RISC V Pipelined Processor

Final Report

Submitted by:
Basil Khowaja
Hania Kashif
Saif Nazir

Research Assistant:
Haseeb Khan

Course Instructor:
Dr. Muhammad Farhan

Computer Engineering
Habib University
Fall Semester'23

Date: December 20, 2023

A report submitted in fulfillment
of the requirements for the lab project
of CE/CS - 321/330: Computer Architecture

Contents

1 Introduction

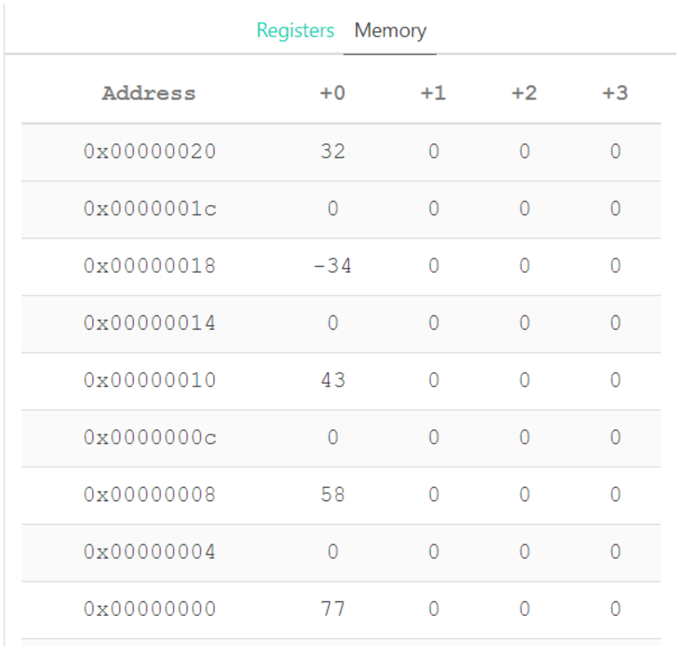
Our project was to build a 5-stage pipelined RISC V processor capable of executing a bubble sorting algorithm. The tasks we performed were:

1. Converting bubble sort pseudocode to RISC V assembly code and verifying on Venus. Modifying the lab 11 single-cycle processor to run the bubble sort code.
2. Pipelining the processor and then performing some test instructions separately to ensure the pipelined version worked correctly.
3. Introducing hazard detection circuitry to detect hazards (data, control, and structural) and trying to handle them in hardware 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.

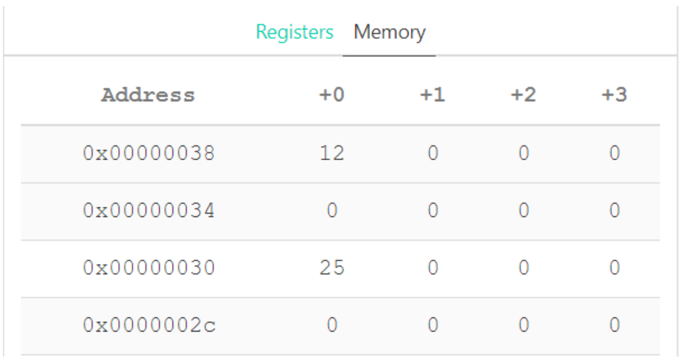
2 Task 1

2.1 Bubble Sort Pseudocode to Machine Code

We first implemented the sorting algorithm in RISC V assembly on the Venus simulator. We had also previously tried implementing sorting in one of the labs so we modified that lab code from the pseudocode.



Address	+0	+1	+2	+3
0x00000020	32	0	0	0
0x0000001c	0	0	0	0
0x00000018	-34	0	0	0
0x00000014	0	0	0	0
0x00000010	43	0	0	0
0x0000000c	0	0	0	0
0x00000008	58	0	0	0
0x00000004	0	0	0	0
0x00000000	77	0	0	0



Registers
Memory

Address	+0	+1	+2	+3
0x00000038	12	0	0	0
0x00000034	0	0	0	0
0x00000030	25	0	0	0
0x0000002c	0	0	0	0

