Final Project Report: RISC-V Processor

Computer Architecture
CE-321L/CS-330L

Ammar Rehman - ar09711
Haider Raza Naqvi - am08853
Muhammad Sadiq Ali - ma10379

Instructor:
Miss Abeera Farooq

3rd December 2026

Contents

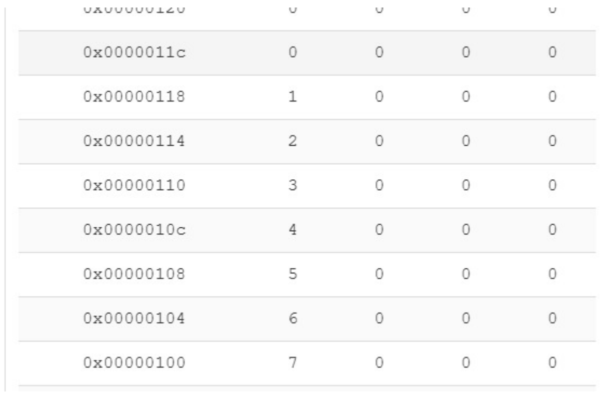
1 Introduction

1.1 Problem Statement

As computing demands grow, there is a need for efficient processor architectures that balance performance and functionality. This project addresses the design and implementation of a pipelined RISC-V processor capable of executing sorting on an array of 7 elements. The processor handles pipeline hazards and memory operations, reflecting core aspects of modern CPU design.

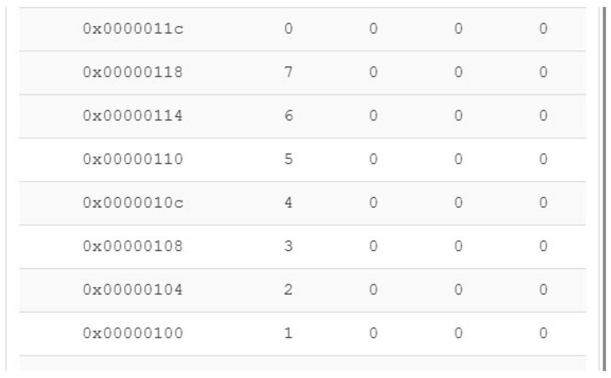
1.2 Objective

To build a 5-stage pipelined processor capable of executing any one array sorting algorithm such as bubble sort, we will need to convert a single-cycle processor into a pipelined one. In most cases, the instructions we have already implemented will allow us to run a sorting algorithm program with minor additions, such as implementing `beq` or `blt` instructions to detect when you need to swap two values. For Example:



0x0000011c	0	0	0	0
0x00000118	1	0	0	0
0x00000114	2	0	0	0
0x00000110	3	0	0	0
0x0000010c	4	0	0	0
0x00000108	5	0	0	0
0x00000104	6	0	0	0
0x00000100	7	0	0	0

Figure 1: Before Sorting



0x0000011c	0	0	0	0
0x00000118	7	0	0	0
0x00000114	6	0	0	0
0x00000110	5	0	0	0
0x0000010c	4	0	0	0
0x00000108	3	0	0	0
0x00000104	2	0	0	0
0x00000100	1	0	0	0

Figure 2: After Sorting

1.3 Sorting Algorithm

We will be using bubble sort to test our processor functionality.

2 Methodology

2.1 Task 1: Processor Enhancements for Bubble Sort

The goal of our project was to extend the single-cycle RISC-V processor from Lab 11 so that it could successfully run a bubble sort program. The original processor supported a limited set of instructions (mainly arithmetic, `BEQ`, and `BNE`), which was insufficient for implementing bubble sort because the algorithm requires relational comparisons and address computations. Therefore, we added support for `BLT`, `BGE`, and effective `SLLI` behavior, and updated the memory module to correctly handle 32-bit loads and stores.

