# RISC V Pipelined Processor

## Final Report

**Submitted by:**
Raahim Hashmi
Meesum Abbas
Laiba

**Research Assistant:**
Maham Tabassum, Ahmed Bilal

**Course Instructor:**
Dr. Farhan Khan

Computer Science, Computer Engineering
Habib University
Spring '24

Date: April 27, 2024

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

# Contents

## 1 Introduction

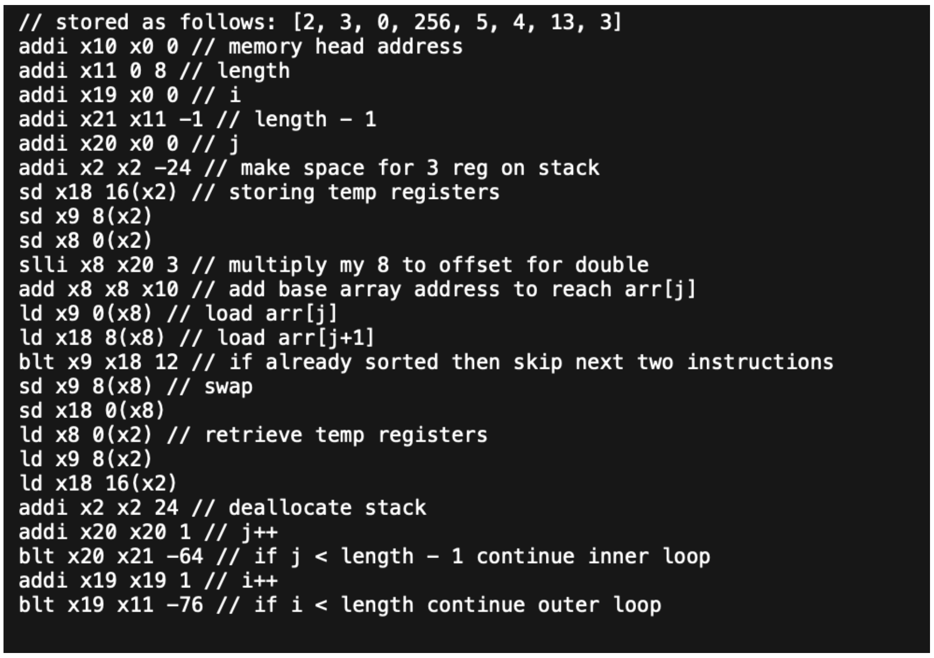
Our project involved building a 5-stage pipelined RISC V processor capable of executing a bubble sorting algorithm. We completed the following tasks:

1. Converted the bubble sort code in python to RISC V assembly code and verified it on the Venus simulator. (Then converted the 32 lw and sw to ld and sd and other changes that were required to shift from 4-bytes to 8-bytes). Modified the lab 11 single-cycle processor to run the bubble sort code.
2. Pipelined the processor and tested individual instructions to ensure the pipelined version worked correctly.
3. Implemented hazard detection circuitry to identify and handle hazards (data, control, and structural) by using techniques such as forwarding, stalling, and flushing the pipeline.
4. Compared the performance of running the array sorting program on a 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, which we had done in a previous Lab. Then we modified it to support double word storing, loading from the data memory. We then converted the assembly code to machine code and initialized the instruction memory with the machine code. And finally, then ran the code on the single-cycle processor to sort an array of numbers. The following is the assembly code for the bubble sort algorithm:



