# Computer Architecture
# CE/CS - 321/330

## Final Project Report

### RISC-V Processor

Maha ozair - mo08695
Muhammad Affan - ma08910
Haris Hussain Khan - hk09521

22 April 2025

# Contents

# Introduction

## 1.1 Objective

To build a 5-stage pipelined processor capable of executing any array sorting algorithm on an array consisting of atleast 7 elements, we will need to convert a single-cycle processor into a pipelined one. We will do this by adding pipeline registers to our single cycle implementation we made in lab 11, then incorporate a hazard detection unit and forwarding unit to eliminate dependencies between instructions.

### 1.1.1 Sorting Algorithm

We will be using bubble sort to test our processor functionality. Complete Project (RISC V Processor on Github)

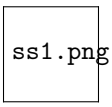


Figure 1.1: Registers and Memory

# Tasks

## 2.1 Task1

### 2.1.1 RISC V Implementation (single cycle)

We implemented the converted algorithm on the RISC V single-cycle processor developed in our lab. To make sure the algorithm worked on our processer we added the encoded instructions in the instruction memory and made a 7 element array using instructions in the instruction memory. Specifically, we added a Branch Unit module to make beq and blt commands used in the algorithm. These changes enabled us to run the algorithm on our processor.

### 2.1.2 Changes

We added the Branch Unit module and added the funct 3's of blt and beq so that their functionality can be added, the module outputs a branchselect signal which is anded with branch signal from Control Unit and sent to MUX module to handle next PC input. Moreover, we added the function for slli ALU Control. Further, we altered the data memory to accomodate lw and sw commands instead of ld and sd, without changing the inner structure too much. We initialised all values to 0 in Data Memory module so as to make it function properly. The array was initialised using machine code instructions in the instruction memory.

### 2.1.3 Code and Waveform

The link for the code can be found here.

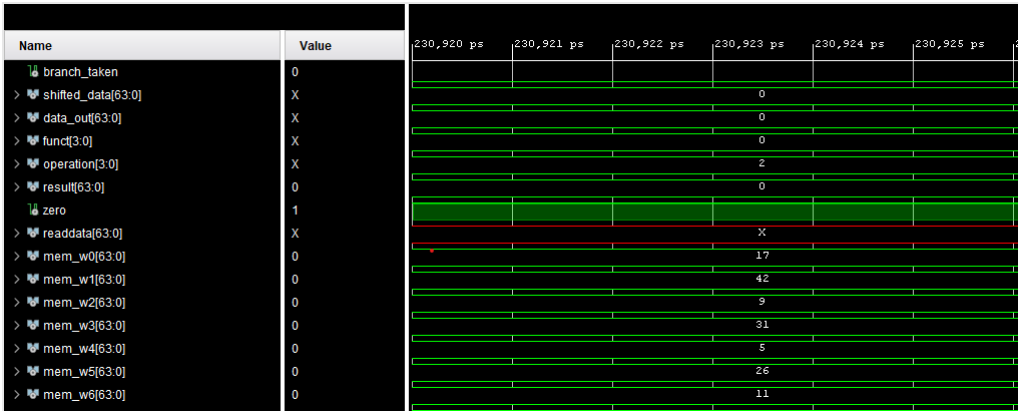


Figure 2.1: Single Cycle Simulation

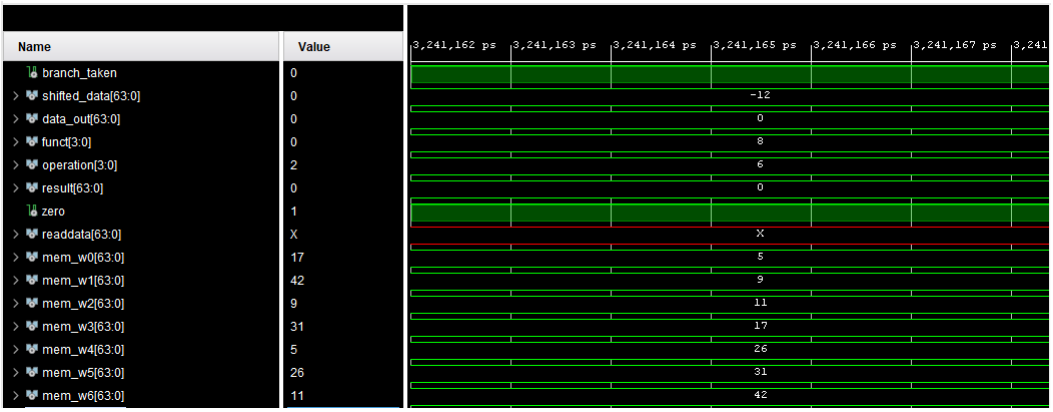


Figure 2.2: Single Cycle Simulation

### 2.1.4 results

By making some changes we were able to sort our array= [5,9,11,17,26,31,42], using bubble sort. These screenshots show the implementation of it across multiple clock cycles.

