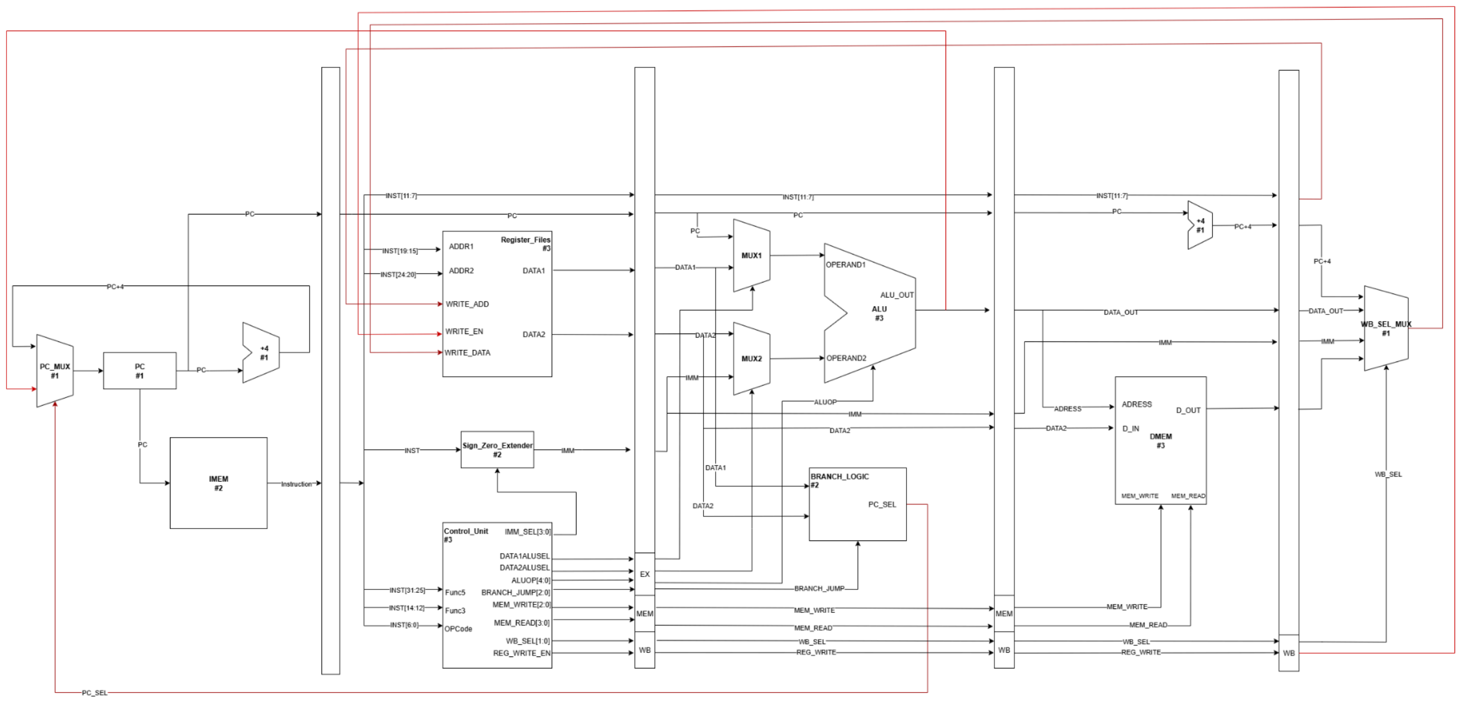
CO502 - Part 4 (Hazard Handling)

Group 1

Table of Contents

Overview

This report presents the final integration of our 5-stage pipelined RISC-V processor, enhanced with hazard detection and handling mechanisms. In the previous implementation, the basic pipeline stages—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)—were completed successfully without hazard handling. However, due to the presence of data and control hazards in pipelined execution, additional mechanisms were required to ensure correct and efficient instruction flow.



RISC-V 5 Stage Pipelined Processor without Hazard Detection

In the earlier version, NOPs were manually inserted to mitigate hazards. In this phase, we focus on resolving these hazards automatically through hardware techniques such as forwarding, hazard detection, and pipeline flushing.

