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

## RISC-V Pipelined Processor

Muhammad Taqi (08073)
Muhammad Ibad (08440)
Eman Fatima (08595)

Dr. Farhan Khan, Ms. Abeera Alam

30th April 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:

```
addi x19, x0, 5 # length of the array
label3:beq x8, x19, exit # exit the program
addi x9, x0, 0 # i = 0
addi x10, x0, 0 # j = 0
addi x6, x19, -1 # Limit for the loops
sub x6, x6, x8
label2: beq x9, x6, label
slli x7, x9, 2 # Multiply i by 4
add x7, x7, x18 # Calc addr of a[j]
lw x5, 0(x7) # Load a[j] into x19
addi x29, x9, 1
slli x28, x29, 2
add x28, x28, x18 # Calc addr of a[j+1]
lw x30, 0(x28)
blt x30, x5, swap # Branch if a[j+1] < a[j]
addi x9, x9, 1
beq x0, x0, label2
swap: addi x31, x5, 0 # swapping starts
addi x5, x30, 0
sw x5, 0(x7)
addi x30, x31, 0
sw x30, 0(x28)
addi x10, x0, 1
addi x9, x9, 1
beq x0, x0, label2 # go back to current line
label: addi x8, x8, 1
beq x10, x0, exit # exit program
beq x0, x0, label3 # go back up
exit:
```

Figure 1.1: Bubble Sort in Assembly

| | | | | |
|---|---|---|---|---|
| 0x00000010 | 01 | 00 | 00 | 00 |
| 0x0000000c | 02 | 00 | 00 | 00 |
| 0x00000008 | 03 | 00 | 00 | 00 |
| 0x00000004 | 04 | 00 | 00 | 00 |
| 0x00000000 | 05 | 00 | 00 | 00 |

Figure 1.2: Before Sorting

| | | | | |
|---|---|---|---|---|
| 0x00000010 | 05 | 00 | 00 | 00 |
| 0x0000000c | 04 | 00 | 00 | 00 |
| 0x00000008 | 03 | 00 | 00 | 00 |
| 0x00000004 | 02 | 00 | 00 | 00 |
| 0x00000000 | 01 | 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. Mainly, we added support for slli and blt instruction as these were the only new instructions in our code which weren't supported in our existing code.

### 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.

We also converted our previous bubble sort code into hex, so that it could be used in our processor. The code is initialized in Instruction_Memory through the following Python script.

```
lst = [ ''' list of encoded instructions ''']

new = [] ; instruction = 1 ; count = 0

for i in lst:
    new.append("{inst_mem["+str(count + 3)+"], inst_mem["+str(count + 2)+"],
    inst_mem["+str(count + 1)+"], inst_mem["+str(count)+"]} = 32'h"+str(i)[2:]
    +f"; //{instruction}")

    instruction += 1
    count += 4

for i in new:
    print(i) # Output: Verilog Code for Instruction Memory
```

