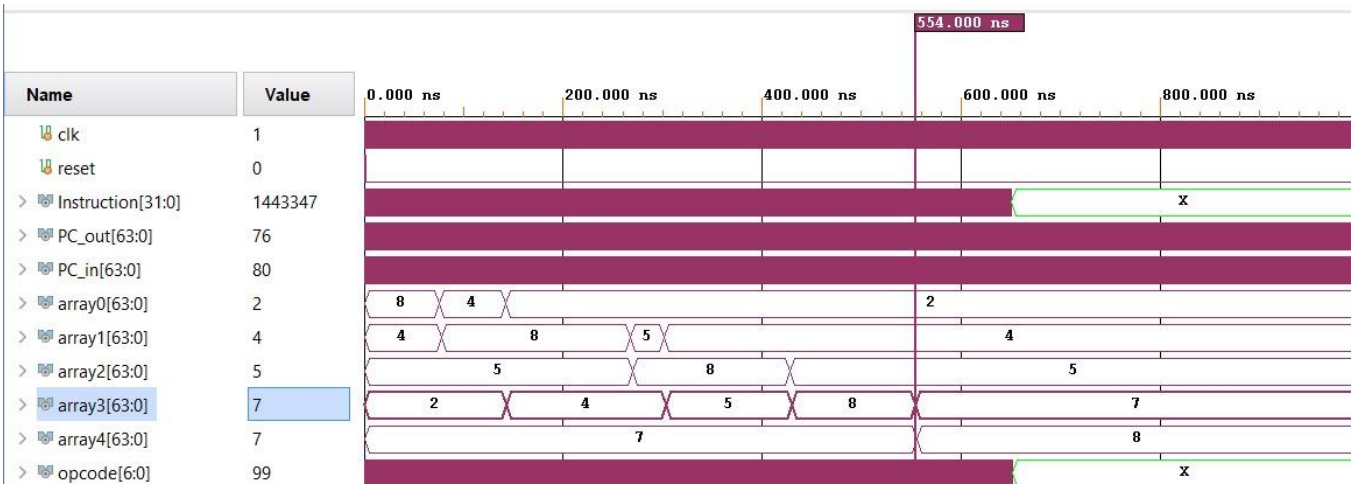

Figure 2.12: Pipeline execution time during last swap

3.1 Challenges

Task 1

One of the challenges encountered while implementing task 1 was that, we knew what algorithm to implement for sorting, but implementing it in the instruction memory was causing confusions.

Task 2

For task 2, one challenge was that we faced issues with tracing back some of the registers and case sensitivity in terms of variable naming in the pipelining process but the main issue was even

if the module was giving proper outputs instead of Xs or Zs, we did not know how to verify those outputs which led to a lot of confusion.

Task 3
For task 3, we had switched devices, which was yet another issue since there issues vivado being able to configure the file and path issues.

3.2 Task Division

The general division for this project was as follows:

- **Task 1** → Iffat and Bushra
- **Task 2** → Iffat, Bushra and Javeria
- **Task 3** → Javeria and Hiba
- **Task 4** → All
- **Report** → Hiba and Javeria

3.3 Conclusion

We were successfully able to implement RISC-V-Pipeline-Processor with forwarding and hazard detection. From task 4 we can conclude that our RISCV pipelined processor performs 1.29 times better than the single cycle processor on bubble sort algorithm.

3.4 References

- Computer Architecture course Textbook
- Digital Design and Computer Architecture RISC-V Edition (Harris, Sarah, Harris, David)
- Computer Organization and design The hardware/software interface RISC -V Edition (David A.Patterson John L. Hennesy)
- PowerPoints provided by our instructors
- Venus

3.5 Appendix

Our final code can be found in the GitHub Repository given here.