Pipeline Hazards

Introduction to Hazards

In computer architecture, particularly in pipelined processors, hazards are conditions that prevent the next instruction from executing in its designated clock cycle. These hazards disrupt the ideal pipeline flow and degrade the expected performance benefits of pipelining. When a hazard occurs, the affected instruction (and potentially subsequent instructions) must be stalled, causing pipeline bubbles, clock cycles where no useful work is done. These bubbles occupy space in the pipeline, causing delays in execution.

These are broadly classified into three types:

Hazard Type	Description
Data Hazard	Occurs when instructions depend on the results of previous instructions still in the pipeline.
Control Hazard	Occurs when the flow of instruction execution changes due to a branch or jump.
Structural Hazard	Occurs when hardware resources are insufficient to support all concurrent pipeline operations.

Structural Hazards

These hazards arise from resource conflicts, such as when two or more pipeline stages require the same hardware resource simultaneously. For example, a shared memory unit used for both instruction fetch and data access can lead to a structural hazard if the processor tries to access both in the same cycle.

Data Hazards (RAW, WAR, WAW)

In our in-order execution pipeline, only Read-After-Write (RAW) hazards can occur. The other types, Write-After-Write (WAW) and Write-After-Read (WAR), are associated with out-of-order processors, which we are not implementing here.

Types of Data Hazards:

Type	Name	Description
RAW	Read After Write	An instruction tries to read a register before a previous instruction writes to it. (True dependence)
WAR	Write After Read	An instruction writes to a register before it is read by an earlier instruction. (Anti-dependence, not applicable to in-order pipelines)
WAW	Write After Write	An instruction writes to a register before an earlier instruction writes to it. (Output dependence, also only relevant in out-of-order execution)

Control Hazards

Control hazards arise when the next instruction address (PC) is not known due to a branch (beq, bne) or jump (jal, jr) instruction. Since the branch decision is made in a later pipeline stage (typically EX), the next instruction fetch is uncertain.

Hazard Handling Techniques

Data Hazard Handling – Forwarding (Bypassing)

To resolve RAW hazards without stalling the pipeline unnecessarily, we implemented data forwarding between pipeline stages. This allows dependent instructions to access results directly from intermediate pipeline registers rather than waiting for them to be written back to the register file.

Forwarding Paths Implemented:

- EX/MEM → EX stage
- MEM/WB → EX stage

Detection Logic:
If the destination register of an instruction in EX or MEM stage matches the source register of the instruction in ID/EX stage with a write enable signal, forwarding is triggered.

In a five-stage pipeline, by using negative-edge triggered register file writes, we effectively reduce the number of instructions that can cause RAW hazards to a two-instruction window. By comparing the destination register (along with its write-enable signal) from the EX and MEM stages with the source registers of the instruction currently in the decode stage (ID/EX), we can determine whether forwarding is needed.

This selective forwarding mechanism ensures correctness while minimizing pipeline stalls.

Hazard Detection Unit (Stall Logic)

Despite forwarding, some cases (like load-use hazards) require inserting a stall (Bubble). Our hazard detection unit detects when the instruction in EX is a load and the next instruction uses the result. In this case:

- A NOP is inserted.
- The pipeline stall signal holds PC and IF/ID registers for one cycle.

Control Hazard Handling – Pipeline Flushing

We do not use branch prediction in this design. Instead, control hazards are handled by flushing instructions after a branch or jump decision.

Branch Flush Strategy:

- The branch decision is made in the EX stage.
- Instructions in the IF and ID stages are flushed if the branch is taken.
- This prevents executing incorrect instructions after a mispredicted branch.

Here we used always not taken policy as the policy and if the branch is taken then only react to it.

Updated Datapath with Hazard Handling

Overview

The datapath has been updated with the following key components to handle data and control hazards effectively:

- Hazard Unit: Located in ID stage with main components as below,
  - Forwarding Unit: Use EX, MEM, and WB stages to detect and resolve data hazards by forwarding the most recent values to the ALU inputs.
  - Hazard Detection Unit: Use ID and EX stages to detect load-use hazards and insert stalls when necessary.
- Flush Logic (Branch Unit): Incorporated between the EX and ID/IF stages to handle control hazards by flushing incorrect instructions after a taken branch or jump.

