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

Final Project Report RISC-V Processor

Submitted By:

Hussain Mustansir,
Muhammad Anas,
Muhammad Wajeeh

Research Assistant:

Ms. Zareen Tabassum

Course Instructor:

Farhan Khan

[25 APR 2024]

Contents

Introduction

To build a 5-stage pipelined processor capable of executing any one array sorting algorithm. We chose bubble sort algorithm, and carried the required steps to achieve the following objectives:

1. Converting bubble sort pseudocode (from lab 05) to RISC V assembly code and check its working on Venus. Modifying the lab 11 single-cycle processor to run and support the bubble sort code.

2. Pipelining the processor and then performing some test instructions separately to ensure the pipelined version is working.

3. Introducing hazard detection circuitry to detect hazards (data, control, and structural) and trying to handle them in hardware by using methods i.e.by forwarding, stalling, and flushing the pipeline.

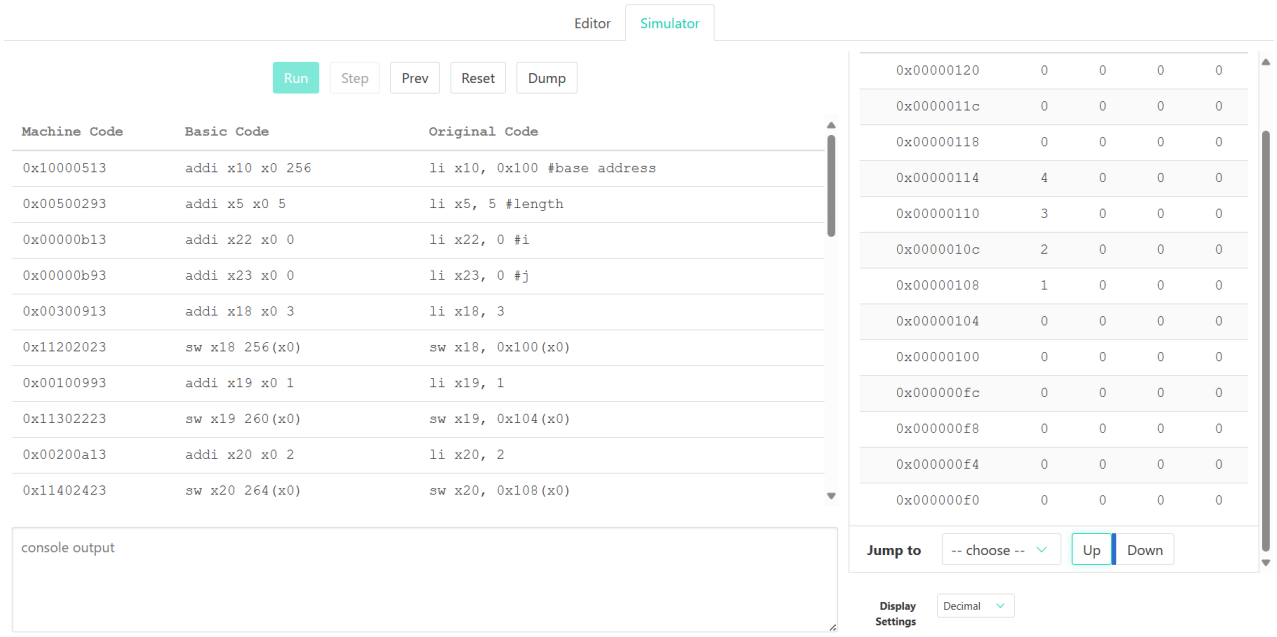
4. Comparing the performance of running the array sorting program on Single Cycle Processor with a pipelined RISC-V Processor in terms of execution time.

Methodology:

Task 1

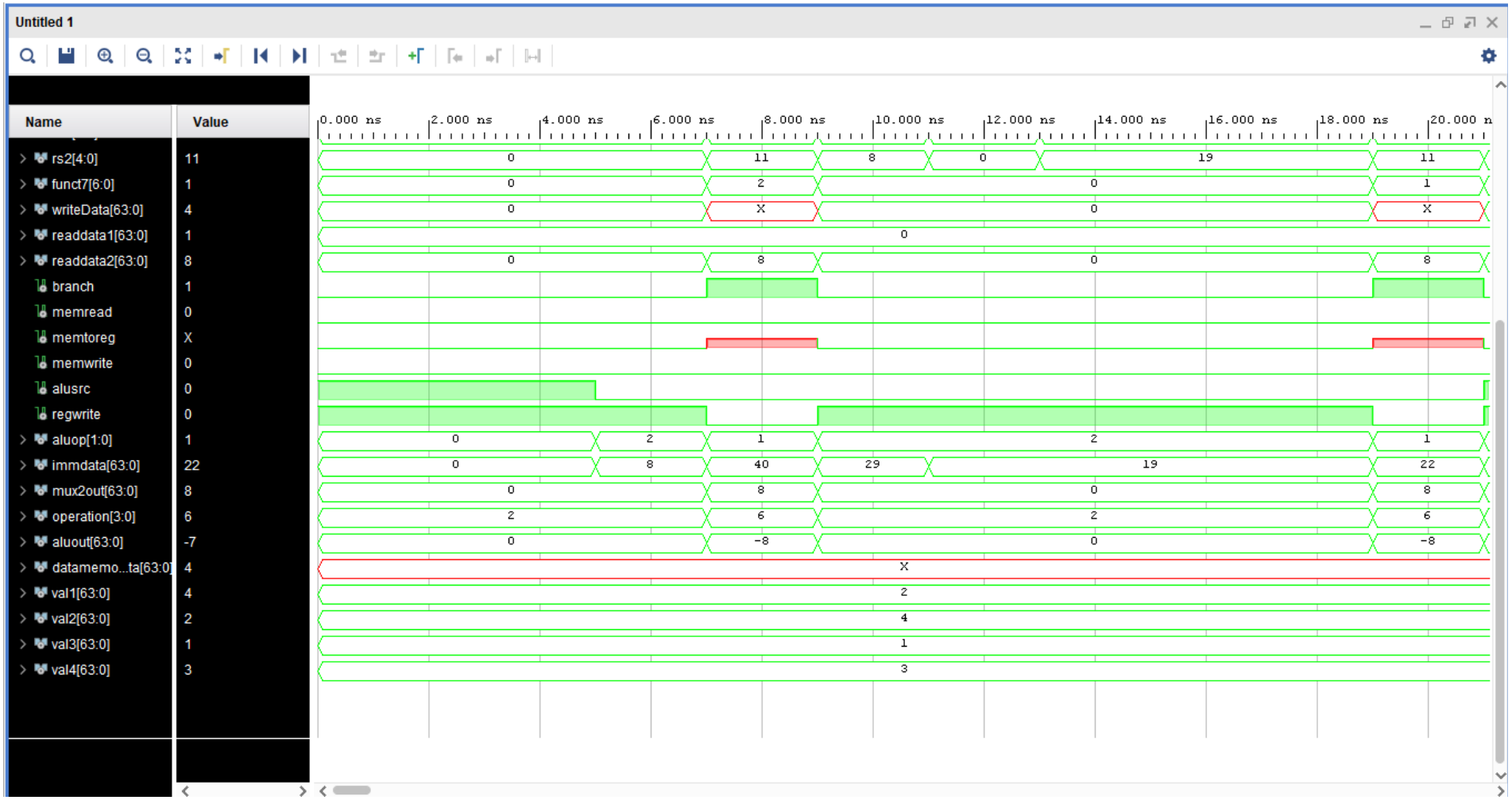
Bubble Sort Pseudocode to Machine Code

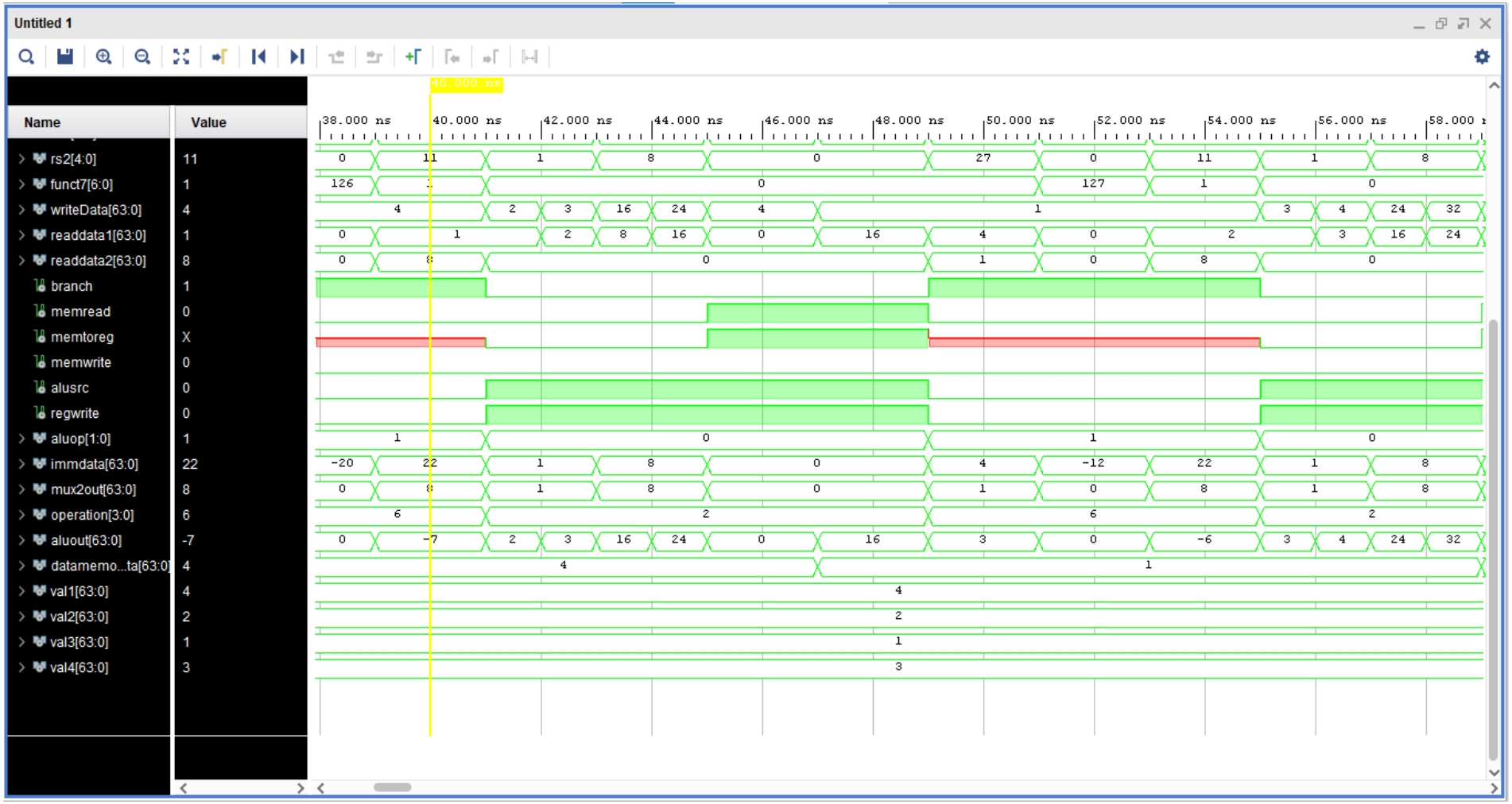
We first implemented the sorting algorithm in RISC V assembly on the Venus simulator.

