
# RISC V Pipelined Processor

## Final Report

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

**Research Assistant:**
Maham Tabassum

**Course Instructor:**
Ahmed Ali Mustansir

Computer Science
Habib University
Spring Semester'24

Date: June 17, 2024

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

# Contents

# 1 Introduction

For our final project, we were tasked with creating a 5-stage pipelined RISC-V processor capable of properly executing a bubble sort algorithm. Following are the main tasks associated with achieving this goal:

1. Enhancing capability of our lab 11 processor to accommodate instructions needed for bubble sort.
2. Encoding the bubble sort assembly code into instruction memory and testing it on the processor.
3. Pipelining the processor by introducing pipeline stage registers, forwarding unit, and hazard detection to produce stall and flush
4. Comparing the performance of the bubble sort algorithm on the pipelined processor versus the single cycle processor

# 2 Task 1

## 2.1 Modifying the existing processor

We modified our lab 11 processor to perform logical shift left immediate (*slli*) and branch less than (*blt*) operations to accommodate the bubble sort. *slli* was implemented by producing the required *Operation* signal from *ALU_Control* and performing the shift left operation in *ALU_64_bit*. *blt* was implemented by creating a *branchOp* signal which is low when the operation is *beq* and high when it is *blt*. This, along with *branch*, *Zero*, and a new signal, *Less*, from *ALU_64_bit* was used in a simple equation (stored in *var3*) to determine when to branch.

```
// 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: Bubble Sort Assembly Code

## 2.2 Bubble Sort Encoding and Testing

We modified our bubble sort algorithm1 to work with a double array and encoded it into the instruction memory. We used the stack in our algorithm and initialized the stack pointer with 92 in *RegisterFile*. Data memory was initialized with the following values: [2,3,0,5,256,4,13,3]. Figures 2 and 3 show the unsorted array and sorted array outputs. Note that each element of the array occupies eight memory locations i.e. 64 bits with the highest index of the highest memory location indicating MSB.

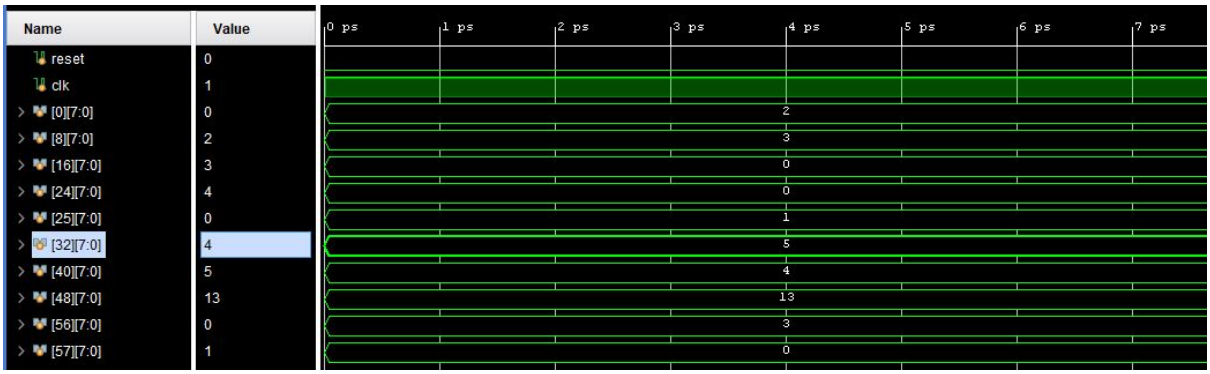


Figure 2: Unsorted Array

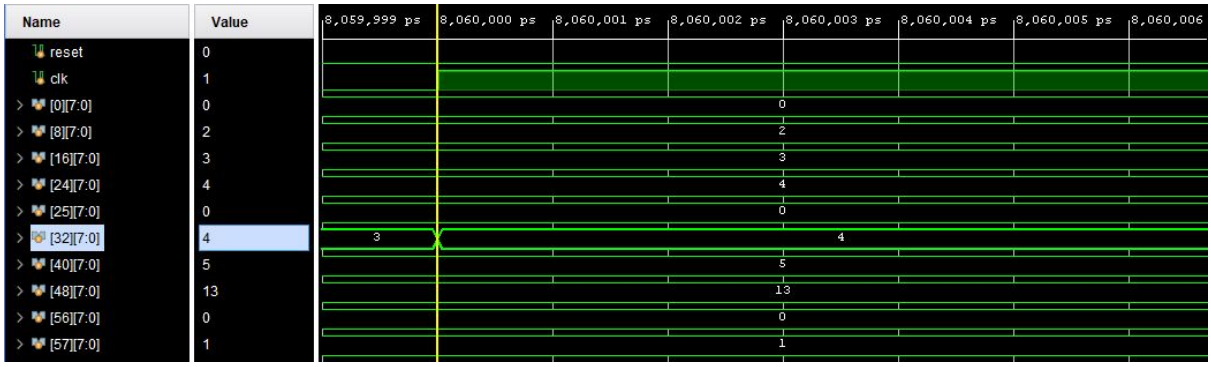


Figure 3: Sorted Array

# 3 Task 2

## 3.1 Pipelining the processor

Once the single cycle processor was functioning properly, we set out to pipeline the processor. For this task we started off by first creating the four pipeline stage registers i.e. *ID_ID_Reg, ID_EXE_Reg, EXE_MEM_Reg, MEM_WB_Reg.* We carefully made the proper connections by following the course textbook. We then ran some individual instructions on the processor to ensure it was functioning properly.