Figure 2.1: Python script for generating encoded instructions

### 2.1.3 Code and Waveform (Single Cycle Waveform)

The code for task 1 can be found here

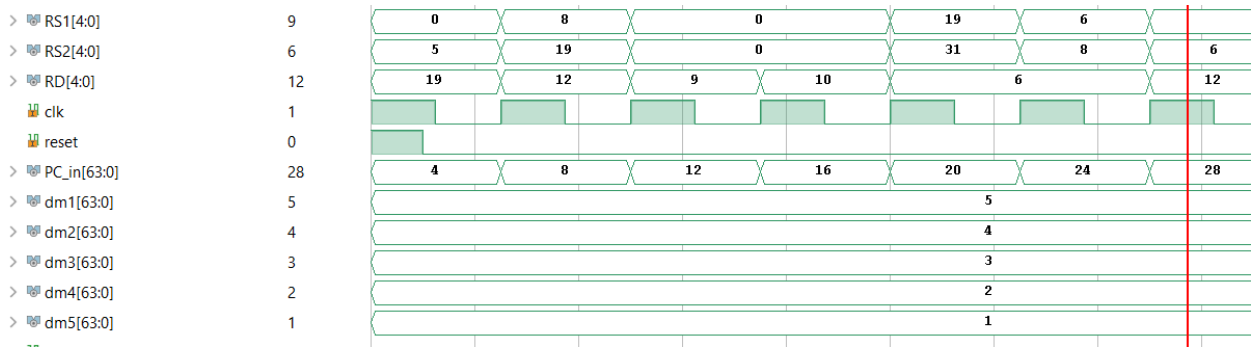


Figure 2.2: Before Sorting

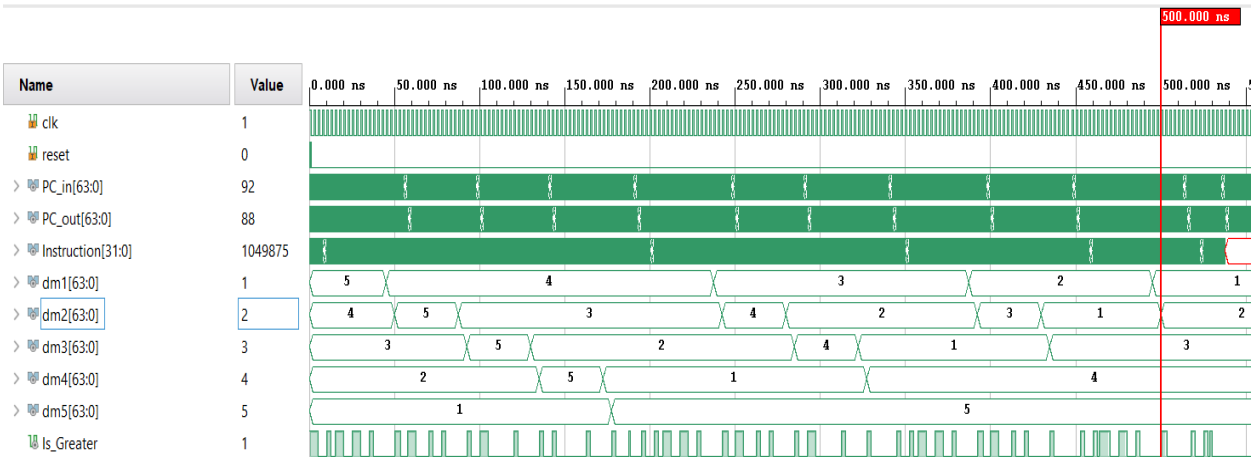


Figure 2.3: Process of Sorting (shows each iteration of the loop and how each value is being swapped until the whole array is swapped at 500ns. This completes the bubble sort code in our Single-Cycle Processor)

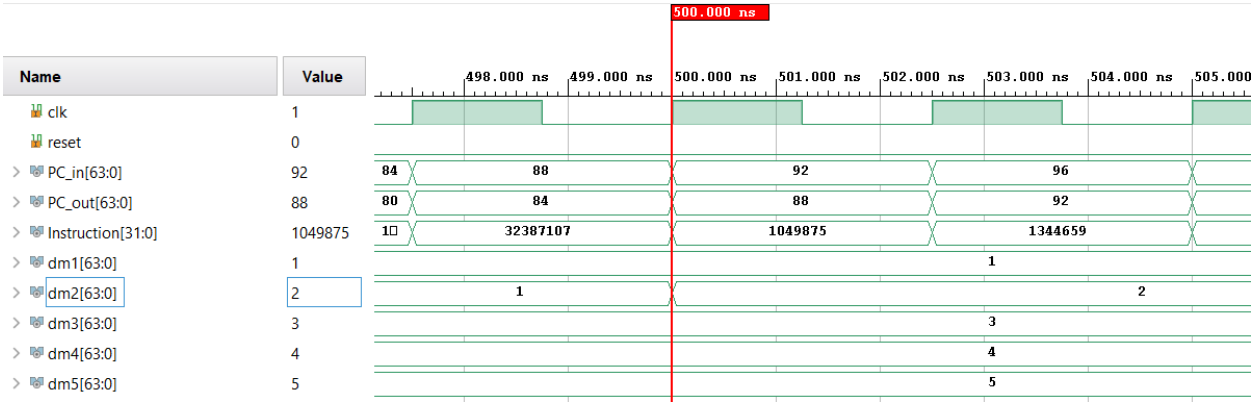


Figure 2.4: After Sorting

# 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

The code for task 2 can be found here.

