Design and Implementation of a 5-Stage Pipelined RISC-V RV32I Processor

Gallo Andrea
Student ID: 2359271
andrea.gallo13@studio.unibo.it

July 27, 2025

Final Project for Digital Electronics

Abstract

This report details the complete design, implementation, and verification of a 5-stage pipelined processor compliant with the RISC-V RV32I base integer instruction set architecture. The processor employs the classic five-stage RISC pipeline (Instruction Fetch, Decode, Execute, Memory Access, and Write-Back) and was described entirely in SystemVerilog. The core of the project focuses on ensuring correct execution in the presence of pipeline hazards. To achieve this, a comprehensive hazard management unit was designed and implemented, capable of resolving data hazards through both forwarding and stalling, and managing control hazards by flushing the pipeline after taken branches and jumps.

The functional correctness of the design was rigorously verified using a custom-written assembly test program, specifically crafted to trigger all critical hazard scenarios. This program was executed in a dedicated testbench, and its behavior was analyzed in detail through waveform inspection with GTKWave. The final implementation successfully demonstrates all the required functionalities, providing a practical and in-depth exploration of the core principles and challenges inherent in modern processor design.

Contents

1 Introduction

1.1 Background

Central Processing Units (CPUs) represent the computational core of virtually every modern electronic system, from large-scale servers to embedded microcontrollers. The Instruction Set Architecture (ISA) of a CPU serves as the fundamental interface between hardware and software, defining the set of operations the processor can execute. Historically, ISAs have been categorized into two primary design philosophies: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC).

CISC architectures, such as the ubiquitous x86 family, aim to minimize the number of instructions required for a program by incorporating complex, multi-step operations into single instructions. In contrast, the RISC philosophy, which gained prominence in the 1980s, advocates for a smaller, simpler, and highly-optimized set of instructions. Each instruction is designed to execute in a single clock cycle in an ideal pipelined implementation, as detailed in foundational texts on digital design [1]. This simplicity generally leads to designs that are more regular, consume less power, and can achieve higher clock frequencies.

In this landscape, the RISC-V ISA has emerged as a modern, transformative force. Unlike most other commercial ISAs such as ARM or x86, RISC-V is a free, open, and extensible standard. This open-source nature eliminates licensing barriers and fosters a global ecosystem of innovation in both academia and industry. Its modular design allows for the implementation of a minimal base integer ISA—like the RV32I targeted in this project—which can be extended with standard optional features. This flexibility makes RISC-V an exceptional platform for teaching computer architecture principles and for developing highly specialized, custom processors.

1.2 Design Goals

The primary objective of this project is to design, implement, and verify a 5-stage pipelined processor capable of executing the complete base integer instruction set of the RISC-V ISA. The design focuses on the practical challenges of pipelined architectures and the mechanisms required to ensure correct execution. To this end, the following core functionalities and design goals were established:

- **RV32I Base Instruction Set Compliance:** To correctly implement the behavior of all instructions in the base 32-bit integer set, as defined in the official manual [2]. The structure and encoding of the implemented instruction formats are detailed in Table 1.
- **Classic 5-Stage Pipeline Implementation:** To structure the processor datapath according to the classic five-stage pipeline model: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB). This architecture is fundamental to modern processor design [3].
- **Data Hazard Resolution:** To implement a comprehensive hazard unit that resolves data dependencies (Read-After-Write hazards) using a combination of:
  - *Forwarding:* Logic to forward results from the EX and MEM stages directly to the EX stage, preventing unnecessary stalls.
  - *Stalling:* Logic to handle the specific case of a load-use hazard, where a one-cycle stall is unavoidable.

- **Control Hazard Management:** To manage control flow changes from branch and jump instructions by flushing incorrect, speculatively-fetched instructions from the pipeline when a branch is taken.
- **Systematic Verification:** To prove the functional correctness of the design through extensive simulation, utilizing a custom-written test program to trigger and validate all core functionalities.