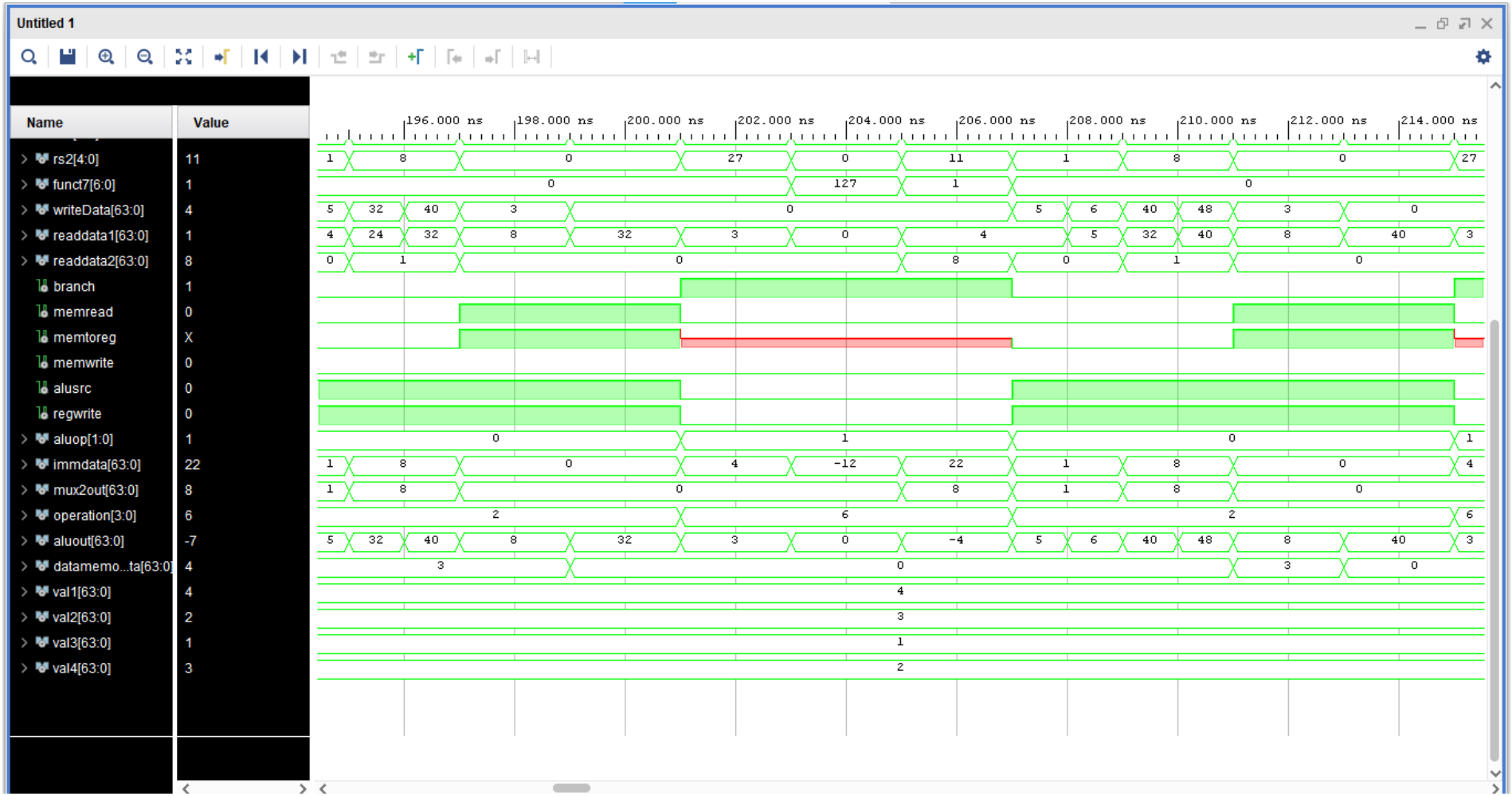


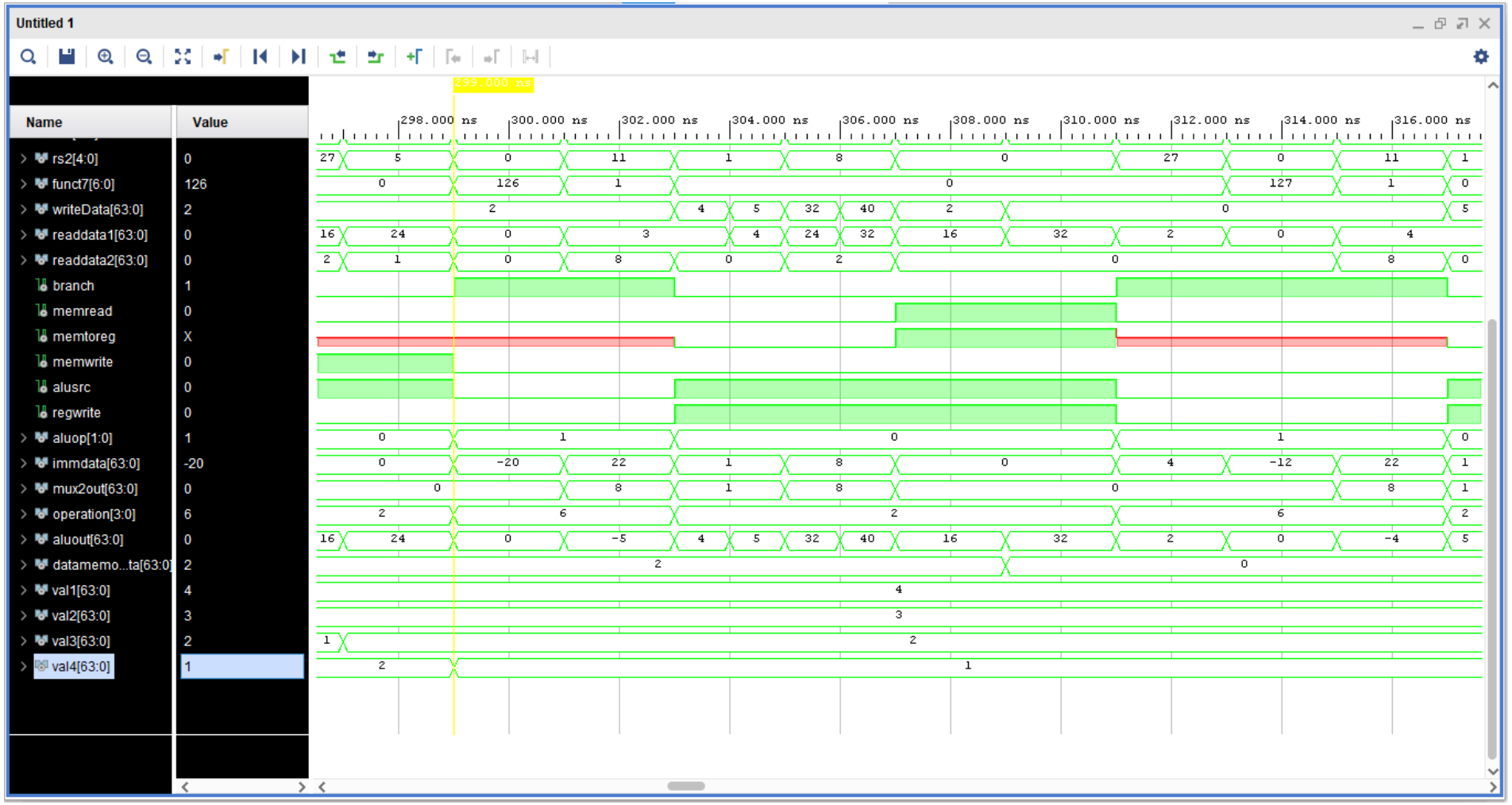
Implementation of Bubble Sort Using RISC V Single cycled processor (without any hazard detection)

The lab 11 module, in which we assembled all the modules to form the processor, has several modifications done to it. We also added a branching module to handle branch operations and changed the code for the ALU, data memory, and instruction memory. As we're using a 64-bit processor, we changed the offset from 4 bytes to 8 bytes in the ALU code and added support for the funct3 bit of the bgt and blt instructions. We utilized the instruction memory of the single-cycle CPU to sort the list after initialising it.