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.

```
30        integer k;
31        initial
32        begin
33        for (k = 0 ; k < 31 ; k = k + 1)
34            Registers[k] = 0;
35        Registers[18] = 64'd10;
36        Registers[20] = 64'd5;
37        Registers[21] = 64'd1;
38        end
```

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

## Test Cases:

**Test case 1**: add x20, x18, x21

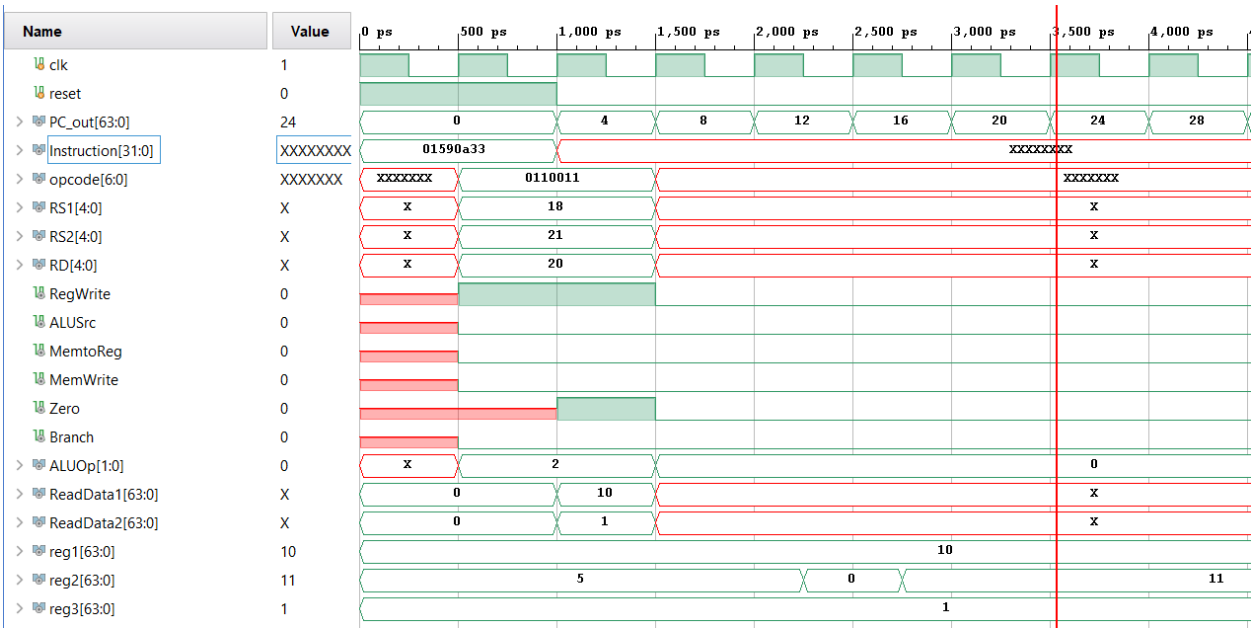


Figure 2.6: testcase 1

**Test case 2:** slli x20, x18, 3


Figure 2.7: testcase 2

**Test case 3:** beq x2, x2, 104 // Taken

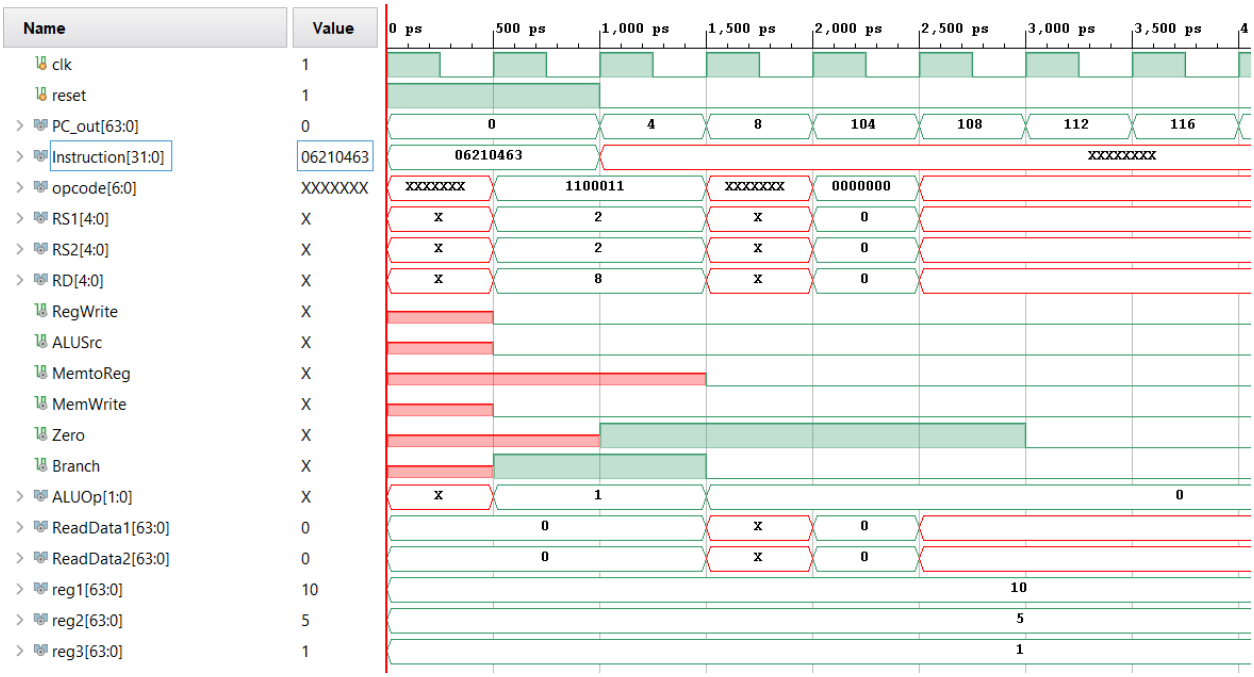