Together, these additions form the Hazard Unit, which integrates both the forwarding logic and the hazard detection mechanism. This unit ensures that the datapath can dynamically identify and respond to pipeline hazards without requiring manual NOP insertion.

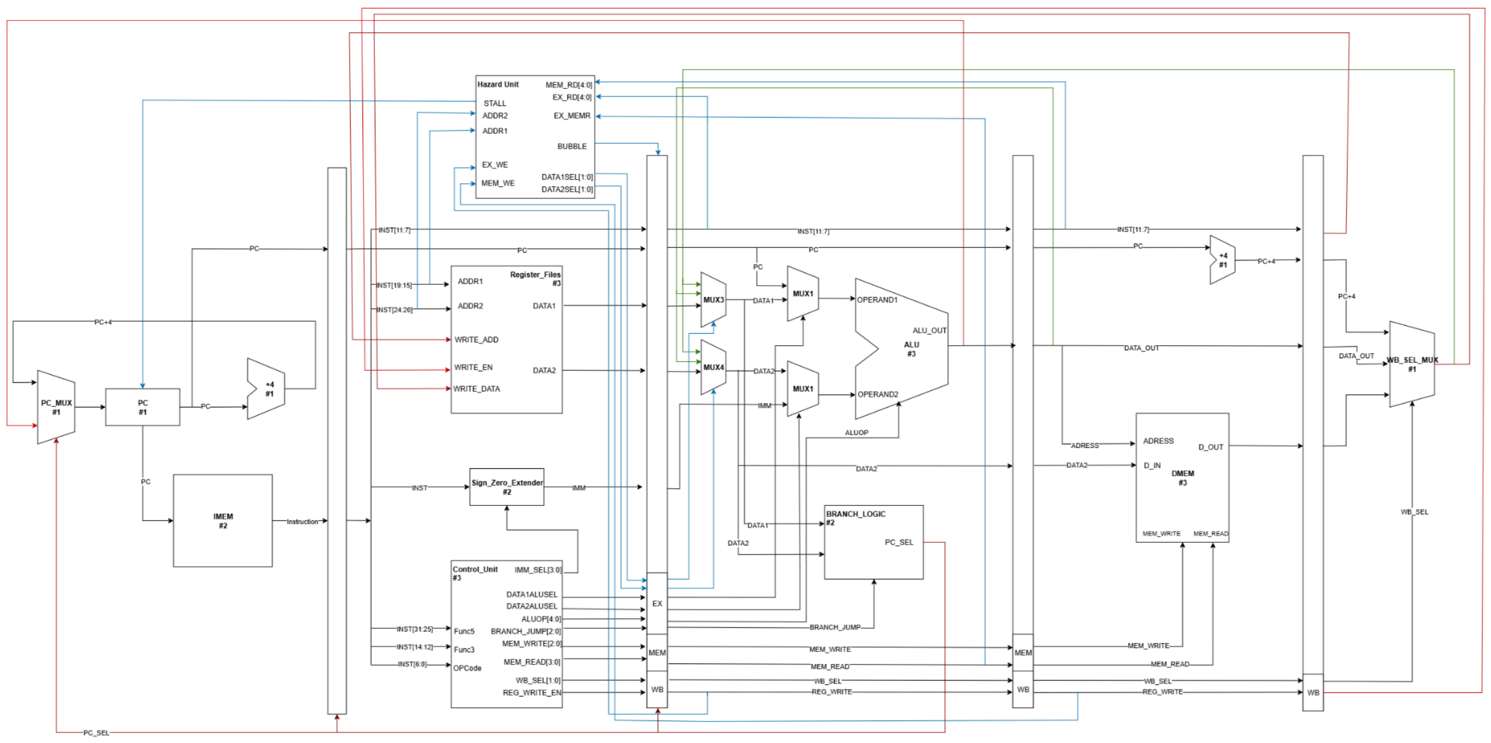
Using multiplexers (MUXes), the forwarding paths are connected to the ALU inputs, allowing the processor to select between:

- The original operand from the register file,
- A forwarded value from the EX/MEM pipeline register, or
- A forwarded value from the MEM/WB pipeline register.