Table 1: Implemented RV32I Instruction Formats and Encodings

Format	Mnemonic	Fields [31:25]	rs2	rs1	Fields [14:12]	rd	Fields [11:7]	Opcode	Description
Register-to-Register Operations (R-Type)									
R	ADD	0000000	rs2	rs1	000	rd	–	0110011	Add
R	SUB	0100000	rs2	rs1	000	rd	–	0110011	Subtract
R	SLL	0000000	rs2	rs1	001	rd	–	0110011	Shift Left Logical
R	SLT	0000000	rs2	rs1	010	rd	–	0110011	Set Less Than
R	SLTU	0000000	rs2	rs1	011	rd	–	0110011	Set Less Than, Unsigned
R	XOR	0000000	rs2	rs1	100	rd	–	0110011	Bitwise XOR
R	SRL	0000000	rs2	rs1	101	rd	–	0110011	Shift Right Logical
R	SRA	0100000	rs2	rs1	101	rd	–	0110011	Shift Right Arithmetic
R	OR	0000000	rs2	rs1	110	rd	–	0110011	Bitwise OR
R	AND	0000000	rs2	rs1	111	rd	–	0110011	Bitwise AND
Immediate Operations (I-Type)									
I	ADDI	imm[11:0]		rs1	000	rd	–	0010011	Add Immediate
I	SLTI	imm[11:0]		rs1	010	rd	–	0010011	Set Less Than Immediate
I	SLTIU	imm[11:0]		rs1	011	rd	–	0010011	Set Less Than Imm., Unsigned
I	XORI	imm[11:0]		rs1	100	rd	–	0010011	XOR Immediate
I	ORI	imm[11:0]		rs1	110	rd	–	0010011	OR Immediate
I	ANDI	imm[11:0]		rs1	111	rd	–	0010011	AND Immediate
I	SLLI	0000000	shamt	rs1	001	rd	–	0010011	Shift Left Logical Imm.
I	SRLI	0000000	shamt	rs1	101	rd	–	0010011	Shift Right Logical Imm.
I	SRAI	0100000	shamt	rs1	101	rd	–	0010011	Shift Right Arithmetic Imm.
Load Operations (I-Type)									
I	LB	imm[11:0]		rs1	000	rd	–	0000011	Load Byte
I	LH	imm[11:0]		rs1	001	rd	–	0000011	Load Halfword
I	LW	imm[11:0]		rs1	010	rd	–	0000011	Load Word
I	LBU	imm[11:0]		rs1	100	rd	–	0000011	Load Byte, Unsigned
I	LHU	imm[11:0]		rs1	101	rd	–	0000011	Load Halfword, Unsigned
Store Operations (S-Type)									
S	SB	imm[11:5]	rs2	rs1	000	–	imm[4:0]	0100011	Store Byte
S	SH	imm[11:5]	rs2	rs1	001	–	imm[4:0]	0100011	Store Halfword
S	SW	imm[11:5]	rs2	rs1	010	–	imm[4:0]	0100011	Store Word
Branch Operations (B-Type)									
B	BEQ	imm[12,10:5]	rs2	rs1	000	–	imm[4:1,11]	1100011	Branch if Equal
B	BNE	imm[12,10:5]	rs2	rs1	001	–	imm[4:1,11]	1100011	Branch if Not Equal
B	BLT	imm[12,10:5]	rs2	rs1	100	–	imm[4:1,11]	1100011	Branch if Less Than
B	BGE	imm[12,10:5]	rs2	rs1	101	–	imm[4:1,11]	1100011	Branch if Greater or Equal
B	BLTU	imm[12,10:5]	rs2	rs1	110	–	imm[4:1,11]	1100011	Branch if Less, Unsigned
B	BGEU	imm[12,10:5]	rs2	rs1	111	–	imm[4:1,11]	1100011	Branch if Greater, Unsigned
Upper Immediate & Jump Operations (U/J-Type)									
U	LUI	imm[31:12]				rd	–	0110111	Load Upper Immediate
U	AUIPC	imm[31:12]				rd	–	0010111	Add Upper Immediate to PC
J	JAL	imm[20,10:1,11,19:12]				rd	–	1101111	Jump and Link
I	JALR	imm[11:0]		rs1	000	rd	–	1100111	Jump and Link Register

1.3 Technology Choice

The successful realization of this project was contingent upon specific architectural and technological decisions. These choices were made to align with the project's learning objectives and to employ tools that are both powerful and suitable for complex digital design verification.