## 2.2 Task2

### 2.2.1 Testing 5-Stage/Pipelined RISC V Processor for Sorting Algorithm

In this task we upgraded our single cycle processor to a pipelined processor by adding pipeline registers between 5 stages.

### 2.2.2 Changes

To implement a pipelined design, we added 4 extra modules IF_ID,ID_EX,EX_MEM,MEM_WB to simulate the pipeline registers in our processor. These registers pass signals and data from one stage of processing to another using the intermediate registers, enabling multiple instructions to be executed simultaneously without data hazards.

### 2.2.3 Code

The link for the code can be found here. We tested this task by using multiple test cases, one for each type of instruction (I-type, R-type, SB-type)

### 2.2.4 Testing R-Type instructions

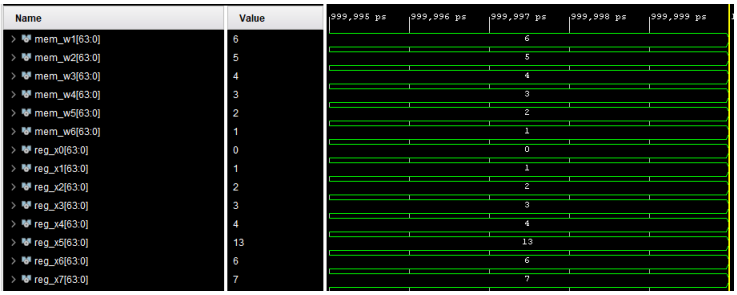


Figure 2.3: instruction: add x5,x6,x7

### 2.2.5 Testing I-Type instructions

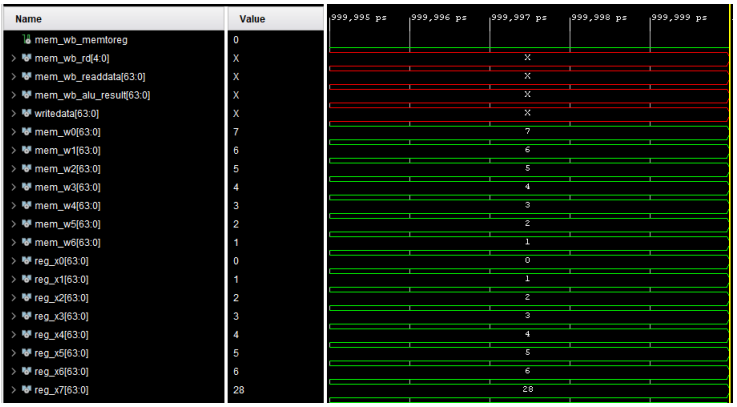


Figure 2.4: instruction: slli x7,x7,2

### 2.2.6 Testing SB-Type instructions (branch was taken)

observe the changes in pc in, it indicates jump in pc value according to immediate calculated.

### 2.2.7 results

By installing pipeline registers we were able to simultaneously run and check all types of instructions.

```
beq x0,x0,exit
add x1,x2,x3
add x4,x4,x5
slli x7,x3,4
addi x2,x3,4
slli x5,x3,4
exit:
addi x6,x6,5
```

Figure 2.5: code for testing

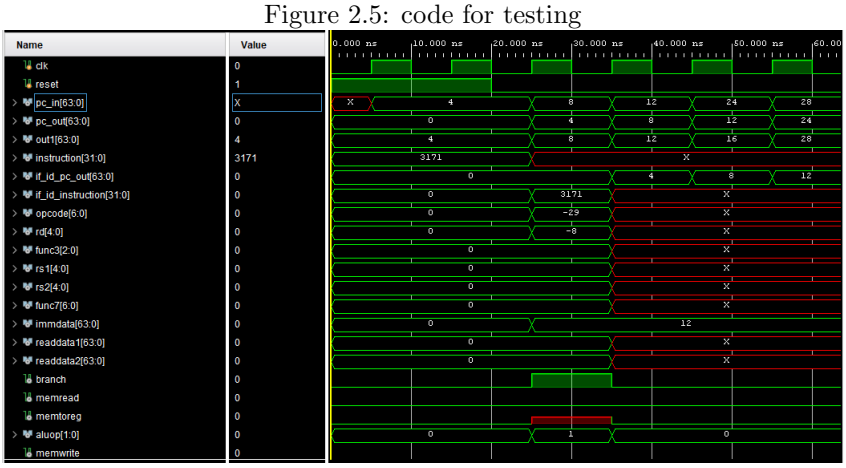


Figure 2.6: waveform

## 2.3 Task 3

This task requires us to remove all hazards to ensure a smooth pipelined processor.