Figure 1: initializing memory with values

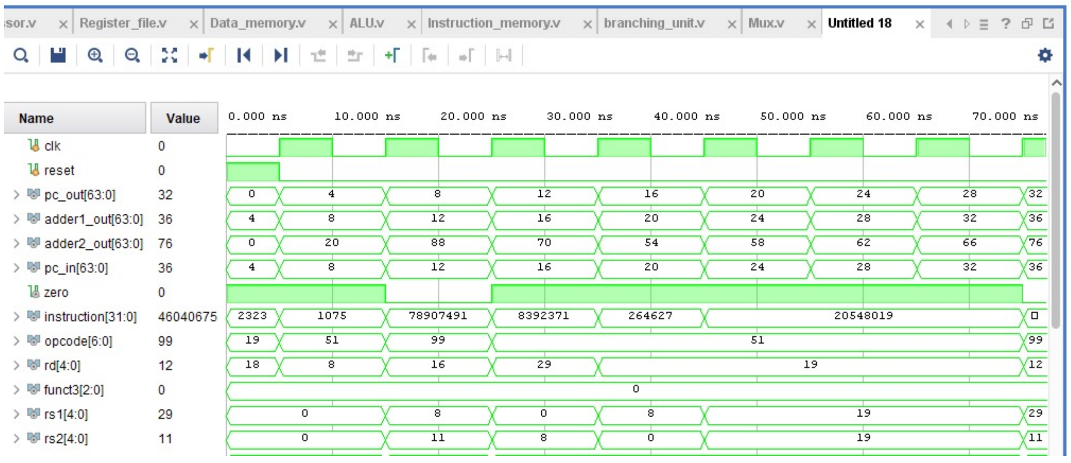
2.2 Bubble Sort Implementation Single Cycle

We made some minor modifications to the lab 11 module where we instantiated all modules together to make the processor. We also modified ALU and instruction memory code and added a branch unit code to support branch operations. In ALU code, we added functionality for funct3 bit of bgt and blt, and also changed the 4-byte offset to an 8-byte offset as we have a 64-bit processor. We initialized a list and then sorted that on the single-cycle processor using the instruction memory.

Listing 1: python script to initialize instruction memory

```
lst = [// list of machine codes as strings ]
    for i in range(len(lst)):
        print(f"inst_mem[{4*i}]=8'h{lst[i][6:8]};")
        print(f"inst_mem[{4*i+1}]=8'h{lst[i][4:6]};")
        print(f"inst_mem[{4*i+2}]=8'h{lst[i][2:4]};")
        print(f"inst_mem[{4*i+3}]=8'h{lst[i][:2]};")
```

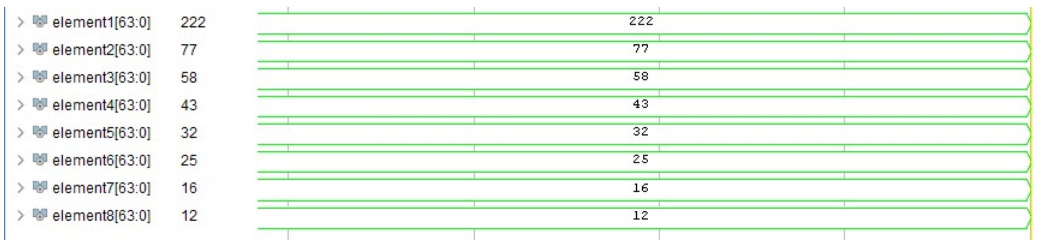
2.3 Simulation Output



Name	Value
clk	0
reset	0
pc_out[63:0]	32
adder1_out[63:0]	36
adder2_out[63:0]	76
pc_in[63:0]	36
zero	0
instruction[31:0]	46040675
opcode[6:0]	99
rd[4:0]	12
funct3[2:0]	0
rs1[4:0]	29
rs2[4:0]	11

```
pc_out[63:0]       0  4  8  12  16  20  24  28  32
adder1_out[63:0]   4  8  12  16  20  24  28  32  36
adder2_out[63:0]   0  20  88  70  54  58  62  66  76
pc_in[63:0]        4  8  12  16  20  24  28  32  36
instruction[31:0]  2323  1075  78907491  8392371  264627  20548019
opcode[6:0]        19  51  99  51  99
rd[4:0]            18  8  16  29  19  12
funct3[2:0]        0
rs1[4:0]           0  8  0  8  19  29
rs2[4:0]           0  11  8  0  19  11
```

