Habib University
Dhanani School of Science and Engineering

EE/CS 371L/330L T1
Computer Architecture

Research Assistant: Abeera Farooq Alam

Pipelined RISC-V Processor

by

Ifrah Chishti[1], Adil Saleem[2], Syeda Hoorain Imran[3]

April 23, 2025

[1]Student ID: 08351, Email: ic08351@st.habib.edu.pk
[2]Student ID: 08818, Email: as08818@st.habib.edu.pk
[3]Student ID: 08704, Email: si08704@st.habib.edu.pk

Contents

1 Introduction

The primary goal of this project was to develop a custom processor using the RISC-V architecture to execute a sorting routine efficiently. Starting from a single-cycle processor architecture, the design was iteratively enhanced to support pipelining. This allowed for parallel instruction execution and performance improvements. The sorting algorithm used—bubble sort—was executed over an array stored in memory, highlighting the effect of architectural enhancements on runtime behavior.

2 Single Cycle Processor: Sorting Implementation

2.1 Assembly Program Overview

Listing 1: Bubble Sort Assembly Code

```
# Load array into memory at 0x100
    lui x5, 0x0
    addi x5, x5, 0x100      # x5 = 0x100 (base address)

    li x6, 34
    sw x6, 0(x5)

    li x6, 12
    sw x6, 4(x5)

    li x6, 78
    sw x6, 8(x5)

    li x6, 23
    sw x6, 12(x5)

    li x6, 56
    sw x6, 16(x5)

    li x6, 9
    sw x6, 20(x5)

    li x6, 1
    sw x6, 24(x5)

    li x6, 45
    sw x6, 28(x5)

    # Start Bubble Sort
    addi x20, x0, 0         # i = 0
    addi x19, x0, 8         # n = 8 (number of elements)

outerloop:
    beq x20, x19, outerexit # if i == n, done
    addi x21, x0, 0         # j = 0

innerloop:
    add x30, x20, x0
    sub x30, x19, x30       # x30 = n - i
    addi x30, x30, -1       # x30 = n - i - 1
    beq x21, x30, innerexit # if j == n-i-1, inner loop done

```

```
43      slli x22, x21, 2          # x22 = j * 4 (offset)
44      add x28, x5, x22          # x28 = base + offset = &a[j]
45
46      lw x23, 0(x28)            # a[j]
47      lw x24, 4(x28)            # a[j+1]
48      blt x24, x23, bubblesort
49
50      addi x21, x21, 1          # j++
51      jal x0, innerloop
52
53  bubblesort:
54      sw x24, 0(x28)            # a[j] = a[j+1]
55      sw x23, 4(x28)            # a[j+1] = a[j]
56      addi x21, x21, 1          # j++
57      jal x0, innerloop
58
59  innerexit:
60      addi x20, x20, 1          # i++
61      jal x0, outerloop
62
63  outerexit:
64      # done
```

2.2 Binary Translation

Machine Code	Basic Code	Original Code
0x00000913	addi x18 x0 0	addi x18, x0, 0 # to track a[i] offset
0x00000433	add x8 x0 x0	add x8, x0, x0 # i iterator (starts at 0)
0x00a00593	addi x11 x0 10	addi x11, x0, 10 # loop bound n = 10
0x04b40663	beq x8 x11 76	beq x8, x11, outerexit # if i == n, exit
0x00800ab3	add x29 x0 x8	add x29, x0, x8 # j iterator = i
0x000409b3	add x19 x8 x0	add x19, x8, x0
0x013989b3	add x19 x19 x19	add x19, x19, x19
0x013989b3	add x19 x19 x19	add x19, x19, x19 # x19 = 4*x8 (byte offset)
0x02be8663	beq x29 x11 44	beq x29, x11, innerexit # if j == n, inner loop done
0x001e8e93	addi x29 x29 1	addi x29, x29, 1 # j++
0x00898993	addi x19 x19 8	addi x19, x19, 8 # offset += 8
0x0009d03	lw x26 0(x18)	lw x26, 0(x18) # load a[i]
0x0009ad83	lw x27 0(x19)	lw x27, 0(x19) # load a[j]
0x01bd4463	blt x26 x27 8	blt x26, x27, bubblesort # if a[i] < a[j], swap
0xfe0004e3	beq x0 x0 -24	beq x0, x0, innerloop # else continue
0x01a002b3	add x5 x0 x26	add x5, x0, x26 # temp = a[i]
0x01b90023	sw x27 0(x18)	sw x27, 0(x18) # a[i] = a[j]
0x0059a023	sw x5 0(x19)	sw x5, 0(x19) # a[j] = temp
0xfe000ce3	beq x0 x0 -40	beq x0, x0, innerloop # restart inner
0x00140413	addi x8 x8 1	addi x8, x8, 1 # i++
0x00890913	addi x18 x18 8	addi x18, x18, 8 # offset += 8
0xfa000ce3	beq x0 x0 -72	beq x0, x0, outerloop # back to outer

