

**CE-321L/CS-330L: Computer Architecture**  
**PIPELINED RISC-V PROCESSOR**  
*Anas Akhtar, Afshad Sidhwa, Hamdan ul Aziz*  
**24th April 2025**

**Introduction**

- Our project aims to design a 5-stage pipelined RISC-V Processor capable of executing a sorting algorithm. Our basic goal was to enhance the single-cycle processor built in Lab 11 with pipelining features, data handling, control hazards and forwarding, and show the sorting of at least 7 numbers (4 bytes) stored in an array at memory location 0x100. This project gave us hands-on experience and deepened our understanding of execution, forwarding, stalling and pipeline control.

**Task 1: Sorting Algorithm on Single-Cycle Processor**

We selected the bubble sort algorithm to implement sorting due to its simplicity and suitability for showing data dependencies as well as conditional branches. We first wrote the sorting algorithm in C-style pseudocode as below, then converted it into RISC-V assembly instructions.

*C-style pseudocode for bubble sort:*

```
5  void bubbleSort(int arr[], int n) {
6      for (int i = 0; i < n - 1; i++) {
7          for (int j = 0; j < n - i - 1; j++) {
8              if (arr[j] > arr[j + 1]) {
9                 
10                 int temp = arr[j];
11                 arr[j] = arr[j + 1];
12                 arr[j + 1] = temp;
13             }
14         }
15     }
16 }
```

## Equivalent bubble sort in RISC-V assembly:

```

addi x10, x0, 0x100

# Store sample values to sort: [29, 84, 78, 47, 132, 6, 17]
li x20, 29      # First value
sw x20, 0(x10)   # Store at address 0x100

li x20, 84      # Second value
sw x20, 4(x10)   # Store at address 0x108

li x20, 78      # Third value
sw x20, 8(x10)   # Store at address 0x110

li x20, 47      # Fourth value
sw x20, 12(x10)  # Store at address 0x118

li x20, 132     # Fifth value
sw x20, 16(x10)  # Store at address 0x120

li x20, 6       # Sixth value
sw x20, 20(x10)  # Store at address 0x128

li x20, 17      # Seventh value
sw x20, 24(x10)  # Store at address 0x130

# Reset x10 to array base address (just to be sure)
addi x10, x0, 0x100

addi x11, x0, 7   # Initialize x11 with value 7
bne x10, x0, ELSE # If x10 != 0, branch to ELSE
bne x11, x0, ELSE # If x11 != 0, branch to ELSE
beq x0, x0, EXIT1  # Unconditional branch to EXIT1

ELSE:
addi x18, x0, 0    # Initialize x18 with 0

BUBBLE1:
beq x18, x11, EXIT1 # If x18 == x11, branch to EXIT1
add x19, x0, x18    # Initialize x19 with value in x18

BUBBLE2:
beq x19, x11, EXIT2 # If x19 == x11, branch to EXIT2
slli x5, x18, 2     # x5 = x18 * 4
slli x6, x19, 2     # x6 = x19 * 4
add x5, x5, x10     # x5 = x5 + x10 (address of array[x18])
add x6, x6, x10     # x6 = x6 + x10 (address of array[x19])
lw x28, 0(x5)       # Load value at address x5 into x28
lw x29, 0(x6)       # Load value at address x6 into x29
bge x28, x29, SKIP   # If x28 >= x29, branch to SKIP
add x30, x0, x28     # x30 = x28 (temp variable for swap)
add x28, x0, x29     # x28 = x29
add x29, x0, x30     # x29 = x30 (original x28)
sw x28, 0(x5)       # Store x28 at address x5
sw x29, 0(x6)       # Store x29 at address x6

SKIP:
addi x19, x19, 1    # Increment x19
beq x0, x0, BUBBLE2 # Unconditional branch to BUBBLE2










EXIT2:
addi x18, x18, 1    # Increment x18
beq x0, x0, BUBBLE1 # Unconditional branch to BUBBLE1

EXIT1:
beq x0, x0, EXIT    # Unconditional branch to EXIT
addi x0, x0, 0       # No-op (this instruction is never executed)

EXIT:
# Added this missing label