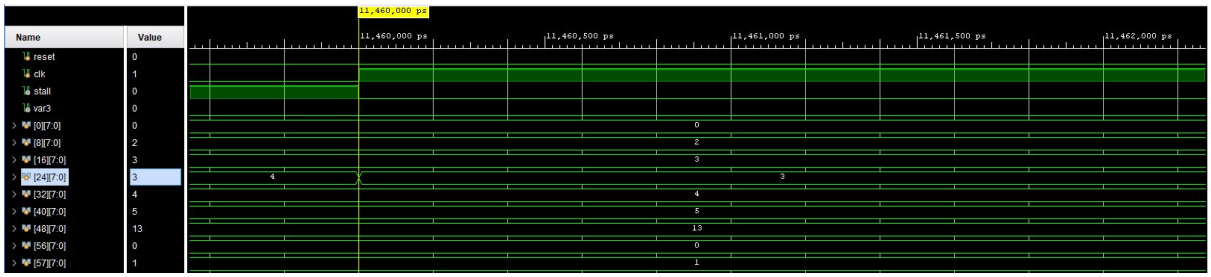


Figure 4: Task 3 Bubble Sort Result

### 3.2 Forwarding Unit

For implementing the forwarding unit, we used the conditions from the course textbook and made the required connections. Once it was implemented, we ran a couple of dependent instructions and observed they were functioning properly.

## 4 Task 3

### 4.1 Data Hazard Detection

In order to deal with the data hazards, we created a hazard detection unit and followed the conditions stated in the course textbook and made the required connections. Once a data hazard was detected, a stall was sent and the outputs for *Program_Counter* and *IF_ID_Reg* would retain their value and all output control signals from *Control_Unit* would become low, essentially acting as a NOP instruction.

### 4.2 Control Hazard Detection

For the control hazards, we used *var3* to act as a signal to flush the *IF_ID_Reg, ID_EX_REG,* and *EX_MEM_Reg* when a branch was detected.

### 4.3 Bubble Sort on Pipelined Processor

We then ran our bubble sort instructions on the pipelined processor and figure 4 shows the results. As can be seen, the bubble sort is functioning properly (the unsorted array is the same as figure 1).

# 5 Performance Comparison

The pipelined processor that we made is slower than the non-pipelined equivalent. Normally, pipelined processors do take more clock cycles than regular processors; however, their clock is much shorter which makes them faster. In our case, we have kept the clock the same as the regular processor and not according to the duration of the longest pipeline stage as we should have, which makes the non-pipelined version faster simply because of less clock cycles.

# 6 Challenges

We faced a number of challenges. Initially during forwarding, we had some issues where the data was being forwarded a clock cycle too late which were resolved by removing the clock from the *RegisterFile* as we already had a clock-activated *MEM_WB_Reg*. In hazard detection, we had some issues with the stall being activated too late which were resolved by some trial and error on what data fields should be held and what should be zero.

# 7 Task Division

- Laiba: Task 3 (Hazard Detection)
- Meesum: Task 2 (Pipelining and Forwarding Unit)
- Raahim: Task 1 (Non-pipelined), debbugging and testing tasks 2 and 3

# 8 Conclusion

Building the processor and overcoming the challenges that came with it was an overall satisfying and fulfilling experience. This project furthered our understanding of Verilog and enabled us to apply our theoretical knowledge of computer architecture in a worthwhile project.

# 9 References

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

# 10 Appendix

The GitHub link for our project can be found here.