- **Pipelined Architecture:** A 5-stage pipelined architecture was deliberately chosen over simpler designs like single-cycle or multi-cycle processors. Although a pipelined implementation introduces the complexity of managing hazards, it offers a substantial increase in instruction throughput. By overlapping the execution phases of multiple instructions, the processor can theoretically approach an ideal performance of one Cycle Per Instruction (CPI), a core principle in modern high-performance CPU design [3].
- **SystemVerilog for Hardware Description:** The processor was described using SystemVerilog. This language was chosen as it is a modern superset of Verilog, offering enhanced features that streamline the design process for complex systems. Its more concise syntax and powerful constructs, such as improved data types and interfaces, allow for more readable and maintainable code, which proved invaluable given the intricate nature of a pipelined processor with hazard-detection logic. Familiarity with Verilog served as a foundation, with SystemVerilog being the natural progression for a project of this scale.
- **Open-Source Verification Toolchain:** For functional verification, a fully open-source toolchain was employed. The logic was simulated using **Icarus Verilog**, a robust and standards-compliant compiler for Verilog and SystemVerilog. The resulting simulation output, captured in Value Change Dump (.vcd) files, was then visually analyzed using **GTKWave**, a powerful and versatile waveform viewer. This combination of tools provides a cost-free yet highly effective environment for rigorous debugging and verification of digital hardware designs.

2 Processor Architecture and Design

The processor's design is centered around the classic five-stage RISC pipeline, an architectural paradigm that is fundamental to achieving high instruction throughput. This section details the overall datapath architecture and the implementation of its core components.

2.1 Datapath Architecture

The core of the processor is a five-stage pipelined datapath, illustrated in Figure 1. This structure allows up to five instructions to be in different phases of execution simultaneously, maximizing hardware utilization. The stages are decoupled by pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB), which latch all necessary data and control signals at the end of each clock cycle to be used by the subsequent stage. The functionality of each stage is described below.

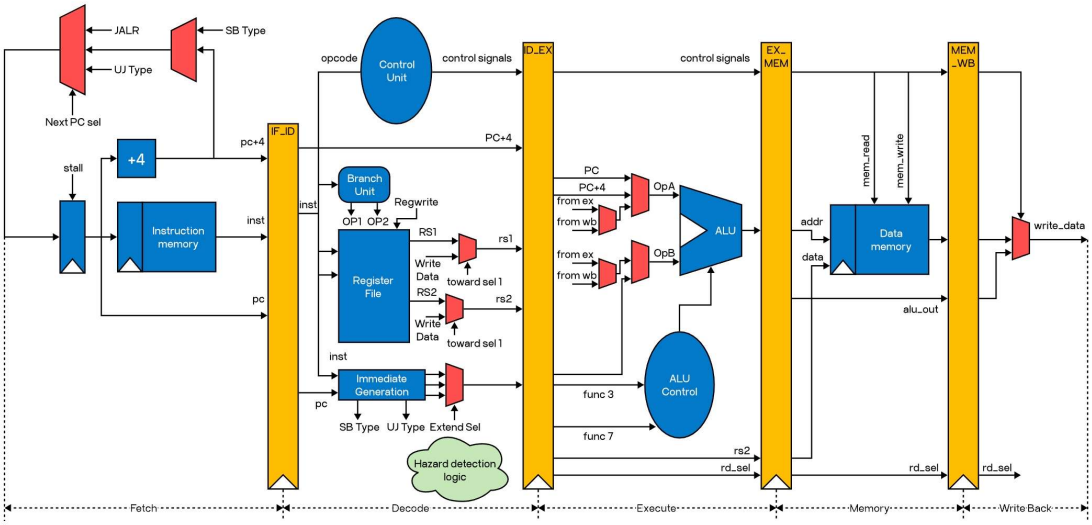

Figure 1: Top-Level Block Diagram of the 5-Stage RISC-V Datapath.