This flexible operand selection mechanism ensures that the ALU always receives the most recent data, thereby preventing RAW hazards while maintaining high instruction throughput. The control logic also generates appropriate stall and flush signals to hold or clear pipeline registers when necessary, ensuring correct program execution in the presence of data and control dependencies.

These enhancements collectively enable smooth and efficient pipelined execution even in the presence of hazards, significantly improving performance over the baseline processor without hazard handling.

Updated Block Diagram



Updated Overview Diagram

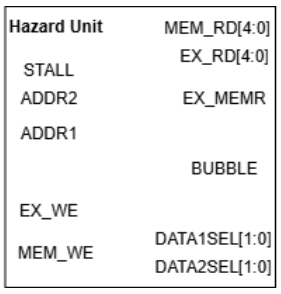
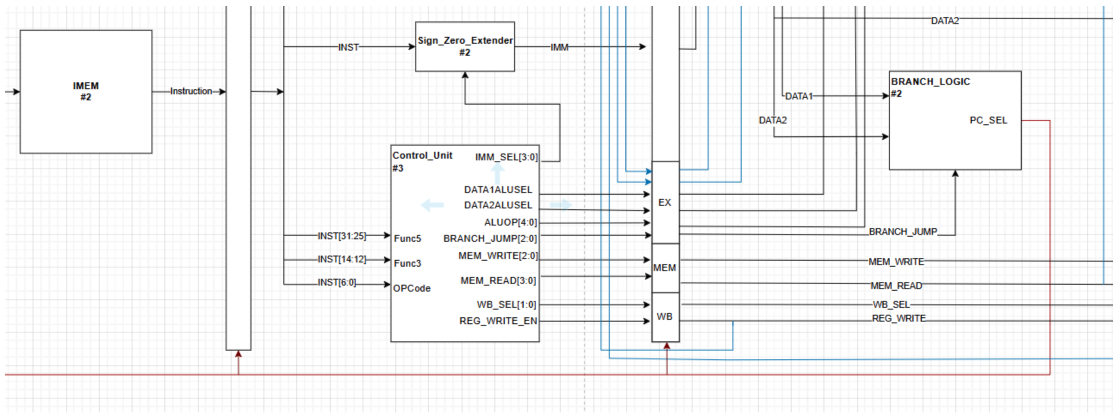


Diagram of Hazard Unit and Signals



Branch unit with Flushing

Control Unit and Signal Summary

To support automatic hazard resolution in the pipeline, several control signals were added and integrated into the datapath and control unit logic:

- STALL

  This signal is asserted when a load-use hazard is detected. It prevents updates to the Program Counter (PC) and the IF/ID pipeline register, effectively stalling the fetch and decode stages for one cycle. This allows time for the load instruction in the EX stage to complete, ensuring that the following instruction receives correct data.

- BUBBLE

  This signal controls the insertion of a NOP (No Operation) in the EX stage. When asserted, the instruction in the decode stage is prevented from advancing into EX, and a neutral operation is inserted instead. This is typically done in response to a load-use hazard to prevent incorrect execution.

- FLUSH (PC_SEL)

  This signal is asserted when a branch or jump instruction is taken. It clears the IF/ID and ID/EX pipeline registers to prevent incorrect instructions from being executed after a control flow change. Flushing avoids the side effects of instructions that were fetched before the actual branch target was known. Here, the same signal is used to select the next PC.

- DATA1SEL, DATA2SEL

These 2-bit forwarding select signals determine the data source for each ALU operand:

  - 00: Use the register file output (default, no hazard).
  - 01: Forward data from the EX/MEM stage.
  - 10: Forward data from the MEM/WB stage.

These signals are generated by the hazard unit and used by MUXes in the execute stage to route the correct operands to the ALU, resolving RAW hazards without stalling the pipeline.

Timing Model

The processor follows a classic 5-stage pipeline, where each instruction ideally takes one clock cycle per stage:

- IF (Instruction Fetch)
- ID (Instruction Decode)
- EX (Execute)
- MEM (Memory Access)
- WB (Write Back)

Hazard detection logic dynamically inserts stall cycles when required, particularly in cases of load-use data hazards, which causes a 1-cycle delay to allow data to become available.