2.1.1 Need for Additional Instructions

Bubble sort relies heavily on comparing array elements and iterating through nested loops. Equality-only branching (`BEQ`/`BNE`) was not enough. The algorithm required:

- **BLT**: Branch if less than.
- **BGE**: Branch if greater than or equal.
- **SLLI**: Shift left logical immediate (for calculating index $\times$ 4).

Since our existing processor didn't support these instructions, modifications were required.

2.1.2 Extending ALU Zero Signal Logic

Original ALU Behavior:
Our ALU (`ALU_64_behavioral`) originally produced a single comparison signal: a `zero flag` asserted when ALU_result == 0. This was the only branch-related output.
Extending the Comparison Mechanism:
Instead of creating new output flags or changing the ALU structure, we extended the logic behind the zero signal so that the same comparison mechanism could be reused for all four branch instructions:

- `BEQ` $\rightarrow$ branch if $zero == 1$.
- `BNE` $\rightarrow$ branch if $zero == 0$.
- `BLT` $\rightarrow$ zero signal driven by ALU's internal "less than" comparison.
- `BGE` $\rightarrow$ zero signal driven by ALU's internal "greater than or equal" comparison.

We reused the same zero flag but changed what it meant depending on the branch type. For `BLT`, the ALU generates a "less-than" result that drives $zero = 1$. This avoided adding multiple comparison outputs and kept the processor design simple.

2.1.3 Implementing SLLI Using ADD Instructions

Bubble sort required calculating memory addresses using index $\times$ 4. Instead of adding an SLLI instruction to the hardware datapath, we handled this at the instruction level by replacing:

```
slli x3, x3, 2
```

with:

```
add x3, x3, x3    # multiply by 2
add x3, x3, x3    # multiply by 4
```

Since our bubble sort only required shifting by 2 bits, this software-level approach avoided adding new shift hardware.

2.1.4 Updating Data Memory for 32-bit LW and SW

The original data memory produced and stored full 64-bit values. However, RV32 instructions (`lw` and `sw`) operate on 32-bit words. To ensure correct behavior for bubble sort (which loads/stores 32-bit array elements), we updated memory functionality:

- `lw` now extracts and outputs only the least significant 32 bits, zero-extended to 64 bits before writing to the register.
- `sw` writes only the lower 32 bits of the register value into memory.

2.1.5 Summary of Enhancements

To run bubble sort on our single-cycle processor, we introduced:

- **ALU & Branching:** Reused the zero signal to support `BLT` and `BGE` without adding extra flags.
- **Shifting:** Replaced `SLLI` with multiple `ADD` instructions to save hardware.
- **Memory:** Modified Data Memory to correctly handle 32-bit loads and stores.