Figure 1: Translated machine code for bubble sort

2.3 Processor Configuration

Using our existing components from earlier labs, including instruction memory, register file, ALU, and control logic, we created a functional single-cycle RISC-V processor. This processor was configured to simulate bubble sort on eight 64-bit integers stored at fixed memory locations. Memory initialization and waveform observation outputs were added for validation.

3 Pipelining Enhancements

3.1 Transition to Multi-Stage Design

To achieve instruction-level parallelism, pipeline registers were inserted between stages: IF/ID, ID/EX, EX/MEM, and MEM/WB. These intermediate buffers allowed each instruction to progress through different pipeline phases simultaneously.

3.2 Encountered Issues

In its initial form, the pipelined processor did not successfully complete sorting. This was traced to data hazards between dependent instructions, which caused incorrect execution due to outdated operand values.

3.3 Task 2: Waveforms

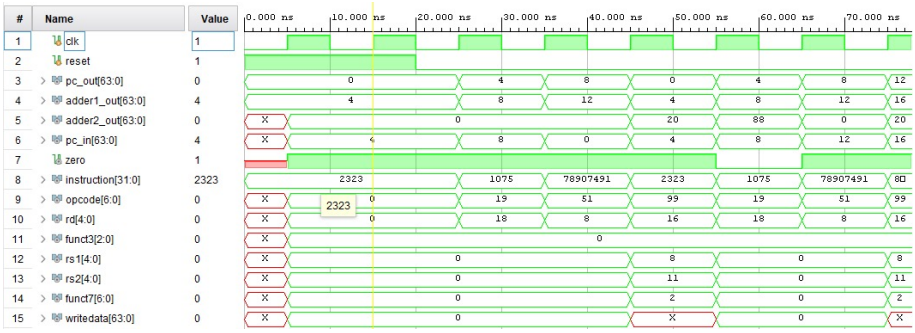

Figure 2: Task 2 waveforms: (1)

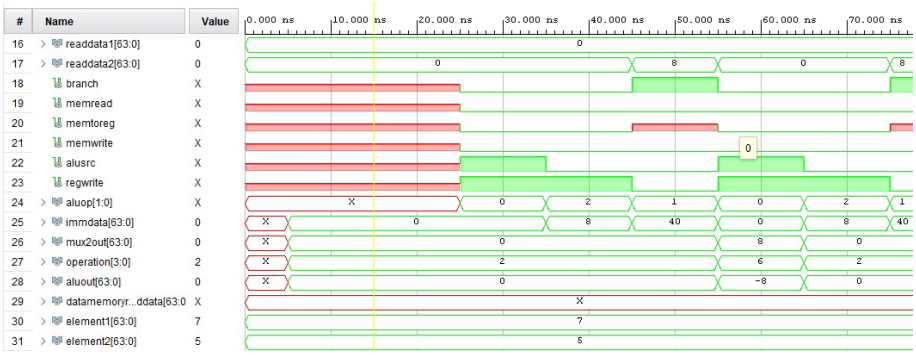

Figure 3: Task 2 waveforms: (2)