- **Instruction Fetch (IF):** The process begins here. The Program Counter (PC) holds the address of the current instruction, which is fetched from the Instruction Memory. In parallel, an adder computes the address of the default next instruction (PC+4). The PC itself is updated at the end of the cycle, with its next value being selected based on whether the processor continues sequentially, handles a stall, or takes a branch or jump.

- **Instruction Decode (ID):** In this stage, the fetched instruction is parsed to determine the operation to be performed. A main **Control Unit** decodes the instruction's opcode to generate the constellation of control signals that will steer the datapath. The operand values are read from the **Register File** based on the 'rs1' and 'rs2' fields. Concurrently, an **Immediate Generation** block sign-extends the immediate value according to the instruction's format (I, S, B, U, J). This stage is also the critical point for control: the **Hazard Detection Logic** analyzes register dependencies and control flow instructions to assert stall or flush signals when necessary.

- **Execute (EX):** This stage performs the primary computation. The **Arithmetic Logic Unit (ALU)** takes two operands and executes the specified function. Forwarding multiplexers at the ALU inputs are crucial here, as they select between operands coming from the ID stage or results being forwarded from the EX, MEM, or WB stages to mitigate data hazards. A secondary **ALU Control** unit determines the specific ALU operation based on the instruction's 'funct3' and 'funct7' fields.

- **Memory Access (MEM):** This stage is responsible for all interactions with the main data memory. The address for the operation is supplied by the result of the ALU from the previous stage. For load instructions ('lw', 'lb', etc.), data is read from memory. For store instructions ('sw', 'sb', etc.), data is written to memory. For all non-memory instructions, the ALU result is simply passed through to the next pipeline register.

- **Write-Back (WB):** In the final stage, the result is committed. A multiplexer selects the data to be written into the **Register File**: either the computed result

from the ALU or the data loaded from memory. The instruction's ‘rd‘ field specifies the destination register.

2.2 Core Component Implementation

2.2.1 Control Unit

The control unit is the central nervous system of the processor. It is a purely combinational logic block responsible for interpreting the instruction currently in the Decode stage and generating a set of control signals that orchestrate the datapath's actions. The primary input to the control unit is the 7-bit opcode field of the instruction.

To manage complexity and maintain a modular design, the control logic is implemented using a two-level decoding strategy, a common practice in processor design:

- **Main Decoder:** This primary unit decodes the instruction's opcode to produce the main control signals that are independent of the specific ALU operation. These include signals to enable memory access, to control the source of the ALU's second operand (register or immediate, ‘ALUSrc‘), to enable the final write-back to the register file (‘RegWrite‘), and to specify the source of the data for this write-back (‘ResultSrc‘).
- **ALU Decoder:** This secondary unit generates the specific 4-bit control word for the ALU itself. Its logic depends not only on a 2-bit ‘ALUOp‘ signal from the main decoder but also on the instruction's function fields (‘funct3‘ and ‘funct7‘). This hierarchical approach simplifies the main decoder, which only needs to differentiate between broad instruction categories, while the specialized ALU decoder handles the specifics of operations like ‘ADD‘, ‘SUB‘, ‘XOR‘, etc.