2.1.6 Waveform and Schematic

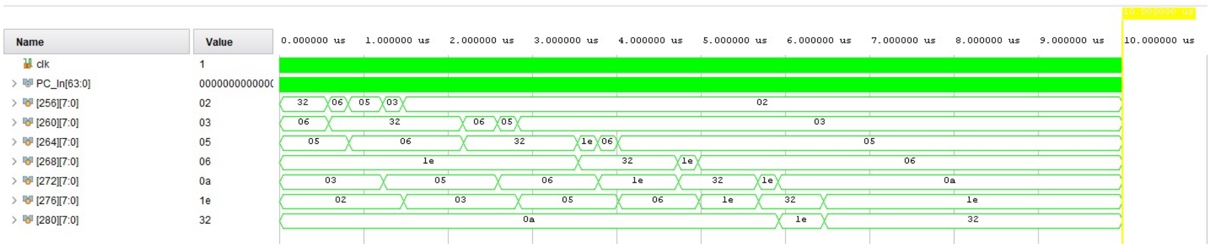


Figure 3: Waveform

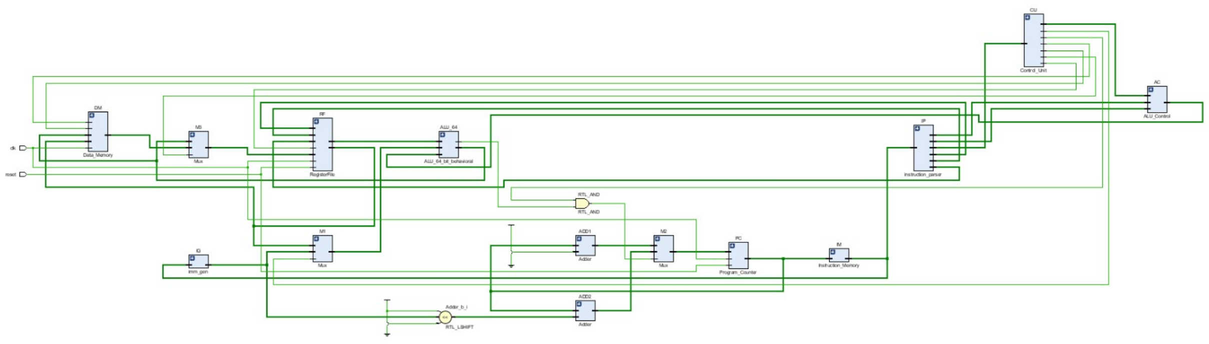


Figure 4: RTL Schematic

2.2 Task 2

2.2.1 Testing 5-Stage/Pipelined RISC-V Processor

Here we upgraded our previously developed processor for Pipeline Execution by implementing a complete 5-stage pipeline architecture. The design integrates four intermediate pipeline registers between each stage and a top-level module that orchestrates the entire pipeline operation.

2.2.2 Changes

To implement an efficient pipelined design, we added the following key components:

- **Four Pipeline Registers:**
  - IF/ID (Instruction Fetch to Instruction Decode)
  - ID/EX (Instruction Decode to Execute)
  - EX/MEM (Execute to Memory Access)
  - MEM/WB (Memory Access to Write Back)
- **Top-Level Module:** A comprehensive control unit that integrates all pipeline stages, manages global control signals, coordinates stage handoffs, and maintains pipeline synchronization. Here is the RTL Schematic:

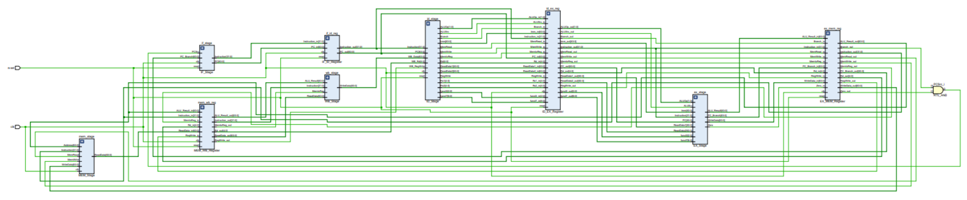


Figure 5: Schematic of the 5-Stage Pipelined (Incomplete) Processor

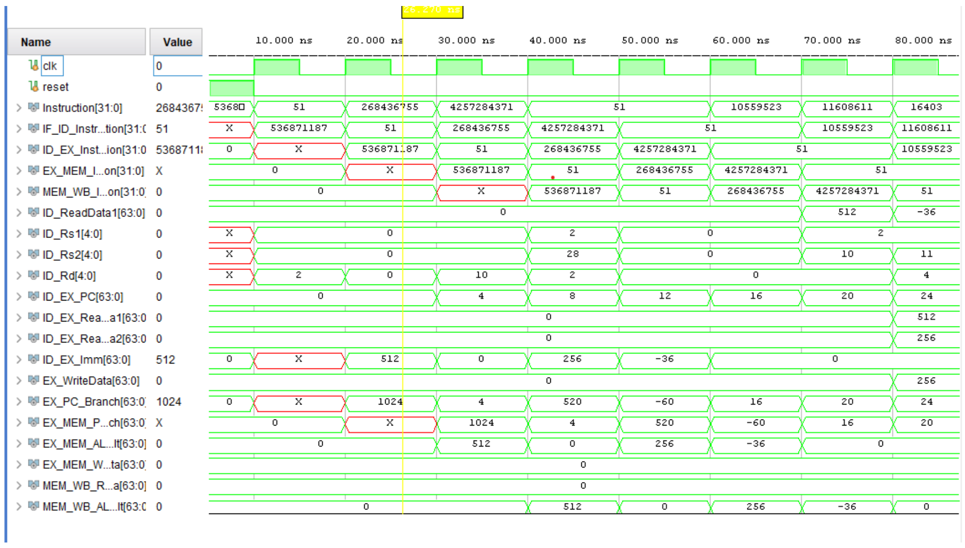


Figure 6: Waveform of the 5-Stage Pipelined (Incomplete) Processor

2.3 Task 3

2.3.1 Handling Data Hazards

To address hazards in our circuitry, we need to develop modules for detecting hazards and stalling the pipeline. We introduced two new modules: the Forwarding Unit and the Hazard Detection Unit.

2.3.2 The Forwarding Unit

Let rs be the source register index (either $rs1$ or $rs2$) in the decode stage. We define the hazard conditions for the Execution (EX) and Memory (MEM) stages as follows:

$$\begin{aligned}\text{Hazard}_{EX} &= (RegWrite_{EX/MEM} \wedge (rd_{EX/MEM} \neq 0) \wedge (rd_{EX/MEM} == rs))\\ \text{Hazard}_{MEM} &= (RegWrite_{MEM/WB} \wedge (rd_{MEM/WB} \neq 0) \wedge (rd_{MEM/WB} == rs))\end{aligned}$$

The Forwarding Control signal (ForwardA or ForwardB) is determined by the following priority logic:

$$\text{Forward Signal} = \begin{cases} 10_2 & \text{if Hazard}_{EX} \\ 01_2 & \text{if Hazard}_{MEM} \wedge \neg(\text{Hazard}_{EX}) \\ 00_2 & \text{otherwise} \end{cases}$$

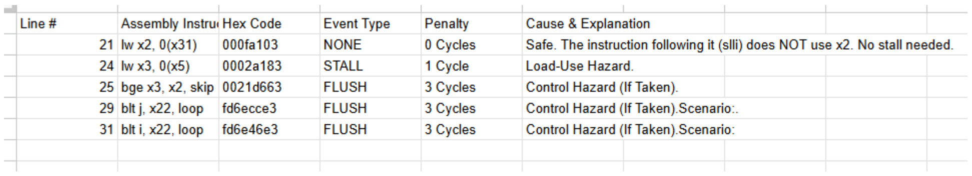
2.3.3 The Hazard Detection Unit

While the Forwarding Unit resolves most data hazards, scenarios like the load-use hazard still require stalling. The HDU outputs three key control signals:

- **stall pc:** Prevents the Program Counter from updating.
- **stall ifid:** Prevents the IF/ID pipeline register from updating.
- **flush idex:** Clears the contents of the ID/EX pipeline register to insert a bubble.

2.3.4 Flushes and Stalls Implemented in Task3

The Stalls and flushes that are expected in our code are as follows:



Line #	Assembly Instru	Hex Code	Event Type	Penalty	Cause & Explanation
21	lw x2, 0(x31)	000fa103	NONE	0 Cycles	Safe. The instruction following it (slli) does NOT use x2. No stall needed.
24	lw x3, 0(x5)	0002a183	STALL	1 Cycle	Load-Use Hazard.
25	bge x3, x2, skip	0021d663	FLUSH	3 Cycles	Control Hazard (If Taken).
29	blt j, x22, loop	fd6ecce3	FLUSH	3 Cycles	Control Hazard (If Taken).Scenario:.
31	blt i, x22, loop	fd6e46e3	FLUSH	3 Cycles	Control Hazard (If Taken).Scenario:

Figure 7: Assembly Code

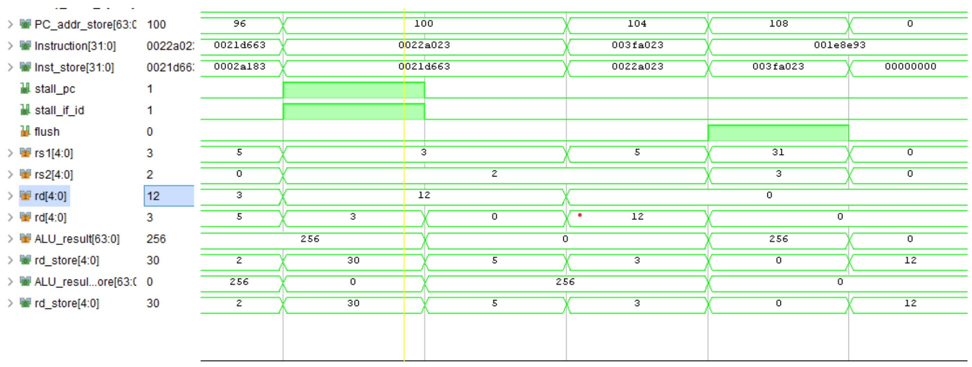
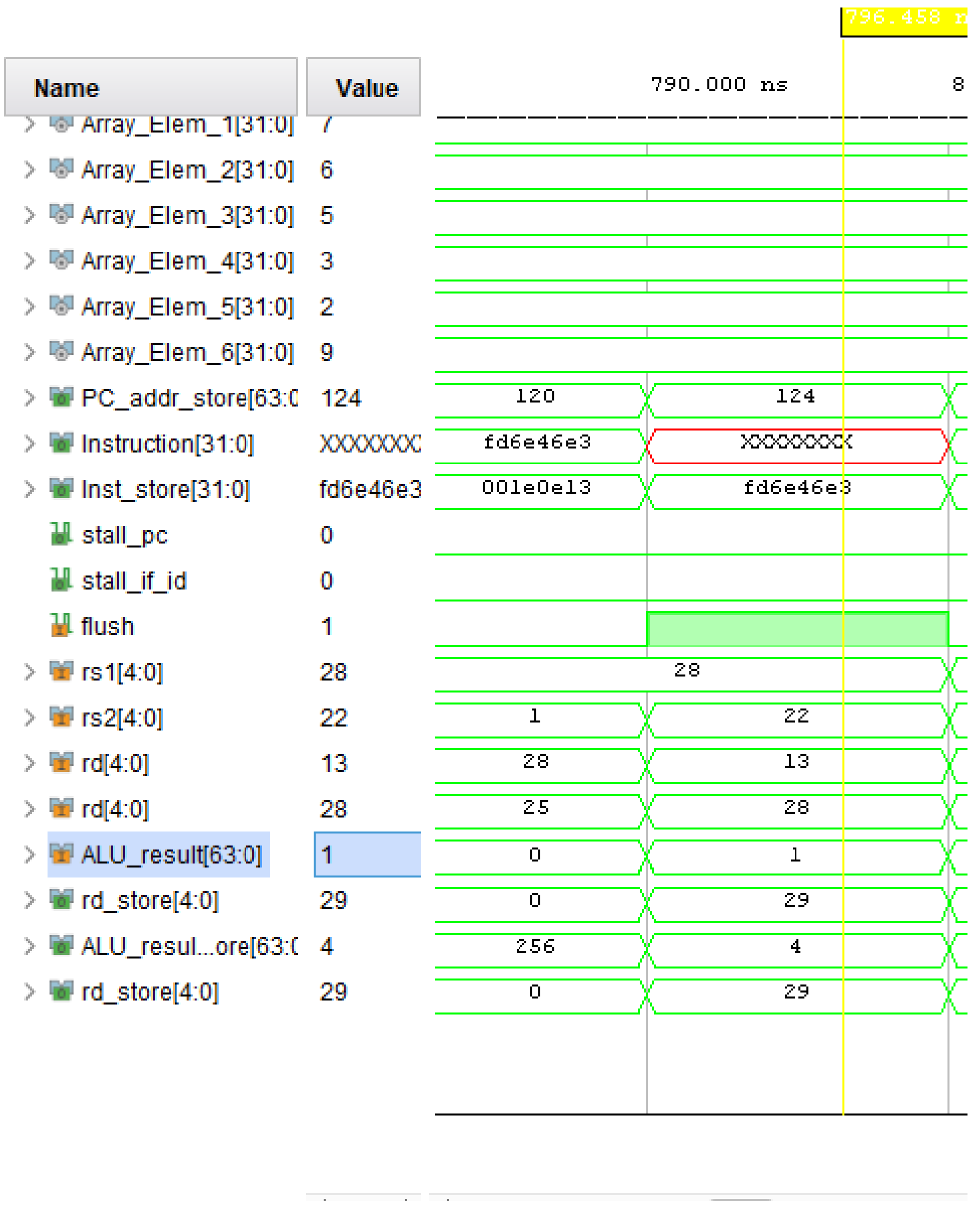
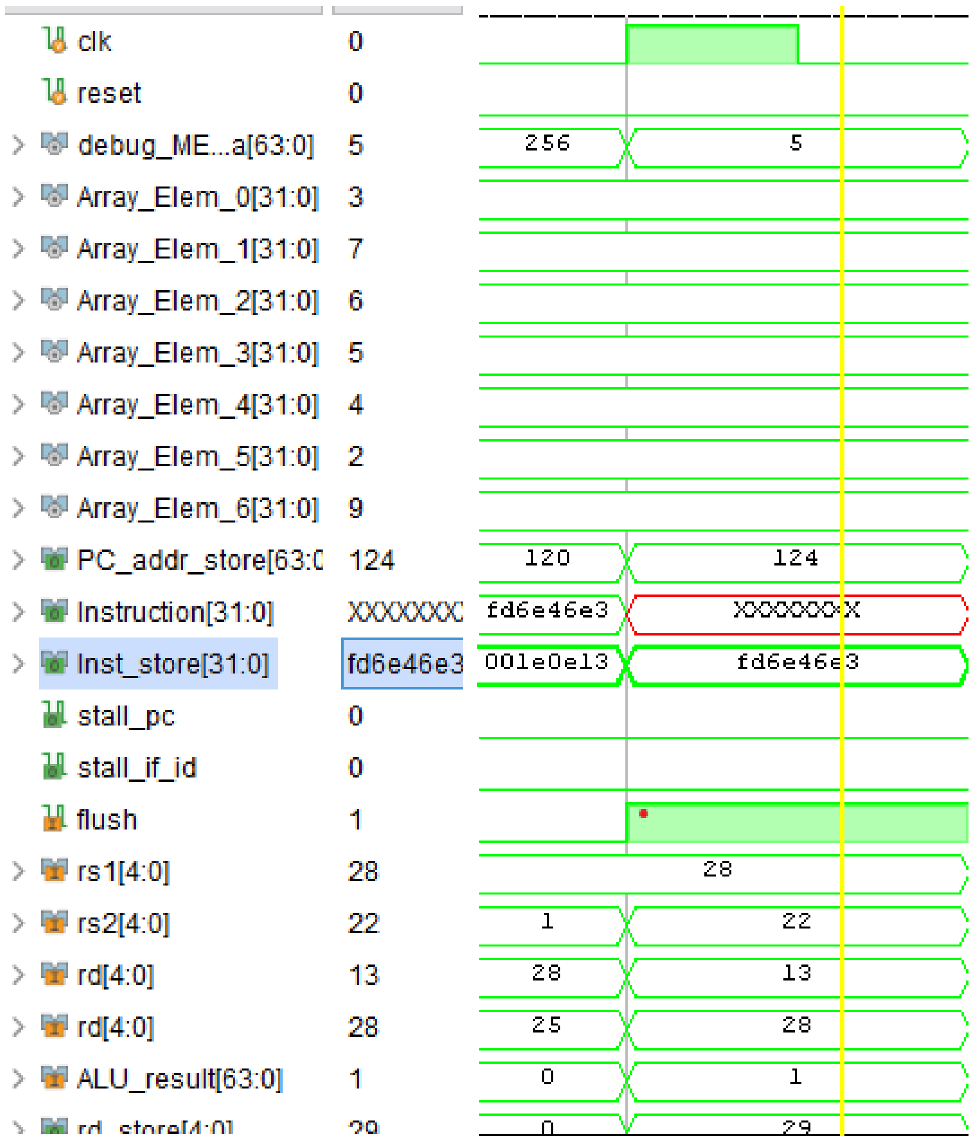