Figure 2: Snippet of simulation output



```
element1[63:0]  222
element2[63:0]  77
element3[63:0]  58
element4[63:0]  43
element5[63:0]  32
element6[63:0]  25
element7[63:0]  16
element8[63:0]  12
```

Figure 3: Sorted array

3 Task 2

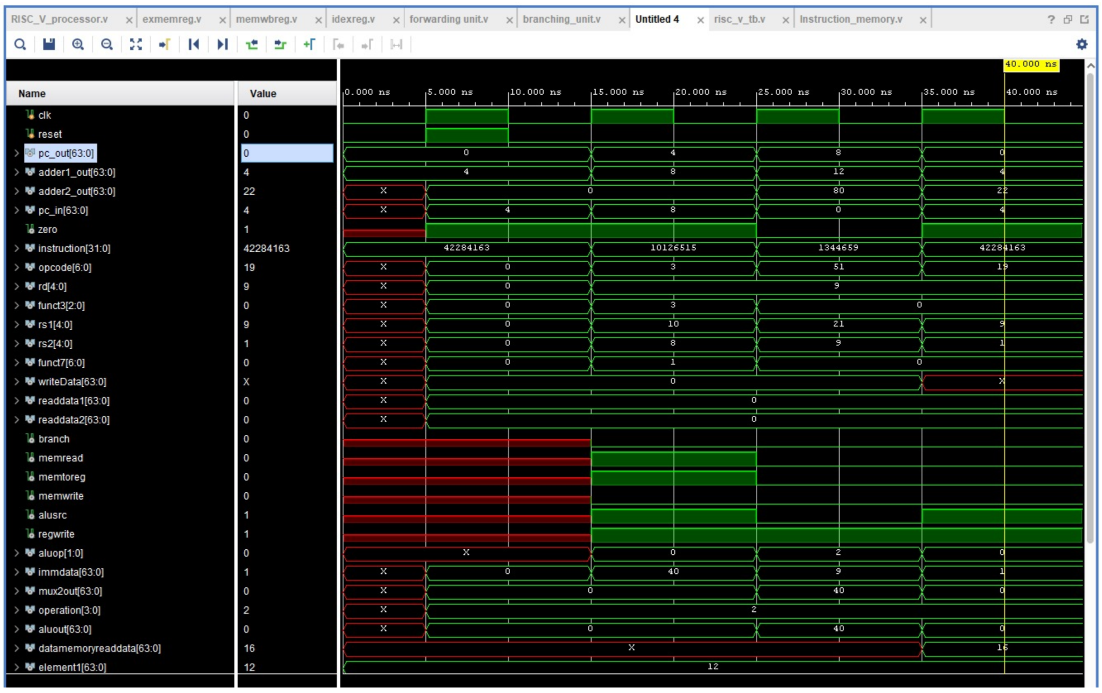
3.1 Pipelined RISC V Processor

After implementing the algorithm on the single-cycle processor we next modified the code to pipeline the modules. For that, we referred to our course book and then added the following modules to act as intermediate registers for the pipeline:

- IF/ID
- ID/EX
- EX/MEM
- MEM/WB

These registers help store the value of previous instruction's data and pass it along the pipeline. We tested each instruction separately to make sure that the pipeline was working correctly. We also implemented the forwarding unit by modifying the previous code so that we can forward data in the pipeline. The forwarding unit will be integrated with the hazard detection unit which determines where to forward data.