Untitled 1
Name
Value
0.000 ns
2.000 ns
4.000 ns
6.000 ns
8.000 ns
10.000 ns
12.000 ns
14.000 ns
16.000 ns
18.000 ns
20.000 n
rs2[4:0] 11
funct7[6:0] 1
writeData[63:0] 4
readdata1[63:0] 1
readdata2[63:0] 8
branch 1
memread 0
memtoreg X
memwrite 0
alusrc 0
regwrite 0
aluop[1:0] 1
immdata[63:0] 22
mux2out[63:0] 8
operation[3:0] 6
aluout[63:0] -7
datamemo...ta[63:0] 4
val1[63:0] 4
val2[63:0] 2
val3[63:0] 1
val4[63:0] 3


Untitled 1
40.000 ns
Name
Value
38.000 ns
40.000 ns
42.000 ns
44.000 ns
46.000 ns
48.000 ns
50.000 ns
52.000 ns
54.000 ns
56.000 ns
rs2[4:0] 11
funct7[6:0] 1
writeData[63:0] 4
readdata1[63:0] 1
readdata2[63:0] 8
branch 1
memread 0
memtoreg X
memwrite 0
alusrc 0
regwrite 0
aluop[1:0] 1
immdata[63:0] 22
mux2out[63:0] 8
operation[3:0] 6
aluout[63:0] -7
datamemo...ta[63:0] 4
val1[63:0] 4
val2[63:0] 2
val3[63:0] 1
val4[63:0] 3


Untitled 1
Name
Value
196.000 ns
198.000 ns
200.000 ns
202.000 ns
204.000 ns
206.000 ns
208.000 ns
210.000 ns
212.000 ns
214.000 ns
rs2[4:0]
11
funct7[6:0]
1
writeData[63:0]
4
readdata1[63:0]
1
readdata2[63:0]
8
branch
1
memread
0
memtoreg
X
memwrite
0
alusrc
0
regwrite
0
aluop[1:0]
1
immdata[63:0]
22
mux2out[63:0]
8
operation[3:0]
6
aluout[63:0]
-7
datamemo...ta[63:0]
4
val1[63:0]
4
val2[63:0]
2
val3[63:0]
1
val4[63:0]
3