Figure 8: For line 24 & 25 (Stall and then flush)



Name	Value
Array_Elem_1[31:0]	7
Array_Elem_2[31:0]	6
Array_Elem_3[31:0]	5
Array_Elem_4[31:0]	3
Array_Elem_5[31:0]	2
Array_Elem_6[31:0]	9
PC_addr_store[63:0	124
Instruction[31:0]	XXXXXXX
Inst_store[31:0]	fd6e46e3
stall_pc	0
stall_if_id	0
flush	1
rs1[4:0]	28
rs2[4:0]	22
rd[4:0]	13
rd[4:0]	28
ALU_result[63:0]	1
rd_store[4:0]	29
ALU_resul...ore[63:0	4
rd_store[4:0]	29

Figure 9: for line 29



```
clk                      0
reset                    0
debug_ME...a[63:0]       5          256 | 5
Array_Elem_0[31:0]       3
Array_Elem_1[31:0]       7
Array_Elem_2[31:0]       6
Array_Elem_3[31:0]       5
Array_Elem_4[31:0]       4
Array_Elem_5[31:0]       2
Array_Elem_6[31:0]       9
PC_addr_store[63:0       124        120 | 124
Instruction[31:0]        XXXXXXXX   fd6e46e3 | XXXXXXXX
Inst_store[31:0]         fd6e46e3   001e0e13 | fd6e46e3
stall_pc                 0
stall_if_id              0
flush                    1
rs1[4:0]                 28         28
rs2[4:0]                 22         1 | 22
rd[4:0]                  13         28 | 13
rd[4:0]                  28         25 | 28
ALU_result[63:0]         1          0 | 1
rd_store[4:0]            29         0 | 29
```