3.2 Simulation Output



Name	Value
clk	0
reset	0
pc_out[63:0]	0
adder1_out[63:0]	4
adder2_out[63:0]	22
pc_in[63:0]	4
zero	1
instruction[31:0]	42284163
opcode[6:0]	19
rd[4:0]	9
funct3[2:0]	0
rs1[4:0]	9
rs2[4:0]	1
funct7[6:0]	0
writeData[63:0]	X
readdata1[63:0]	0
readdata2[63:0]	0
branch	0
memread	0
memtoreg	0
memwrite	0
alusrc	1
regwrite	1
aluop[1:0]	0
immdata[63:0]	1
mux2out[63:0]	0
operation[3:0]	2
aluout[63:0]	0
datamemoryreaddata[63:0]	16
element1[63:0]	12

Figure 4: Snippet of simulation output

4 Task 3

4.1 Implementing Hazard Detection Circuitry

Hazards such as data, structural, and control are dealt with within the code by implementing hazard detection circuitry and stalling the pipeline. These hazards mostly arise from dependencies in the code or if the data needs to be forwarded further at some point. For this, we tried to implement the hazard detection unit that controls when to stall the pipeline or forward the data by signaling the forwarding unit to stall or flush the pipeline.

4.2 Simulation Output

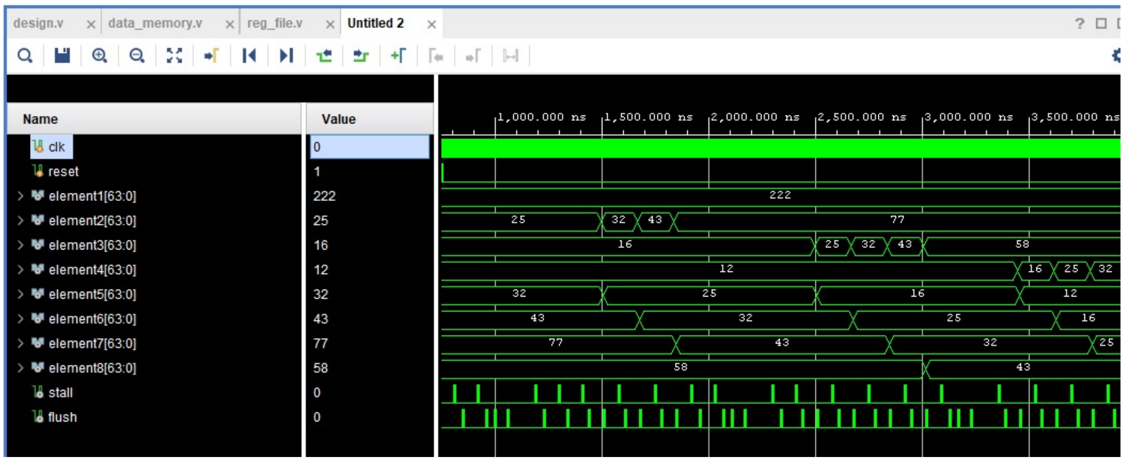


Figure 5: Snippet of simulation output

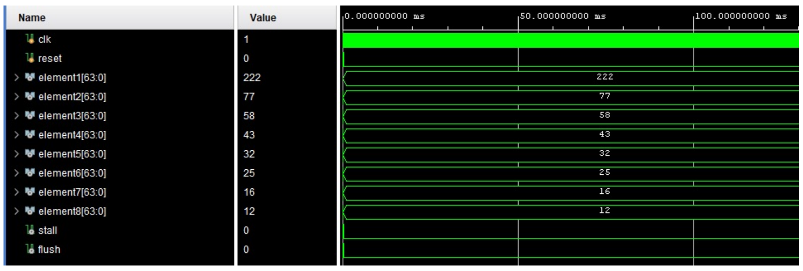


Figure 6: sorted array

5 Performance Comparison

Pipelined processors are usually faster than non-pipelined processors, but in our case, the pipelined processor is not functioning optimally, with a latency of about 6000ns, which is longer than the 3000ns latency of the non-pipelined processor. Despite the fact that we were able to implement the pipelining of the processor, it came with many issues in efficiency. To improve the performance of our pipelined processor, we need to analyze and optimize its design.

6 Conclusion and Challenges

Building the processor was a challenge and we had lots of issues because Vivado kept crashing and many of our zipped files didn't open on another PC, or our laptops. The main aim of pipelining was to increase the efficiency of the processor which allows us to perform multiple tasks with ease. We faced a lot of issues with simulations and incorporating stalls at critical points but it was very hard to make the simulation work until the end. The branch conditions were also difficult to incorporate. We faced stall issues due to complexity and dependency issues, but we were still able to make a pipelined processor that dealt with most hazards.

7 References

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

8 Appendix

The GitHub link for our project can be found here.