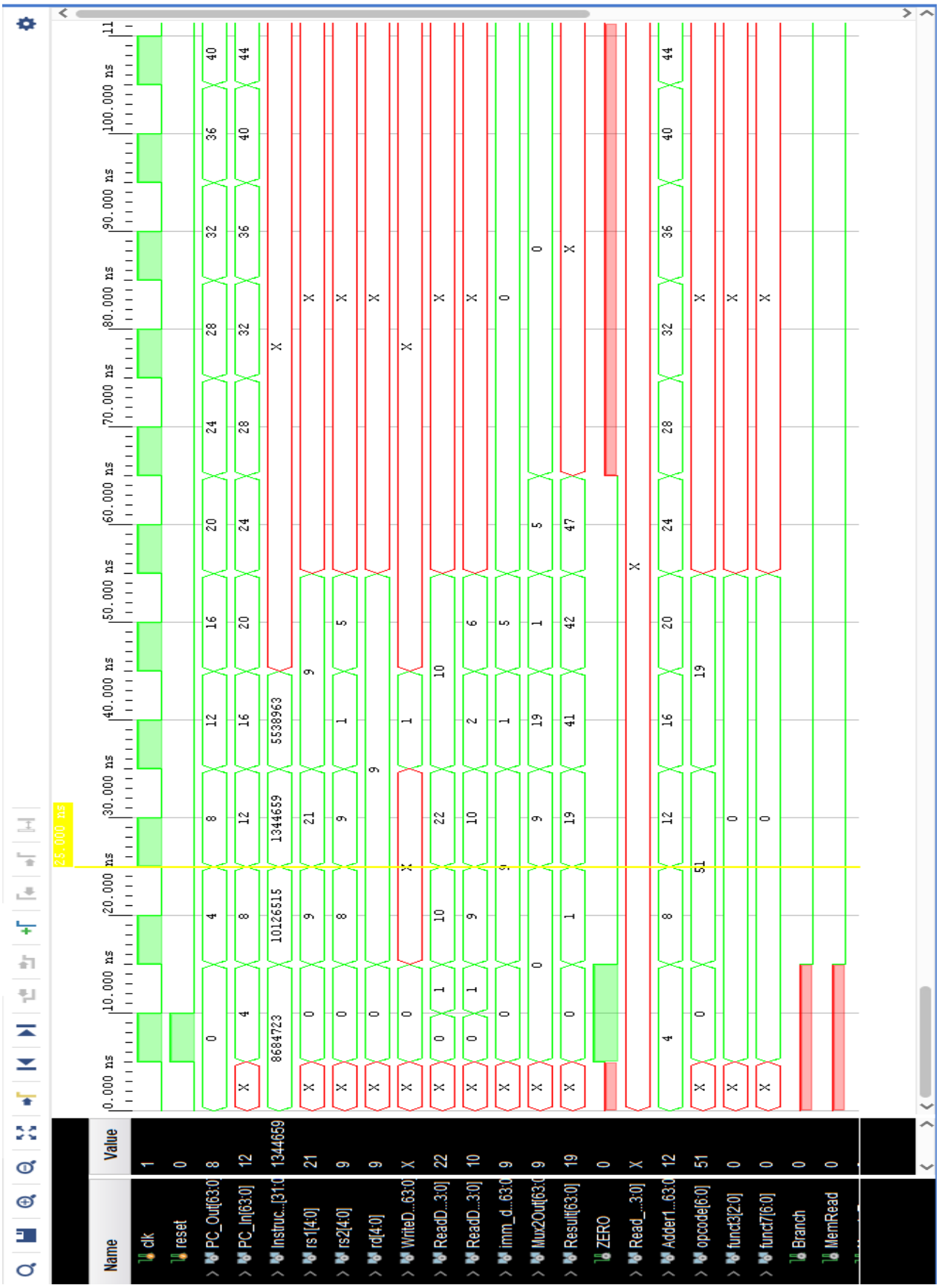

Untitled 1
299.000 ns
298.000 ns
300.000 ns
302.000 ns
304.000 ns
306.000 ns
308.000 ns
310.000 ns
312.000 ns
314.000 ns
316.000 ns
Name
Value
rs2[4:0]
0
funct7[6:0]
126
writeData[63:0]
2
readdata1[63:0]
0
readdata2[63:0]
0
branch
1
memread
0
memtoreg
X
memwrite
0
alusrc
0
regwrite
0
aluop[1:0]
1
immdata[63:0]
-20
mux2out[63:0]
0
operation[3:0]
6
aluout[63:0]
0
datamemo...ta[63:0]
2
val1[63:0]
4
val2[63:0]
3
val3[63:0]
2
val4[63:0]
1

Task 2

Pipelined RISC V Processor:

We then moved on to pipelining the modules after the algorithm was successfully implemented on the single-cycle processor. We added the IF/ID, ID/EX, EX/MEM, and MEM/WB modules to act as intermediary registers for the pipeline. The data from earlier instructions was stored in these registers and sent through the pipeline with their assistance. In order to make sure the pipeline was operating correctly, we tested each instruction independently.