Figure 10: for line 31

2.3.5 Final Results of the array:

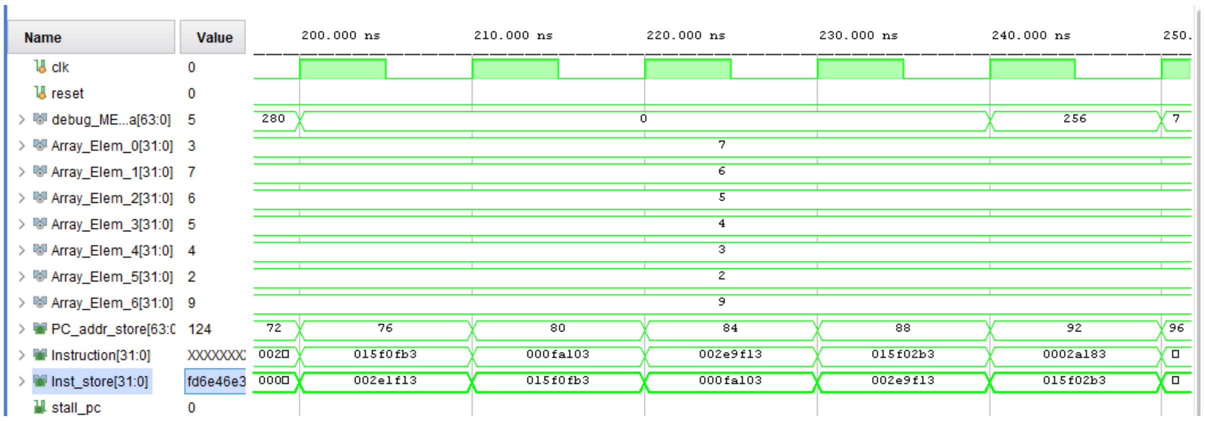


Figure 11: Unsorted array

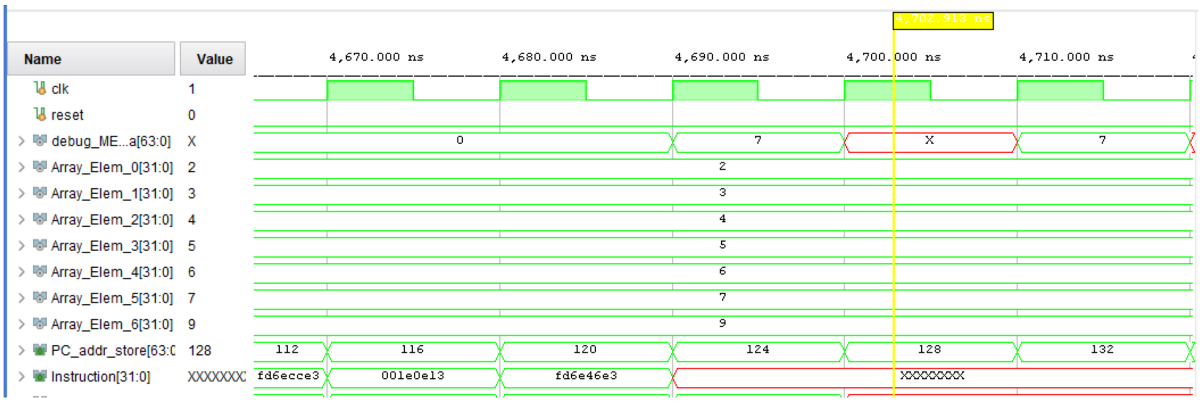


Figure 12: Sorted array

2.4 Task 4: Performance Analysis

2.4.1 Single-Cycle vs Pipelined Processor

Comparison	Single-Cycle	Pipelined
Instruction Count	684	238
Clock Period	5 ns	5 ns
Total Execution Time	7250 ns	4700 ns
Average CPI	2.12	3.95

Table 1: Measured Performance Metrics

$$\text{Speedup} = \frac{\text{Execution Time}_{\text{Single-Cycle}}}{\text{Execution Time}_{\text{Pipelined}}} = \frac{7250 \text{ ns}}{4700 \text{ ns}} \approx 1.54 \tag{1}$$

3 Challenges

3.1 Implementation Challenges

- **Pipeline Hazards:** Control dependencies caused frequent stalls, requiring careful scheduling of NOP operations.
- **Timing Constraints:** The 5-stage pipeline reduced the clock period by 22.4%, requiring careful balancing of stage delays.

4 Task Division

- **Muhammad Sadiq Ali:** Pipelined Processor and Debugging (Task 2) and Report
- **Ammar Rehman:** Design and Implementation of Hazard Detection Unit and Forwarding Unit (Task 3) and Report
- **Haider Raza Naqvi:** Single-Cycle and Static Branch Prediction Implementation

5 Conclusion

This project successfully implemented a 5-stage pipelined RISC-V processor capable of executing a bubble sort algorithm on 7-element arrays. The final architecture integrates forwarding units and hazard detection logic to optimize instruction flow.

5.1 Key Results

– Speedup of 1.54 via Hazard Detection and Forwarding Unit

– Forwarding unit resolved data hazards

– Hazard Detection Unit prevented load-use stalls and branch flushing.

– Sorted 7-element arrays in 492 cycles (normal cycles =238 , stall cycles= 254) and 4600 ns