Branch and jump instructions introduce a 1-cycle penalty due to the decision being made in the EX stage. If a branch is taken, the instructions in the IF and ID stages are flushed to avoid incorrect execution.

Test Programs and Validation

Test Programs

To comprehensively evaluate the functionality of our pipelined RISC-V processor with hazard handling, we developed and simulated a representative program containing arithmetic dependencies, load-use hazards, and control hazards. The following assembly code was used for simulation:

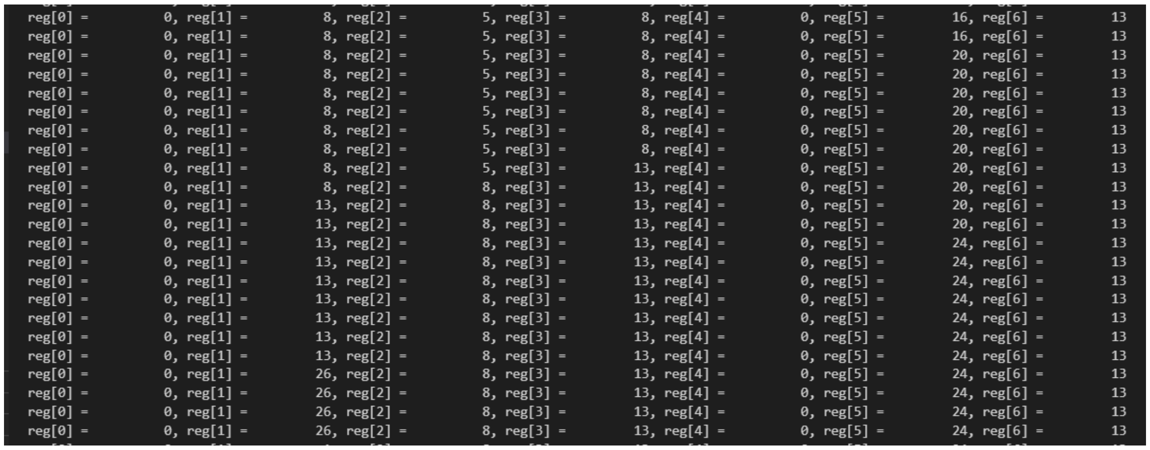
Test Code:

```
_boot:
   addi x1, x0, 1       // x1 = 1
   addi x2, x0, 0       // x2 = 0
   addi x5, x0, 0       // x5 = 0 (store address base)
   addi x7, x0, 6       // x7 = loop counter = 6

Loop:
   add x3, x1, x2       // x3 = x1 + x2  → RAW hazard with x1, x2
   addi x2, x1, 0       // x2 = x1       → Forward x1
   addi x1, x3, 0       // x1 = x3       → Forward x3
   sw x3, 0(x5)         // store x3      → Memory write
   addi x5, x5, 4       // increment address
   addi x7, x7, -1      // decrement counter
   beq x7, x0, Done // branch if x7 == 0 → Control hazard
   jal x0, Loop         // jump back to Loop

Done:
   lw x6, -4(x5)        // load last value  → Load-use hazard
   add x1, x6, x6       // x1 = x6 + x6    → Forward from MEM/WB
   jal x0, _boot        // restart program
```

Final Output of the registers:

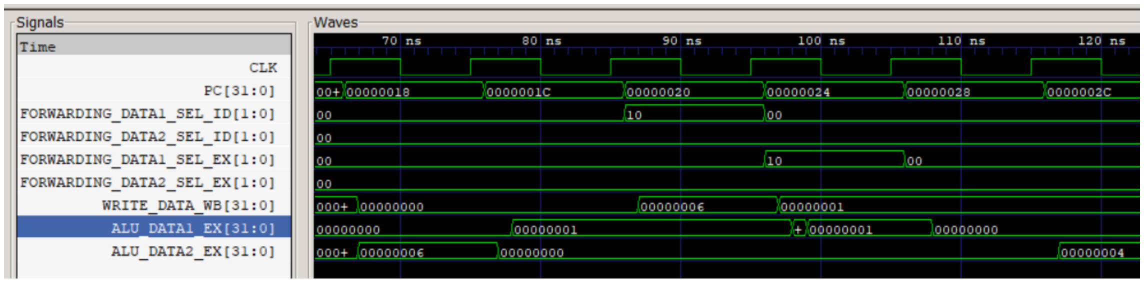


```
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         16, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         16, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =          8, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          5, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =          8, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         20, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         13, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
reg[0] =          0, reg[1] =         26, reg[2] =          8, reg[3] =         13, reg[4] =          0, reg[5] =         24, reg[6] =         13
```