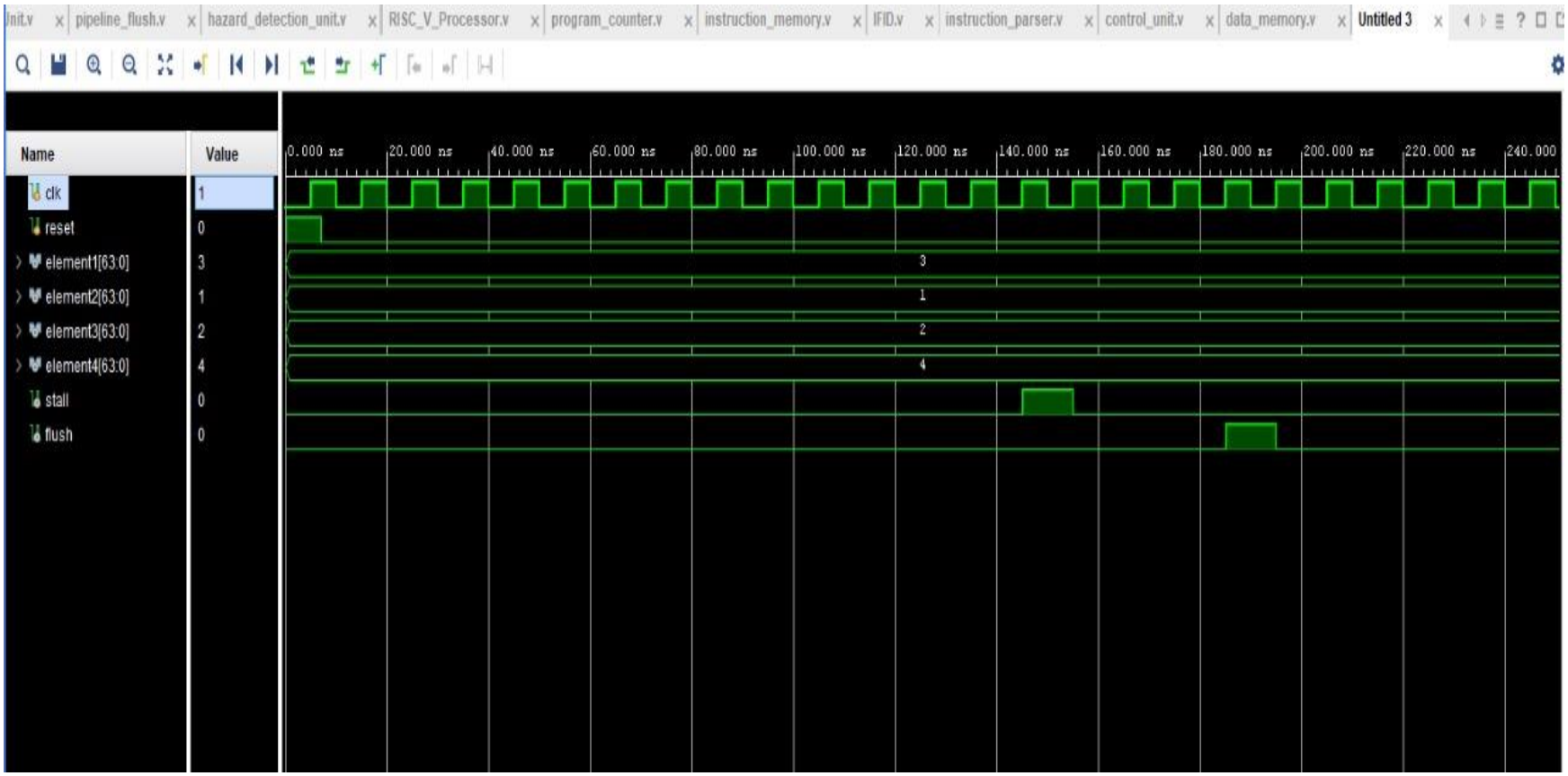
In addition, we improved the code to include a 3 to 1 Mux and a passing Unit, enabling data passing inside the pipeline. In order to minimize stalls and increase pipeline performance, this forwarding unit was connected with the hazard detecting unit to determine the best way for sending data.