Figure 2.8: testcase 3

## 2.2.4 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 appropiate 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:**

- addi x18, x18, 10
- add x20, x18, x21
- sub x18, x20, x18

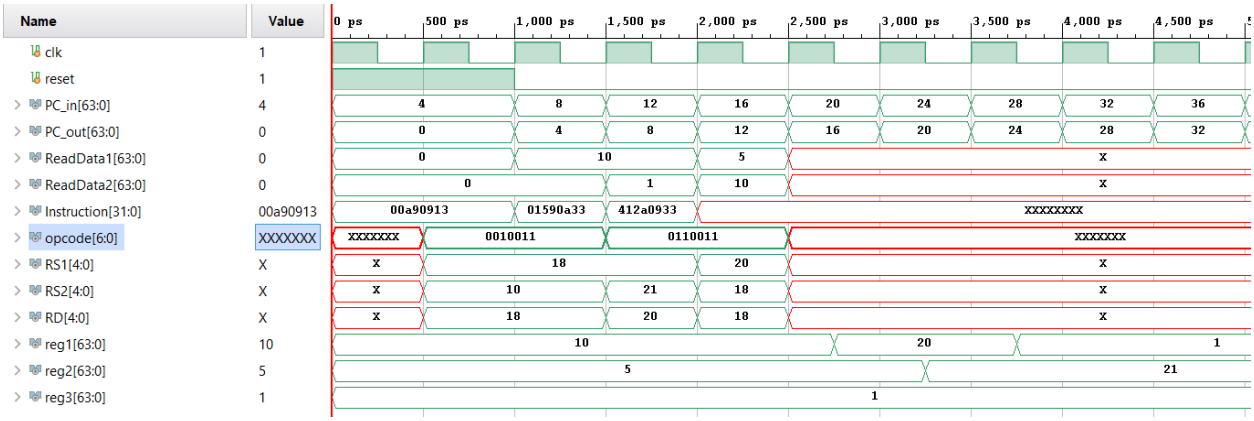


Figure 2.9: No stalls required by forwarding the values

# 2.3 Task 3

## 2.3.1 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.

## 2.3.2 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 Final_Branch through Funct[3] 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.

## 2.3.3 Code and Waveform

The code for task 3 can be found here.
Final Waveform:

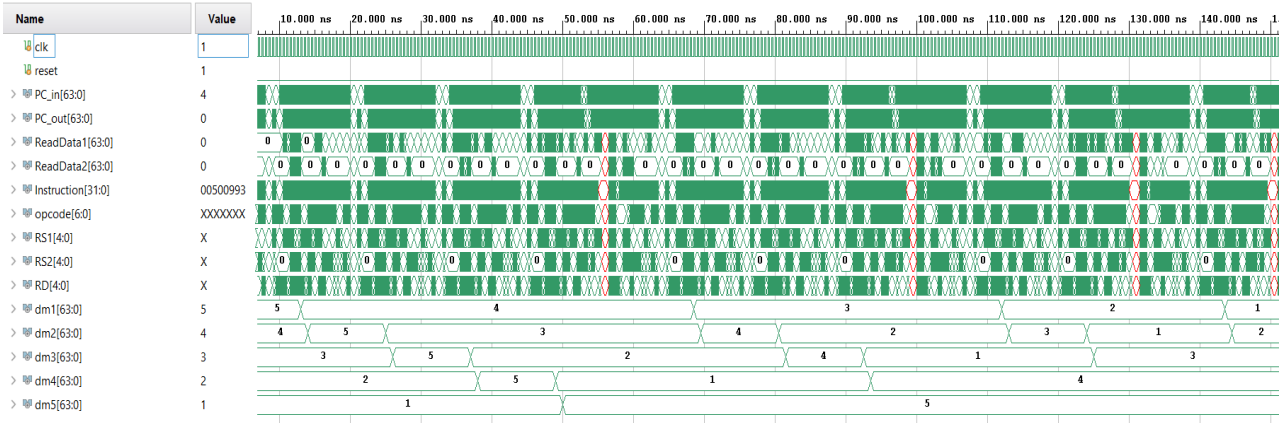


Figure 2.10: Final Waveform

# 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 = 2.5ns
Execution Time = 500ns
Clock Cycles = Execution time / clock cycle time
= 500 / 2.5
= 200cc

**Pipelined processor:**
Clock Cycle Time = single cycle clock time/5
= 2.5 /5
= 0.5ns
Execution Time = 145ns
Clock Cycles = Execution time / clock cycle time
= 145 / 0.5
= 290cc

Note: (dividing by 5 since each instruction has now split up into 5 stages and takes 0.5ns each)
**Speed Up:**
Speed up = ExTimeSingleCycle / ExTimePipelined
= 500 / 145
= 3.448

Therefore, our Pipeline Processor is **3.448** times faster than the Single Cycle Processor.