```

## Before Sorting:

Name	Value	0 ps	1 ps	2 ps	3 ps	4 ps	5 ps	6 ps
 clk	0							
 reset	0							
>  index1[63:0]	132				29			
>  index2[63:0]	84				84			
>  index3[63:0]	78				78			
>  index4[63:0]	47				47			
>  index5[63:0]	29				132			
>  index6[63:0]	17				6			
>  index7[63:0]	6				17			

### After Sorting:

Signal	Value	Time (ns)	0	100	200	300	400	500	600	700	800	900	1000
clk	0												
reset	0												
> index1[63:0]	132		132										
> index2[63:0]	84		84										
> index3[63:0]	78		78										
> index4[63:0]	47		47										
> index5[63:0]	29		29										
> index6[63:0]	17		17										
> index7[63:0]	6		6										

## Task 2: Pipelined Processor Implementation

To build upon our single-cycle processor from Task 1, we transformed it into a 5-stage pipelined RISC-V processor. The goal was to allow instructions to execute in overlapping stages, thereby improving efficiency. The pipelined processor is divided into the following stages, with each having its dedicated module and pipeline register:

1. *Instruction Fetch (IF)*
  - Fetches instructions from memory.
  - Handled in module: `IF\_ID.v`
2. *Instruction Decode (ID)*
  - Decodes instructions and reads source registers.
  - Generates control signals.
  - Handled in module: `ID\_EX.v`
3. *Execute (EX)*
  - Executes ALU operations or calculates memory addresses.
  - Handled in module: `EX\_MEM.v`
4. *Memory Access (MEM)*
  - Accesses memory for load/store operations.
  - Handled in module: `MEM\_WB.v`
5. *Write Back (WB)*
  - Writes results back into the register file.
  - Handled in module: `MEM\_WB.v`

To transition from the single-cycle to the pipelined architecture, we introduced pipeline registers ('IF ID', 'ID EX', 'EX MEM', 'MEM WB') to pass data and control signals between stages.

While separating instruction processing into the five distinct stages listed above, we updated the processor design to allow multiple instructions to be in different stages simultaneously.

### **Task 3: Hazard Detection and Handling**

We then implemented a forwarding and hazard detection unit - “Forwarding\_Unit”, “Hazard\_Detection”- to resolve data hazards between EX, MEM and WB stages using control signals as studied in the lectures. It detects whether the result of a previous instruction is required by the current instruction in the EX stage or whether the result is not yet written back to the register file.

### **Forwarding Unit:**

<pre> module Forwarding_Unit (   input [4:0] EXMEM_rd, MEMWB_rd,   input [4:0] IDEX_rs1, IDEX_rs2,   input EXMEM_RegWrite, EXMEM_MemtoReg,   input MEMWB_RegWrite,   output reg [1:0] fwd_A, fwd_B );  always @(*) begin    if (EXMEM_rd == IDEX_rs1 &amp;&amp; EXMEM_RegWrite &amp;&amp; EXMEM_rd != 0)     begin       fwd_A = 2'b10;     end   else if (MEMWB_RegWrite &amp;&amp; MEMWB_rd!=0 &amp;&amp; MEMWB_rd==IDEX_rs1 )     begin       fwd_A = 2'b01;     end   else     begin       fwd_A = 2'b00;     end end </pre>	<div style="display: flex; align-items: center;">○</div> <pre>   if (EXMEM_rd == IDEX_rs1 &amp;&amp; EXMEM_RegWrite &amp;&amp; EXMEM_rd != 0)     begin       fwd_A = 2'b10;     end   else if (MEMWB_RegWrite &amp;&amp; MEMWB_rd!=0 &amp;&amp; MEMWB_rd==IDEX_rs1 )     begin       fwd_A = 2'b01;     end   else     begin       fwd_A = 2'b00;     end end </pre> <div style="display: flex; align-items: center;">○</div> <pre>   if ((EXMEM_rd == IDEX_rs2) &amp;&amp; (EXMEM_RegWrite) &amp;&amp; (EXMEM_rd != 0))     begin       fwd_B = 2'b10;     end   else if (MEMWB_RegWrite &amp;&amp; MEMWB_rd!=0 &amp;&amp; MEMWB_rd==IDEX_rs2)     begin       fwd_B = 2'b01;     end   else     begin       fwd_B = 2'b00;     end end </pre> <div style="display: flex; align-items: center;">○</div> <pre> endmodule // Forwarding_Unit </pre>
--	---