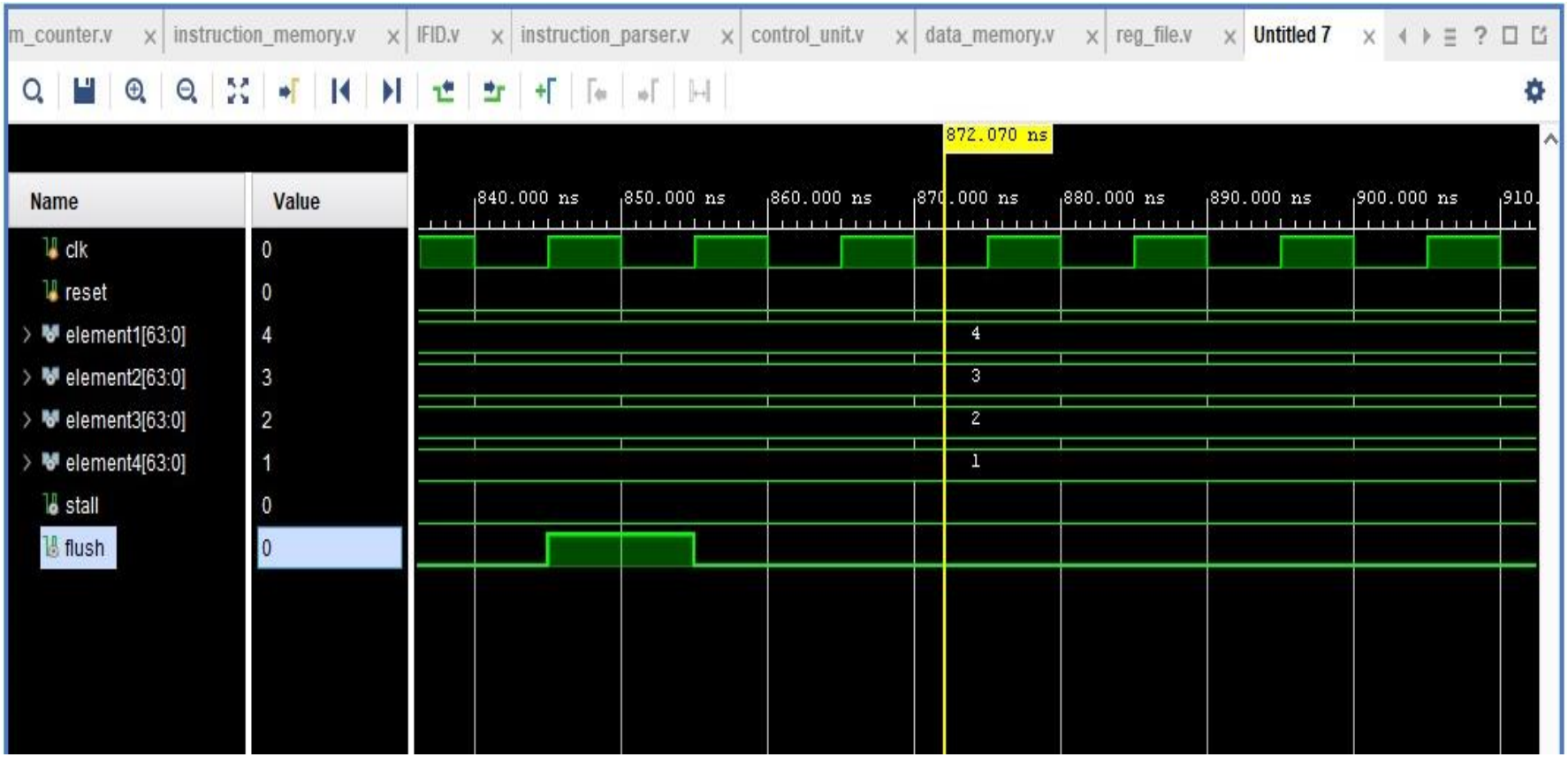

Name
Value
clk
reset
PC_Out[63:0]
PC_In[63:0]
Instruc...[31:0]
rs1[4:0]
rs2[4:0]
rd[4:0]
WriteD...63:0]
ReadD...3:0]
ReadD...3:0]
imm_d...63:0]
Mux2Out[63:0]
Result[63:0]
ZERO
Read_...3:0]
Adder1...63:0]
opcode[6:0]
funct3[2:0]
funct7[6:0]
Branch
MemRead
25.000 ns

Task 3

Implementing Hazard Detection

In order to address data, structural, and control risks, we incorporated methods to pause the pipeline when necessary and integrated hazard detection circuits into the code. Code dependencies or the requirement to transmit data later cause these dangers. To lessen these dangers, a hazard detection unit is used. This unit uses signals to communicate with the forwarding unit to determine whether to flush the pipeline or stop it in order to transfer data.


Unit.v
pipeline_flush.v
hazard_detection_unit.v
RISC_V_Processor.v
program_counter.v
instruction_memory.v
IFID.v
instruction_parser.v
control_unit.v
data_memory.v
Untitled 3
Name
Value
clk
1
reset
0
element1[63:0]
3
element2[63:0]
1
element3[63:0]
2
element4[63:0]
4
stall
0
flush
0
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
160.000 ns
180.000 ns
200.000 ns
220.000 ns
240.000
3
1
2
4


m_counter.v
instruction_memory.v
IFID.v
instruction_parser.v
control_unit.v
data_memory.v
reg_file.v
Untitled 7
Name
Value
clk
0
reset
0
element1[63:0]
4
element2[63:0]
3
element3[63:0]
2
element4[63:0]
1
stall
0
flush
0
872.070 ns
840.000 ns
850.000 ns
860.000 ns
880.000 ns
890.000 ns
900.000 ns

Task 4

Performance Comparison

Pipelined processors are designed to be fast as such they can achieve speedups of up to 4 times. Our processor without pipelining took about 300 ns with each instruction being 1 ns. So total cycles are total time instructions/(2*clk time), hence 150. As seen in Task 3 our bubble sort is working with hazard detection and pipelining. It took about 800 ns with each instruction being 1 ns. So total cycles are total time instructions/(2*clk time), hence 400. Which means that our pipelined processor was not able to achieve lesser clock cycles.

Conclusion and Challenges

Building this processor came several challenges, primarily due to very little technical support to us, scanty time, and lack of in-depth understanding of the processor of RISC V. Other challenges was implementing the sorting algorithm, as such it required support of slli and blt, bgt and adding the support was itself arduous then the sorting algorithm was not working correctly which caused more issues in task 1. In task 2 we encountered issues in understanding and implementing the logic of pipelining.

We faced many challenges during the simulations, including troubleshooting errors and implementing stalls at crucial junctures. Completing the simulations successfully proved to be quite difficult. Furthermore, integrating branch conditions came with its own set of challenges.

We persisted and created a pipelined processor that can handle the majority of risks in spite of dependencies and complexity that caused stalls.

Task Distribution:

Task 1: Anas
Task 2: Hussain with some help of Wajeeh
Task 3: Wajeeh with some help of Hussain

References

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

Project link

The GitHub link for our project can be found here:
Repository link of our code