### 2.3.1 Changes

In this task we created a hazard detection unit and forwarding unit to solve data hazards and install stalls according to dependencies.

### 2.3.2 Forwarding Unit

To address hazards in our circuitry and make sure instructions are accessing the right data values, we need to develop modules for providing updated values at the right time. The detection modules will identify where forwarding is required. Once detected, we can use the forwarding unit to actually perform the forwarding by using the appropriate pipeline registers to forward appropriate values.

### 2.3.3 Waveform for all 5 test cases

The link to the code can be found here.

### 2.3.4 Hazard Detection Unit

To address hazards in our circuitry that occur due to dependency between instructions, we need to develop modules for detecting hazards and stalling the pipeline. The detection modules will

identify where hazards are occurring and include a stall accordingly.

### 2.3.5 Results

This task enabled us to get rid of the data hazards and dependencies by adding Hazard Detection module and Forwarding unit to make sure appropriate data is fetched from the pipeline registers when instructions run simultaneously, and to add stalls wherever necessary by turning all control signals to zero once a hazard is detected.

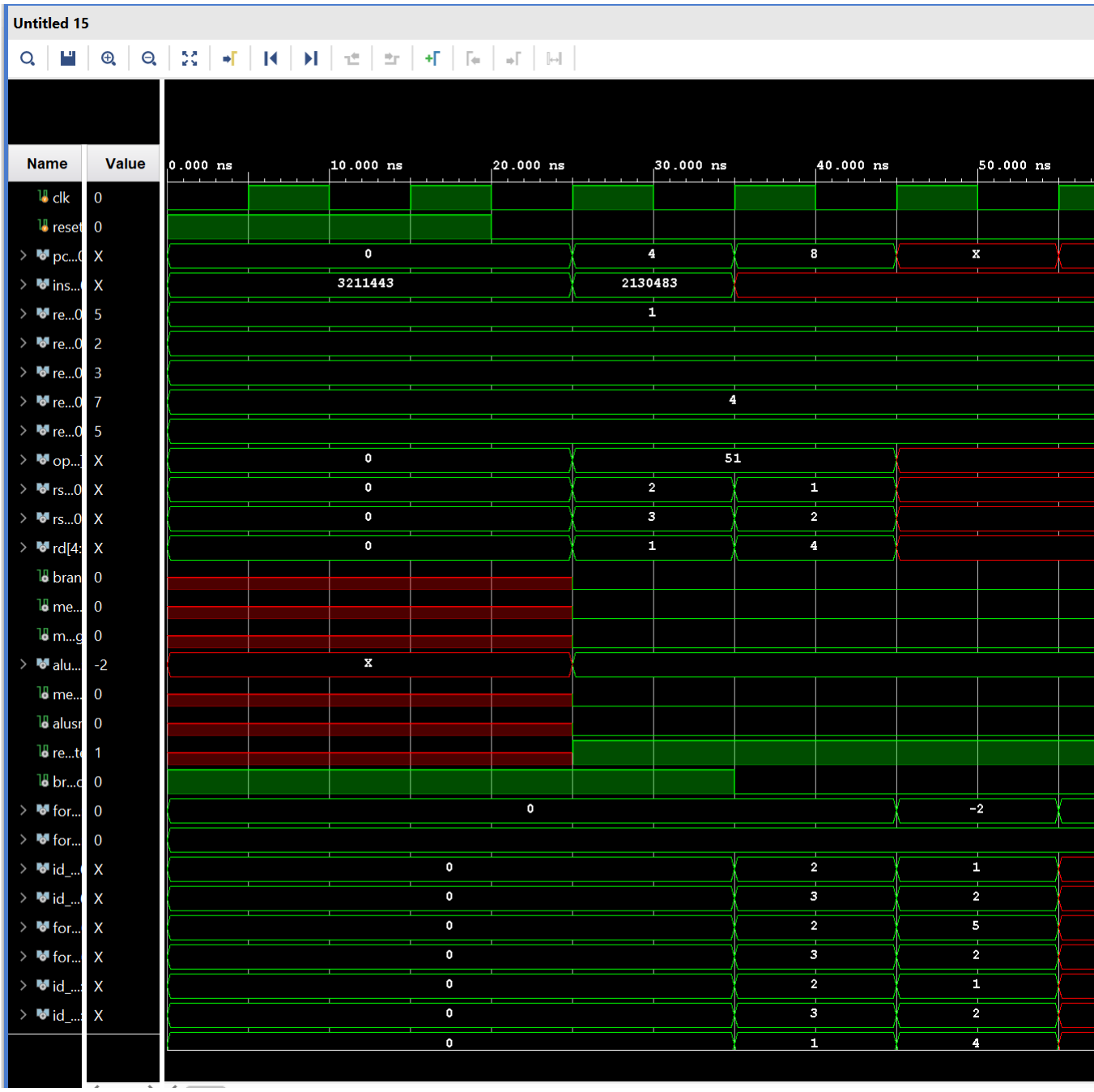


Figure 2.7: Test Case 1: forward A 10

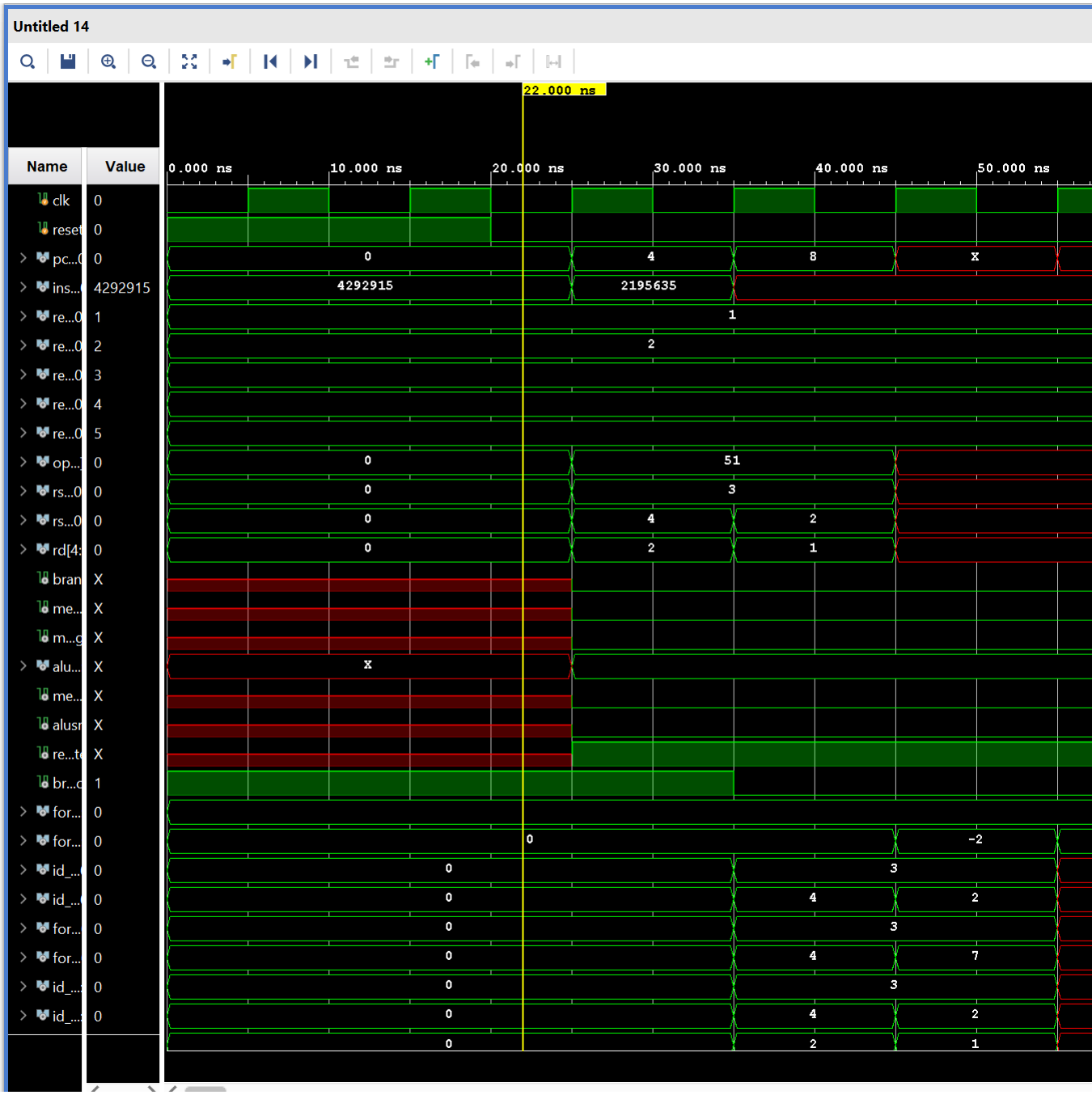


Figure 2.8: Test Case 2: forward B 10

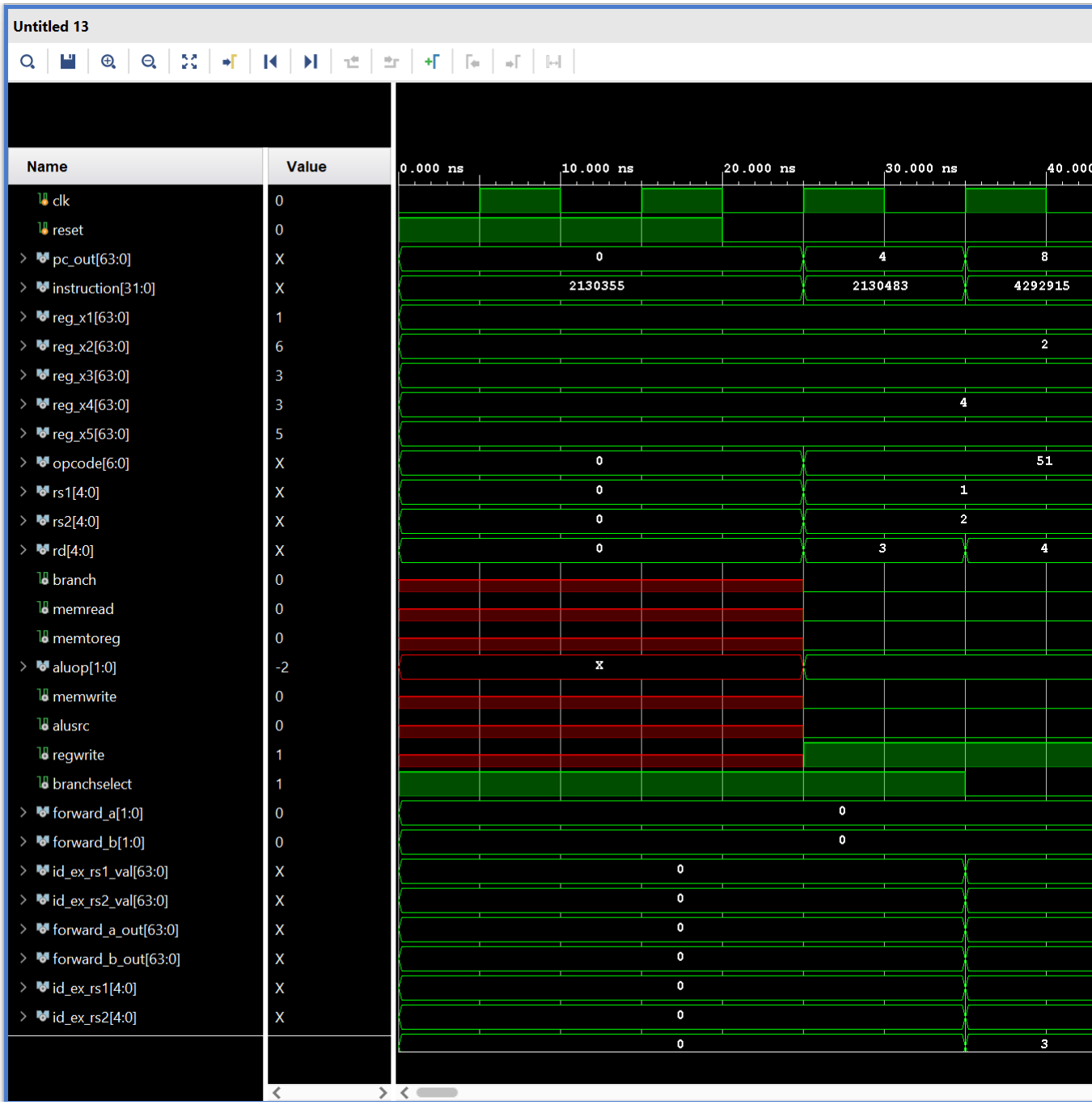


Figure 2.9: Test Case 3: forward A 01

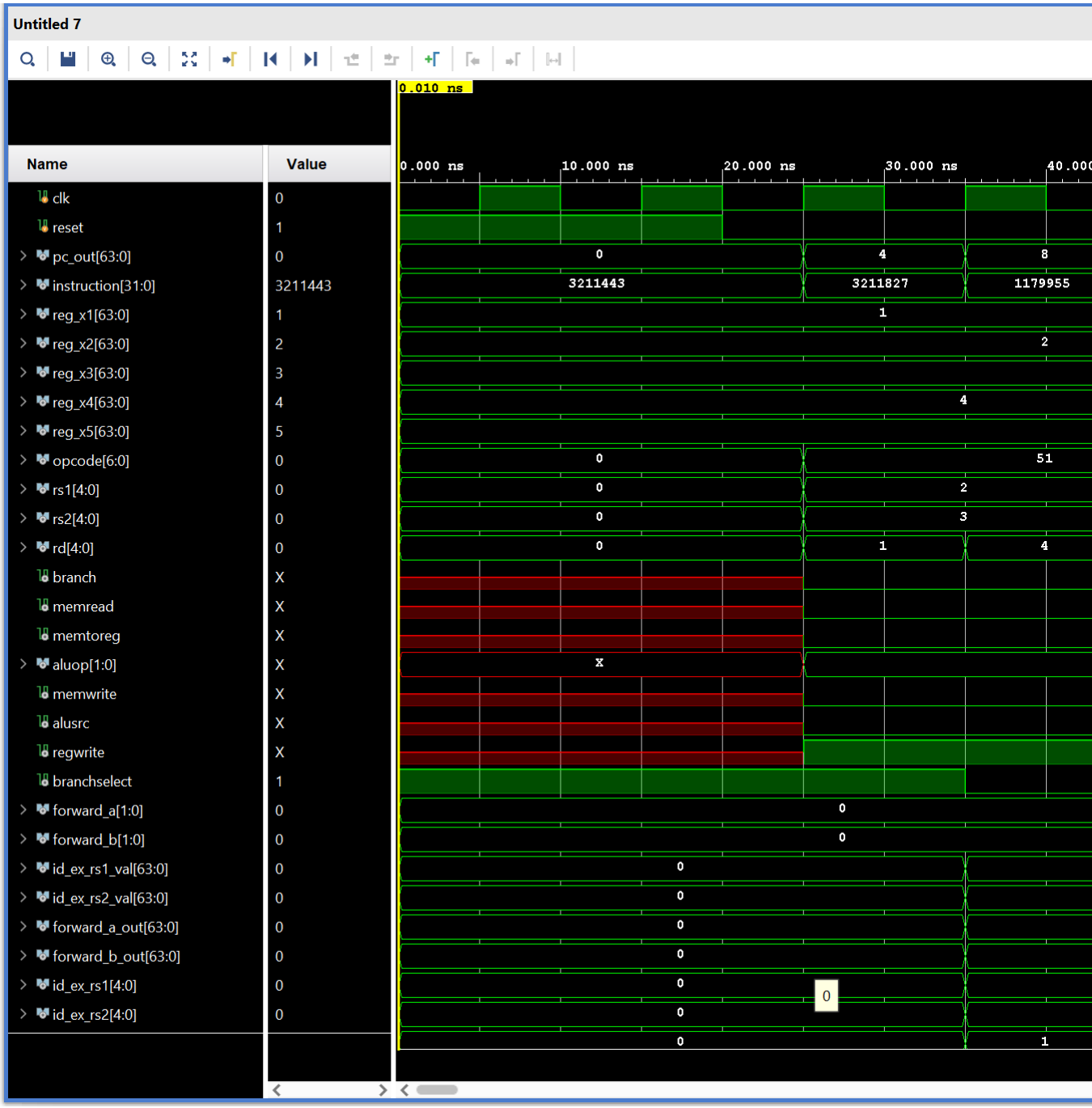


Figure 2.10: Test Case 4: forward B 01

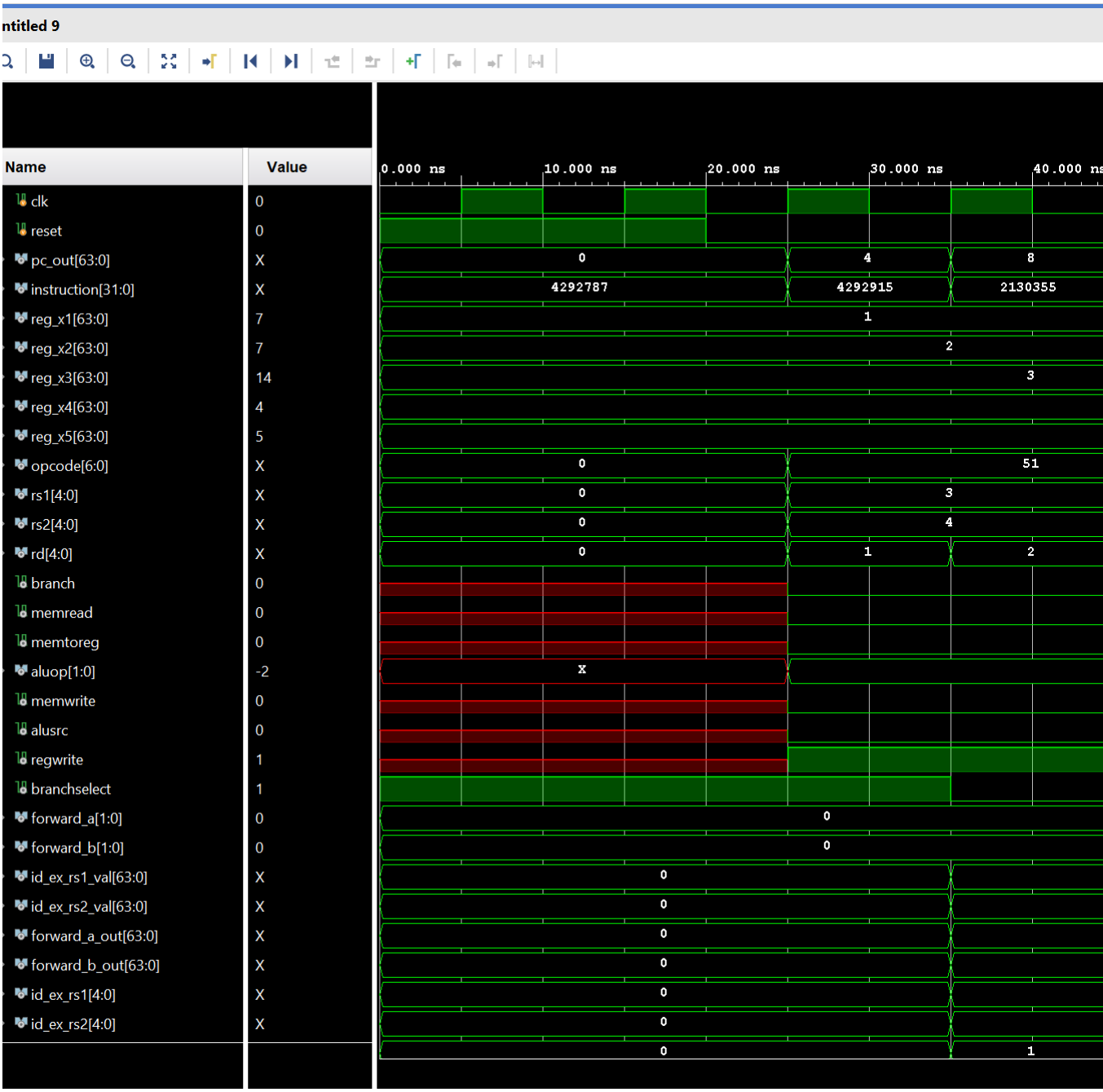


Figure 2.11: Test Case 5: Both

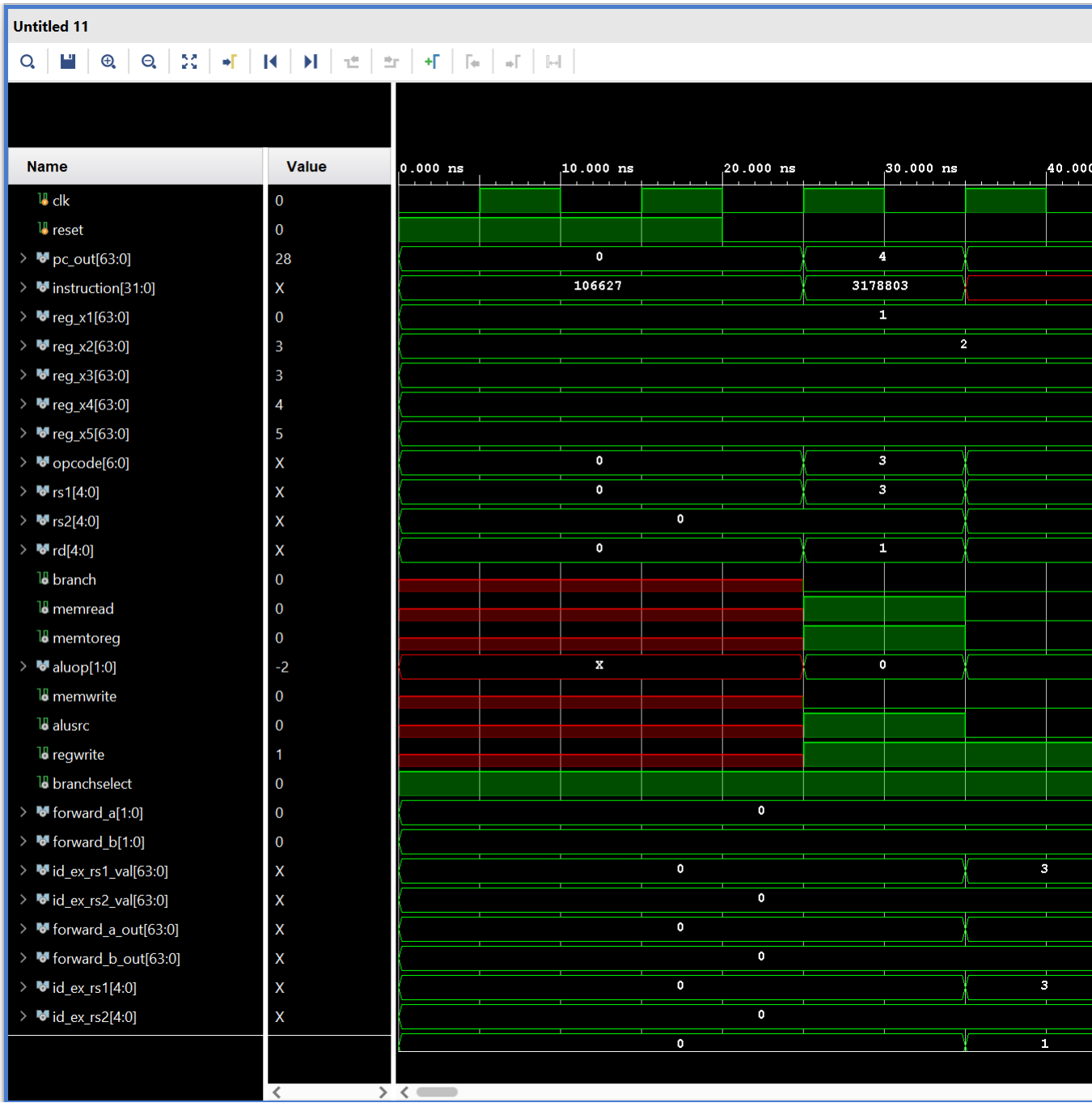


Figure 2.12: Testing using load type and r type instruction

# Conclusion and Challenges

Our first challenge was to alter the functionality so it could accomodate 4 byte data values instead of 8 byte how we had done during our labs.