## 2.4.2 Simulation Comparison

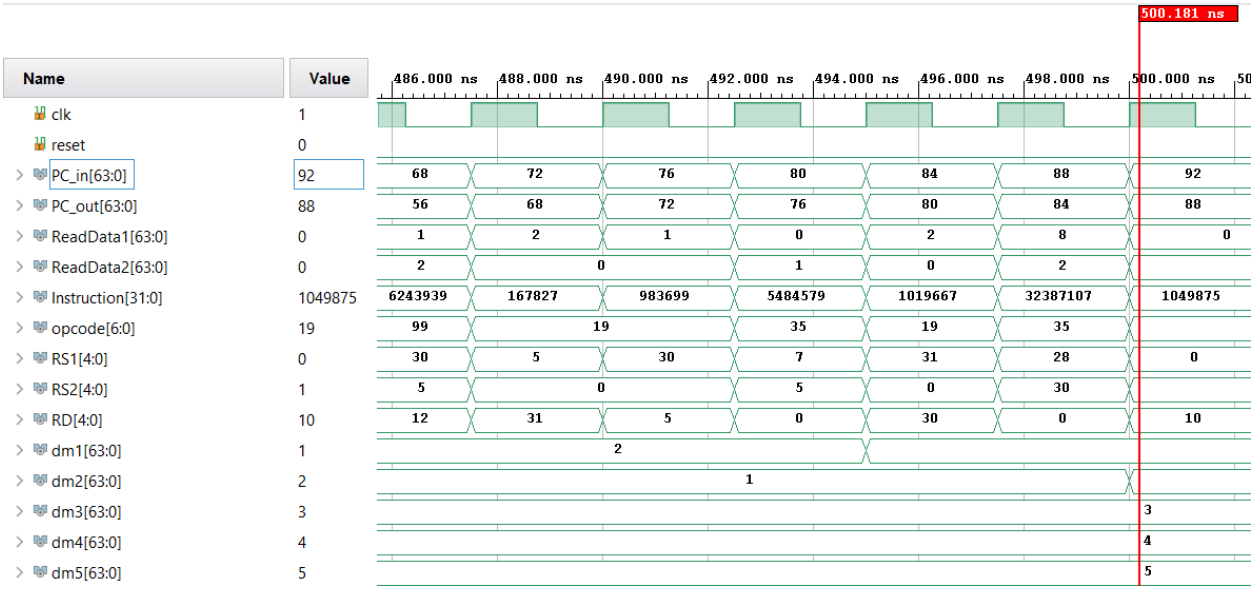


Figure 2.11: Single-Cycled execution time during the last swap

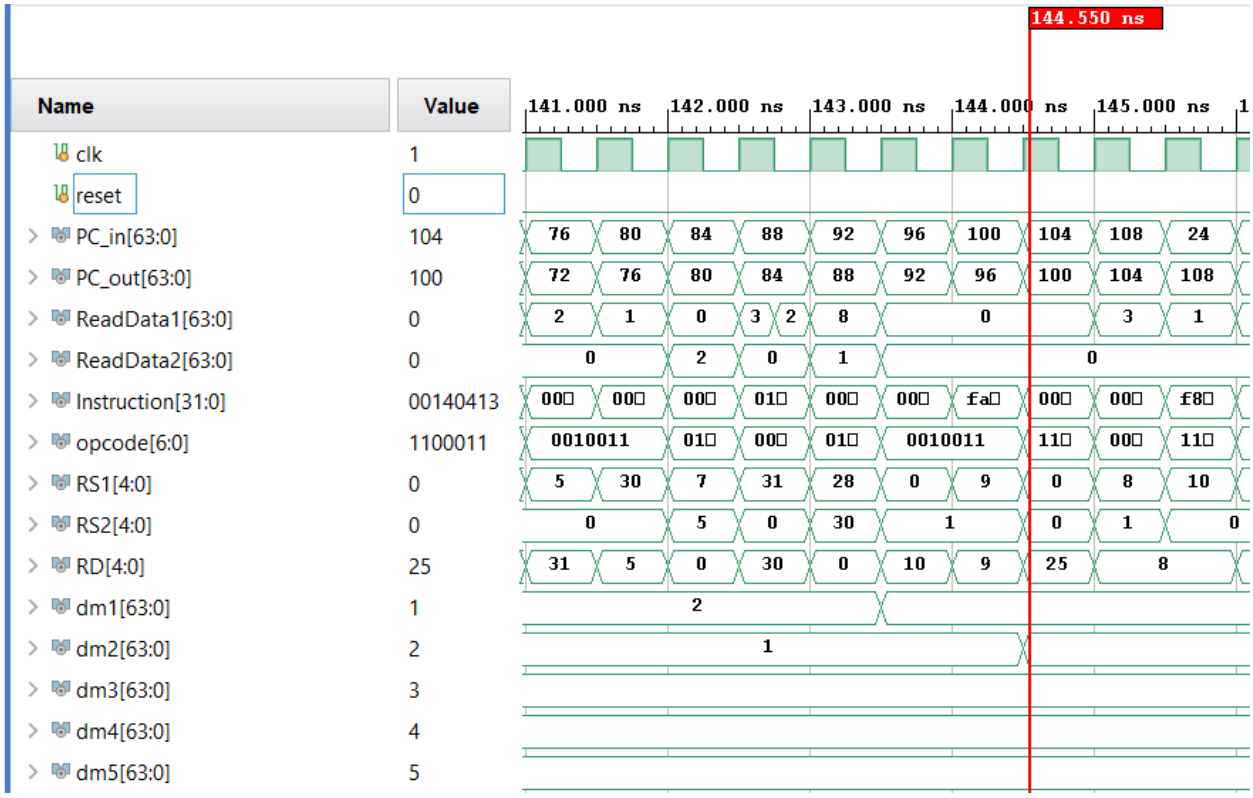


Figure 2.12: Pipeline execution time during the last swap

# 3.1 Challenges

**Task 1**

One of the challenges encountered while implementing task 1 was that, since Venus Simulator does not support double word instructions, we had to modify the code and do its encoding accordingly to run the load and store double word instructions.

**Task 2**

For task 2, while checking if each instruction was working as expected we noticed that MemtoReg was not working as intended and was giving us a Z value. After spending a considerable amount of time debugging our code, we found out that the problem arose due to naming one of the variables incorrectly. For debugging purposes, we checked each MemtoReg signal after it passed through each Pipeline Register and checked if it contained the same data as the previous Register.

**Task 3**

For mitigating control and load use data hazards we implemented a Hazard_Detection_Unit successfully. However, since Vivado kept crashing continuously during the testing phase, we had to restart our progress **twice**. We also had a hard time figuring out how to implement Flushing into our Processor as there were no clear references regarding it. However, after much debugging and theory revising, we successfully implemented flushing too.

## 3.2 Task Division

The general division for this project was as follows:

- **Task 1** → Taqi
- **Task 2** → Eman
- **Task 3** → Ibad
- **Task 4** → Ibad / Eman
- **Report** → Taqi

## 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 3.4 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
- Encoding simulator
- Venus


# 3.5 Appendix

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