The core of the main decoder is implemented using a ‘case‘ statement based on the opcode. This allows for a clear and readable mapping from each instruction type to its corresponding control signal vector. A SystemVerilog code snippet illustrating this logic for several key instruction categories is shown in Listing 1. The code demonstrates how control signals are asserted differently for R-type, load, store, and branch instructions.

```
1  // Main Combinational Decoding Logic
2  // Default values are set first, then overridden by the case statement.
3  always_comb begin
4      // Set default values for all control signals (e.g., for NOP)
5      o_RegWrite  = 1'b0;
6      o_MemWrite  = 1'b0;
7      o_Branch    = 1'b0;
8      // ... other defaults ...
9
10     case (i_opcode)
11         // Load Word (lw, etc.)
12         OP_LOAD: begin
13             o_ALUSrc    = 1'b1;          // ALU operand is immediate
14             o_RegWrite  = 1'b1;          // Write result back to register
       file
15             o_ResultSrc = RES_SRC_MEM;  // Result comes from data memory
16             o_ImmSrc    = IMM_I;        // Use I-type immediate
17             o_ALUOp     = ALUOP_LD_ST;  // ALU performs ADD for address
       calculation
18         end
19
20         // Store Word (sw, etc.)
```

```
21          OP_STORE: begin
22              o_ALUSrc    = 1'b1;          // ALU operand is immediate
23              o_MemWrite  = 1'b1;          // Enable memory write
24              o_ImmSrc    = IMM_S;         // Use S-type immediate
25              o_ALUOp     = ALUOP_LD_ST;   // ALU performs ADD for address
    calculation
26          end
27
28          // Register-Register operations (add, sub, etc.)
29          OP_R_TYPE: begin
30              o_RegWrite  = 1'b1;          // Write result back to register
    file
31              // ALUSrc is 0 (default), ResultSrc is ALU (default)
32              o_ALUOp     = ALUOP_R_I;     // Defer to ALU Decoder
33          end
34
35          // Conditional Branch (beq, bne, etc.)
36          OP_BRANCH: begin
37              o_Branch    = 1'b1;          // This is a branch instruction
38              // ALUSrc is 0 (default) for Reg-Reg comparison
39              o_ImmSrc    = IMM_B;         // Use B-type immediate for target
    address
40              o_ALUOp     = ALUOP_BRANCH; // ALU performs SUB for comparison
41          end
42
43          // ... cases for other opcodes like I-Type, JAL, etc.
44
45          default: begin
46              // For any invalid opcodes, assert safe, inactive values.
47          end
48      endcase
49  end
```

Listing 1: SystemVerilog snippet of the main decoder logic from ‘decoder.sv‘.

2.2.2 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is the computational core of the processor, responsible for performing all arithmetic and logical operations. It is a purely combinational block located in the Execute (EX) stage of the pipeline. The ALU takes two 32-bit operands as input and, based on a 4-bit control signal from the ALU Decoder, produces a 32-bit result along with several status flags.

The implemented ALU supports a comprehensive set of operations required by the RV32I standard, including:

- **Arithmetic Operations:** Addition, Subtraction.
- **Logical Operations:** AND, OR, XOR.
- **Shift Operations:** Shift Left Logical (SLL), Shift Right Logical (SRL), and Shift Right Arithmetic (SRA).
- **Comparison Operations:** Set on Less Than (SLT) for both signed and unsigned operands.

A notable implementation detail is the use of a unified adder/subtractor core. Subtraction is achieved using the two’s complement method, where `A - B` is computed as

`A + (B̃) + 1`. This is efficiently implemented by using one bit of the ALU control signal to conditionally invert the second operand and to serve as the carry-in for the addition.

In addition to the 32-bit result, the ALU generates four status flags that are critical for evaluating conditional branches: a **zero flag** (asserted if the result is zero), a **sign flag** (the most significant bit of the result), a **signed overflow flag**, and an **unsigned carry flag**.

The central logic of the ALU is built around a 'case' statement that selects the appropriate operation based on the 4-bit control input, as shown in the SystemVerilog snippet in Listing 2.

```
// -- Main ALU Result Multiplexer --
// Selects the final result based on the ALU control signal.
always_comb begin
    case (i_alu_control)
        ALU_ADD,
        ALU_SUB:  o_result = sum;                    // Result from shared adder/subtractor
        ALU_AND:  o_result = i_operand_a & i_operand_b; // Bitwise AND
        ALU_OR:   o_result = i_operand_a | i_operand_b; // Bitwise OR
        ALU_SLL:  o_result = i_operand_a << i_operand_b[4:0]; // Shift Left
        ALU_SRA:  o_result = $signed(i_operand_a) >>> i_operand_b[4:0]; // Shift Right Arithmetic

        // Set on Less Than (signed)
        ALU_SLT:  o_result = ($signed(i_operand_a) < $signed(i_operand_b)) ? 32'd1 : 32'd0;

        // Set on Less Than (unsigned)
        ALU_SLTU: o_result = (i_operand_a < i_operand_b) ? 32'd1 : 32'd0;

        default: o_result = 32'bx; // Default to 'x'
    endcase
end

// -- Status Flag Generation --
// Zero flag is asserted if the result of the main operation is exactly zero.
assign o_zero_flag = (o_result == 32'b0);

// Signed overflow for ADD/SUB operations.
assign o_overflow_flag = (i_operand_a[31] == b_operand[31]) &&
                         (i_operand_a[31] != sum[31]) &&
                         ((i_alu_control == ALU_ADD) || (i_alu_control == ALU_SUB));
```

Listing 2: SystemVerilog snippet of the core ALU logic from 'alu.sv'.