The module receives the following inputs:

- **IDEX\_rs1, IDEX\_rs2**: Source registers of the current instruction in the **ID/EX** stage.
- **EXMEM\_rd, MEMWB\_rd**: Destination registers from the **EX/MEM** and **MEM/WB** pipeline stages.
- **EXMEM\_RegWrite, MEMWB\_RegWrite**: Control signals indicating whether those stages are writing back results.
- **EXMEM\_MemtoReg**: Additional signal that indicates if the value being written back is from memory (used in extension).

The output is **fwd\_A** and **fwd\_B**: Two 2-bit control signals used to determine the forwarding path for operands A and B, respectively.

### Forwarding Logic

- If the **EX/MEM stage** is writing to a register and its destination register matches one of the current source registers, the data is forwarded from the EX stage (**fwd = 2'b10**).
- If the match is with the **MEM/WB stage**, and the EX/MEM stage isn't involved, the data is forwarded from the WB stage (**fwd = 2'b01**).
- If no match is found, the data is read directly from the register file (**fwd = 2'b00**).

### Hazard Detection:

We implemented a Hazard Detection Unit - module “Hazard\_Detection” - to identify and resolve load-use data hazards in the pipeline.

```
module Hazard_Detection (
    input [4:0] IDEX_rd,           // Destination register in EX stage
    input [4:0] IFID_rs1,         // Source register 1 in ID stage
    input [4:0] IFID_rs2,         // Source register 2 in ID stage
    input       IDEX_MemRead,      // EX stage instruction is a load
    output reg  IDEX_mux_out,      // Control signal to stall ID/EX
    output reg  IFID_Write,       // Control signal to write/stall IF/ID
    output reg  PCWrite           // Control signal for PC update
);

always @(*) begin
    // If there's a load-use hazard, stall pipeline
    if (IDEX_MemRead && (IDEX_rd == IFID_rs1 || IDEX_rd == IFID_rs2)) begin
        IDEX_mux_out = 0;
        IFID_Write   = 0;
        PCWrite      = 0;
    end else begin
        // No hazard, continue normal execution
        IDEX_mux_out = 1;
        IFID_Write   = 1;
        PCWrite      = 1;
    end
end

endmodule // Hazard_Detection
```

The hazard detection module receives the following:

### Inputs:

- **IDEX\_rd:** Destination register of the instruction in the EX stage
- **IFID\_rs1, IFID\_rs2:** Source registers of the instruction in the ID stage
- **IDEX\_MemRead:** Indicates if the instruction in EX is a memory read (load)

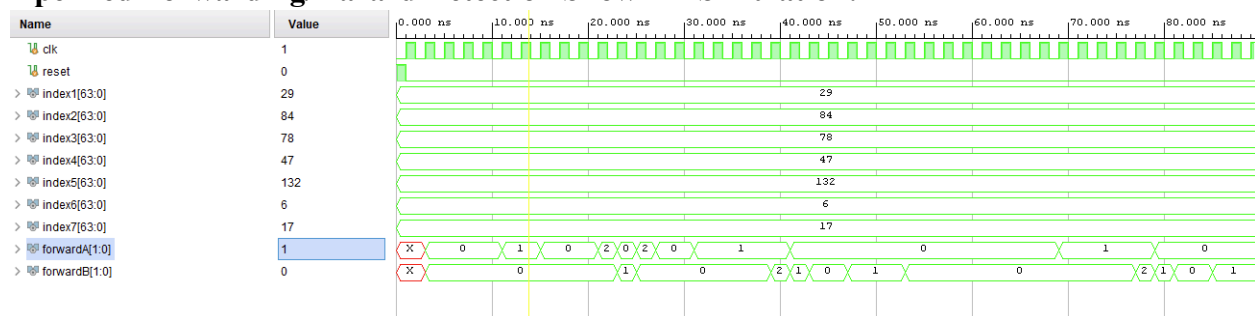
### Outputs:

- **IDEX\_mux\_out:** Signal to stall ID/EX register updates
- **IFID\_Write:** Enables or stalls updates to IF/ID register
- **PCWrite:** Enables or stalls program counter update

### Detection Logic:

- If the EX stage has a load instruction - MemRead - and its rd matches rs1 or rs2 in the ID stage, a stall is introduced by:
  - Disabling PC and IF/ID register write
  - Freezing control signals to insert a bubble
- Otherwise, the pipeline proceeds normally with all control signals enabled.

### Pipelined Forwarding/Hazard Detection Shown in Simulation:



From the above screenshot, we see that around timestamp 10,000 ns, we have a forwarding at rs1, taking place during the EXE stage. Similarly, around 20,000 ns, we have another forwarding at rs1, this time at the MEM stage.

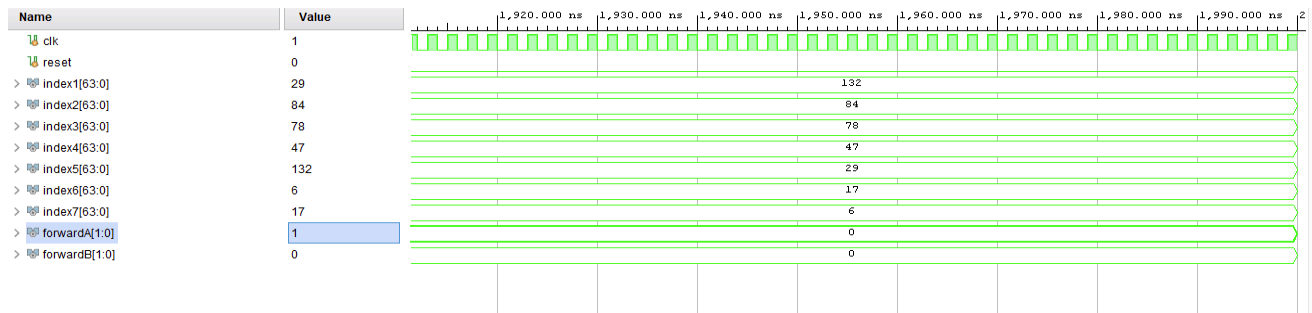
## Task 4: Performance Comparison (Single vs Pipelined Processor)

We compared the performance of the single-cycle processor and the pipelined processor based on instruction cycle count:

- **Single-Cycle Processor:** Took  $n$  cycles for  $n$  instructions.
- **Pipelined Processor:** Achieved overlap and took approximately  $n + 4$  cycles for  $n$  instructions, depending on hazard conditions.
- The performance of our Pipelined processor wasn't significant because we had less number of instructions in our program. This Pipeline processor will improve the performance significantly for a program with a large number of instructions.

### Results

- Final simulation



### Challenges

- We faced challenges in implementing the read-after-write hazard detection condition due to its complex structure. It needed precise detection and stalling between pipeline stages, which we found challenging.

### Task Division

- Anas - Task 1
- Anas, Hamdan and Afshad - Task 2 and 3
- Hamdan and Afshad - Report
- Overall, we did most of the project together in the lab

## Conclusions

- The final implementation successfully executed a sorting algorithm on both the single-cycle and pipelined RISC-V processors. In the pipelined version, all instructions were verified to execute correctly with proper handling of data and control hazards using forwarding and hazard detection units. The simulation results showed correct sorting of 7 integers stored at memory address **0x100**, and the pipeline operated efficiently with minimal stalls.
- Our project was successful overall. We understood all concepts and were able to implement them in Verilog and assembly. Our project also efficiently carried out the sorting. The only area we lacked was in implementing read-after-write hazard detection, though we still understood the working behind it and why it was giving us errors.

## Github Repo:

<https://github.com/aanasakhtar/RISCVPipelineProcessor>