In the second task we had some difficulty checking the workings of some instructions like branch, however, eventually we were able to successsfully check its functioning.

During task 3, we were unable to run the sorting algorithm successfully across our pipelined processor due to the fact that we were not able to add flushing functionality in time. We were however able to get rid of dependency issue caused between load and r type instructions using our hazard detection. Moreover, we were able to solve all forwarding cases using our forwarding unit.

This project really helped us gain a deeper knowledge and understanding of the inner workings of a processor, and the value of a pipelined processor

# Comparison

Pipelined processors are usually designed to be significantly faster than non-pipelined processors, with a potential speedup of up to four times. However, in our case, the pipelined processor is not functioning optimally. This can be due to a lack of flushing logic implementation or other issues with pipeline hazards, control that are affecting the efficiency of our processor. Hence, there is room for improvement in this design to make a much more efficient design.

# Task Division

- Haris Hussain Khan: Bubble Sort Code and Single Cycle Processor
- Muhammad Affan: Pipelined Processor and Debugging
- Maha Ozair: Pipelined Processor and Debugging
- All Three: Debugging and Integration

# References

[1] Book. *Course Book*. Computer Organization and Design: The Hardware/Software Interface RISC-V Edition by David A. Patterson, John L. Hennessy

# Appendix

GitHub links for codes of different processors:

- Single Cycle Processor (Task 1): https://github.com/affan002/RISC-V-pipelined-processor/tree/main/task_1
- Pipelined Processor (Task 2): https://github.com/affan002/RISC-V-pipelined-processor/tree/main/task_2_pipelined_with_hazards
- Pipelined Processor with Hazard Detection (Task 3): https://github.com/affan002/RISC-V-pipelined-processor/tree/main/CA%20Lab