The above register dump shows the correct progression and results of the sample program execution. The Fibonacci-like loop successfully computes intermediate values using arithmetic and memory instructions, and stores the final results in memory and registers. Notably, register x1 (used as an accumulator) evolves from 5 to 8 to 13 to 26, confirming that dependencies between instructions are correctly resolved. Additionally, the consistent value of x6 = 13 across cycles after the lw instruction confirms that the load-use hazard was detected and correctly stalled to ensure the loaded value was ready before use. The value of x5 increases by 4 each iteration as expected (acting as a memory pointer), and x3 temporarily holds the sum in each loop cycle. The control flow is preserved correctly, and all instructions execute without corruption or bypass errors, validating that the hazard unit, forwarding logic, and pipeline control signals work as intended. This confirms that the processor produces functionally correct results without needing manually inserted NOPs.

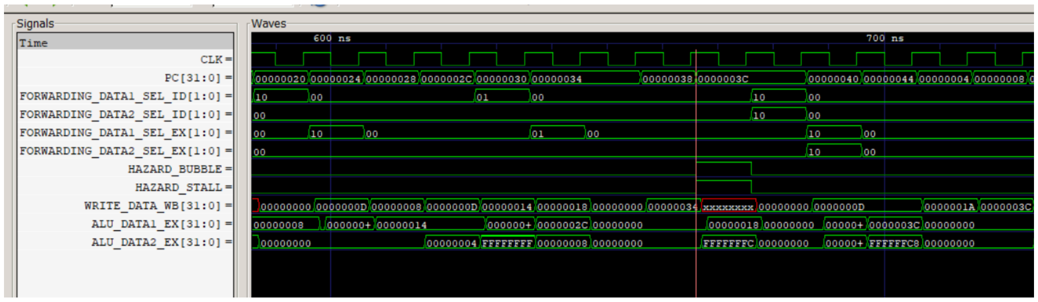
Test Cases Covered

- Arithmetic Dependency Test
    - Instructions: add x3, x1, x2, followed by addi x2, x1, 0, addi x1, x3, 0
    - Hazard Type: RAW hazards
    - Validation: Forwarding from EX and MEM stages was triggered appropriately. No stalls were needed, and correct ALU inputs were selected through forwarding.