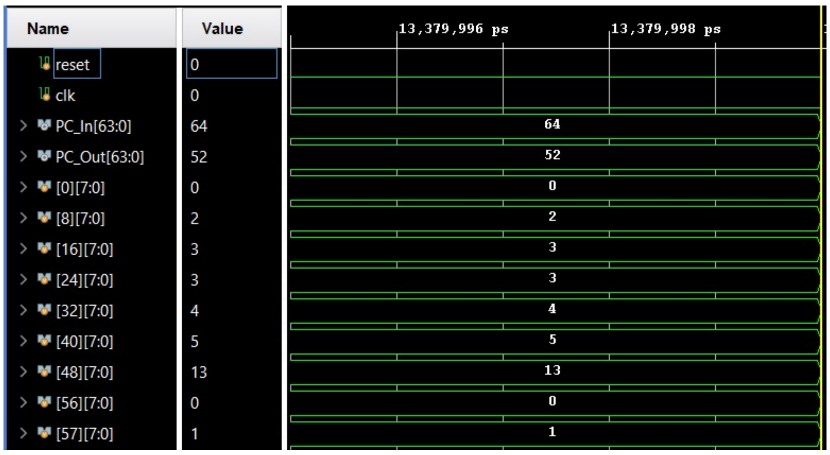
```
// stored as follows: [2, 3, 0, 256, 5, 4, 13, 3]
addi x10 x0 0 // memory head address
addi x11 0 8 // length
addi x19 x0 0 // i
addi x21 x11 -1 // length - 1
addi x20 x0 0 // j
addi x2 x2 -24 // make space for 3 reg on stack
sd x18 16(x2) // storing temp registers
sd x9 8(x2)
sd x8 0(x2)
slli x8 x20 3 // multiply my 8 to offset for double
add x8 x8 x10 // add base array address to reach arr[j]
ld x9 0(x8) // load arr[j]
ld x18 8(x8) // load arr[j+1]
blt x9 x18 12 // if already sorted then skip next two instructions
sd x9 8(x8) // swap
sd x18 0(x8)
ld x8 0(x2) // retrieve temp registers
ld x9 8(x2)
ld x18 16(x2)
addi x2 x2 24 // deallocate stack
addi x20 x20 1 // j++
blt x20 x21 -64 // if j < length - 1 continue inner loop
addi x19 x19 1 // i++
blt x19 x11 -76 // if i < length continue outer loop
```

Figure 1: Assembly code for bubble sort

## 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 blt, and also changed the 4-byte offset to an 8-byte offset as we have a 64-bit processor. We also added functionality for slli. We then initialized the instruction memory with the machine code and ran the code on the single-cycle processor to sort an array of numbers. The following is the manually written machine code in verilog:

## 2.3 Simulation Output



| Name | Value |
|---|---|
| reset | 0 |
| clk | 0 |
| PC_In[63:0] | 64 |
| PC_Out[63:0] | 52 |
| [0][7:0] | 0 |
| [8][7:0] | 2 |
| [16][7:0] | 3 |
| [24][7:0] | 3 |
| [32][7:0] | 4 |
| [40][7:0] | 5 |
| [48][7:0] | 13 |
| [56][7:0] | 0 |
| [57][7:0] | 1 |

Figure 2: Snippet of simulation output/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 Dr. Farhan's lecture slides 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

Figure 3: Snippet of simulation output

# 4 Task 3

## 4.1 Detecting Hazards and Resolving them

Data and Control hazards are dealt with within the code by implementing hazard detection circuitry and stalling/flushing the pipeline when needed. 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 4: Snippet of simulation output

Figure 5: 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

It was challenge for us to coordinate with each other and specify a time to work on together, so we had to divide the work. That too, however, came with challenges cause one of us (Meesum) works on a mac which does not support vivado. He had to use VS Code, which was still challenging as there was no way of getting the schema of the code. However, we used github and worked as a team to overcome the issues, while dividing the tasks as well. We were able to implement the pipelined processor and the hazard detection unit, and it is working perfectly as 5-cycle pipelined processor. The performance ...

## 7 References

[1] Book. *Course Book.* Computer Organization and Design: The Hardware/Software Interface RISC-V Edition by David A. Patterson, John L. Hennessy [2] Lecture Notes. *Dr. Farhan Khan.* Computer Architecture, Habib University

## 8 Appendix

The GitHub link for our project can be found here.