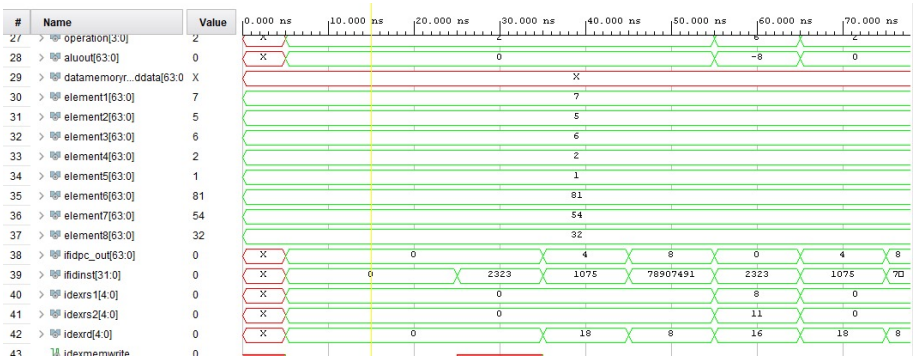

Figure 4: Task 2 waveforms: (3)

4 Hazard Mitigation Mechanisms

4.1 Forwarding and Detection Logic

To resolve incorrect behavior, forwarding units were added alongside hazard detection and pipeline flushing mechanisms. These components identified dependency issues and either forwarded data or stalled the pipeline as needed.

4.2 Modules Introduced

- `Hazard_Unit.v` – Monitors source-destination overlaps
- `Forwarding_Unit.v` – Enables bypass paths
- `Pipeline_Flush.v` – Clears intermediate states on branch
- Each pipe-lined register has its own module

5 Performance Evaluation

When comparing the non-pipelined and pipelined versions of the processor, the latter demonstrated enhanced throughput. Bubble sort execution time decreased from approximately 1000ns to 800ns due to overlapping instruction phases.

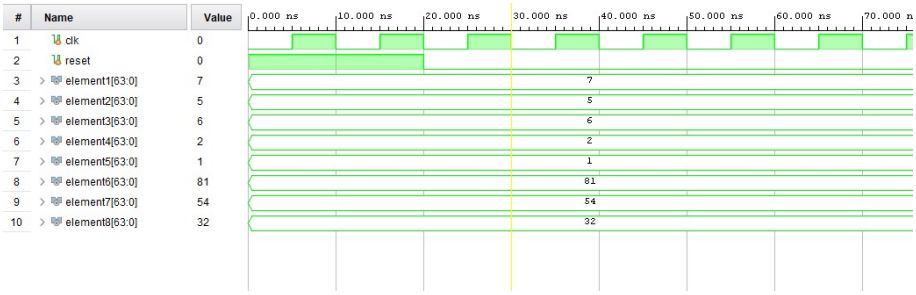

Figure 5: Task 3 unsorted values

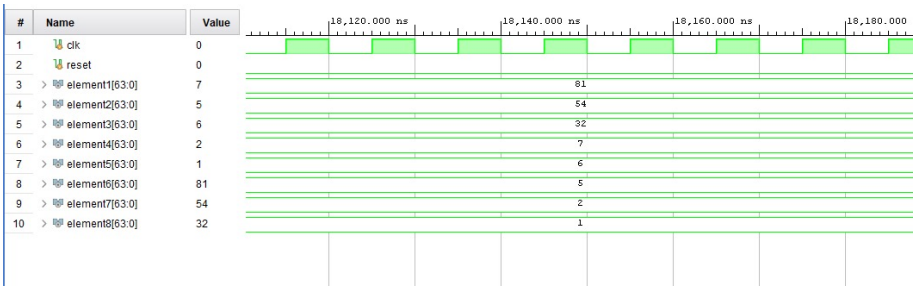

Figure 6: Task 3 sorted values

6 Challenges and Solutions

Initially we faced issue with all values not being sorted and then later we had to debug the code to fix that so we have 7 sorted values.

7 Conclusion

The processor we built successfully transitioned from a single-cycle to a pipelined design capable of executing a sorting algorithm. The enhancements in architecture significantly improved performance and reliability. Overall, this project deepened our understanding of RISC-V pipelines, hazard management, and system-level debugging.

8 Github repository link

`https://github.com/IfrahC/RISC-V-Pipelined-Processor`