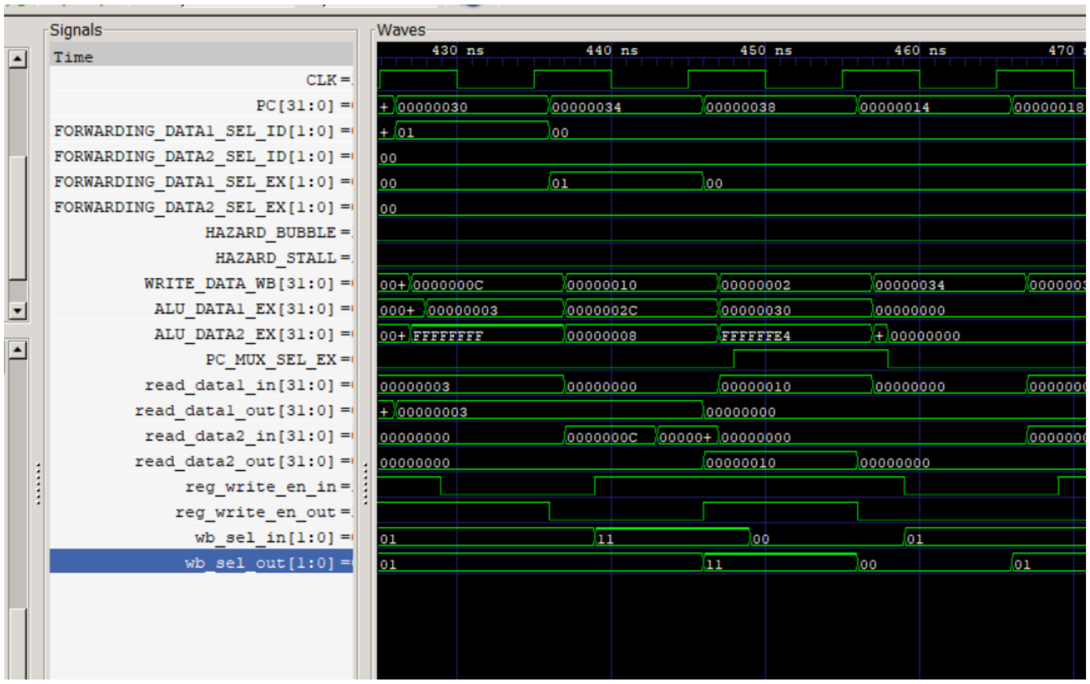
Waveform of Arithmetic Dependency Test

- Load-Use Hazard Test
  - Instructions: lw x6, -4(x5) followed by add x1, x6, x6
  - Hazard Type: Load-use data hazard
  - Validation: One stall cycle was inserted by the hazard detection unit. A bubble was observed in the EX stage. Forwarding was disabled during this cycle to allow correct data to be written back.



Waveform of Load Use Hazard

- Control Hazard Test
  - Instructions: beq x7, x0, Done, jal x0, Loop
  - Hazard Type: Control hazard
  - Validation: Upon branch resolution in the EX stage, the instructions in IF and ID were flushed. This avoided the execution of incorrectly fetched instructions.



Waveform of Control Hazard Test

Validation Summary

Correct program behavior was observed for each case, with outputs and memory values matching expectations.

Clock cycle analysis confirmed pipeline efficiency: unnecessary stalls were avoided thanks to forwarding logic.

Waveform inspection using GTKWave showed:

- Assertion of control signals (stall, bubble, flush)
- Correct operation of forwarding signals (forwarding_data1sel, forwarding_data2sel)
- Precise instruction timing and hazard resolution

Design Decisions and Limitations

Design Choices

To balance simplicity and correctness, several key design choices were made during the implementation of hazard handling in our pipelined RISC-V processor:

- In-order execution
    - All instructions are executed strictly in program order, simplifying pipeline control and avoiding the complexities of out-of-order execution models.
- Explicit hazard handling using dedicated units
    - A hazard unit was implemented to detect and resolve data and control hazards. This modular design avoids the complexity of techniques like scoreboarding or Tomasulo's algorithm, which are typically used in more advanced processors.
- No branch prediction
    - Branch instructions are resolved deterministically in the EX stage, and a static flush mechanism is used for mispredicted instructions. This significantly reduces control complexity and hardware overhead.
- Modular forwarding logic
    - Forwarding decisions are made based on clearly defined rules, allowing data to be forwarded from EX/MEM and MEM/WB stages using simple MUX logic controlled by the hazard unit.

Limitations

While the design achieves functional correctness and pipeline efficiency, it has several limitations:

- No branch prediction
    - All control hazards incur a fixed 1-cycle penalty, as branch decisions are only known in the EX stage. This limits performance in branch-heavy code segments.

- No support for WAW or WAR hazards
    - Since the processor executes instructions in order, Write-After-Write (WAW) and Write-After-Read (WAR) hazards are not handled. These hazards only occur in out-of-order designs and are not applicable here.
- No support for multi-cycle operations
    - All instructions are assumed to execute in a single cycle per stage. Operations like multiplication or division that may require multiple cycles are not supported natively and would need additional control and stalling logic.
- Static forwarding and stall logic
    - The hazard detection and forwarding logic are hardwired for a specific pattern of hazard scenarios. The processor cannot dynamically adapt to future optimization techniques like register renaming or speculative execution.
- Limited scalability
    - The current hazard logic is tailored for a 5-stage, single-issue pipeline. Scaling this architecture to support superscalar execution or deeper pipelines would require significant redesign.