2.2.3 Register File

The register file is the core state-holding element of the processor, containing the 32 general-purpose registers (`x0`–`x31`) as specified by the RV32I ISA. Each register is 32 bits wide. This module is critical for the data flow, as it must supply operands to the Execute stage and commit results from the Write-Back stage. The design follows a standard and highly-efficient architecture featuring two asynchronous read ports and one synchronous write port.

The key implementation details are:

- **Two Asynchronous Read Ports:** The register file provides two independent, combinational read ports. The data on the output ports (‘o_data_rd1‘, ‘o_data_rd2‘) changes immediately in response to a change on the address inputs (‘i_addr_rd1‘, ‘i_addr_rd2‘). This design is essential for performance in the Instruction Decode (ID) stage, as it allows the two source operands (‘rs1‘ and ‘rs2‘) to be fetched simultaneously and without clock delay as soon as the instruction is decoded.

- **One Synchronous Write Port:** In contrast to reading, the write operation is synchronous and occurs only on the rising edge of the clock. A write is performed if the ‘i_reg_write‘ control signal (originating from the control unit and passed down the pipeline to the Write-Back stage) is asserted. This approach is fundamental for a stable pipeline, ensuring that the processor's state is updated predictably only at discrete clock-cycle boundaries, thus preventing race conditions.

- **Hardwired Zero Register (‘x0‘):** As per the RISC-V specification, register `x0` is immutable and must always read as zero. This is enforced with dedicated logic, as shown in the code snippet in Listing 3. Any attempt to write to address ‘5’b0‘ is ignored by the write-port logic. Similarly, if either read port is addressed to ‘5’b0‘, the module bypasses the memory array and outputs a constant ‘32’b0‘. This provides a convenient, hardwired source for the value zero.

The interaction with the pipeline is precisely timed: the read addresses are supplied by the instruction residing in the ID stage, while the write address and data are determined by the instruction completing its execution in the WB stage. This temporal separation of read and write operations is what intrinsically prevents structural hazards on the register file itself.

```
// --- Synchronous Write Port ---
// Writes occur on the rising edge of the clock if write enable is high.
// The check for i_addr_wr != 0 enforces the hardwired-zero nature of x0.
always_ff @(posedge i_clk) begin
    if (i_reg_write && (i_addr_wr != 5'b0)) begin
        rf_data[i_addr_wr] <= i_data_wr;
    end
end

// --- Asynchronous Read Ports ---
// Reading register x0 (address 0) always returns 0. Otherwise, the data
// is read combinationally from the register array.
assign o_data_rd1 = (i_addr_rd1 == 5'b0) ? 32'b0 : rf_data[i_addr_rd1];
assign o_data_rd2 = (i_addr_rd2 == 5'b0) ? 32'b0 : rf_data[i_addr_rd2];
```

Listing 3: SystemVerilog snippet showing the core logic of the Register File.

3 Hazard Management and Verification

A pipelined architecture achieves high throughput by overlapping the execution of multiple instructions. However, this overlapping introduces potential conflicts known as *hazards*, which can lead to incorrect program execution if not managed properly. These hazards are primarily categorized into data hazards, control hazards, and structural hazards. This processor implements a dedicated combinational **Hazard Unit** responsible for detecting these conflicts and generating the necessary control signals to resolve them. The following sections detail the logic for managing data and control hazards through forwarding, stalling, and flushing.

3.1 Hazard Unit Implementation

The Hazard Unit is the brain of the pipeline's control system. It operates in the Instruction Decode (ID) stage, as this is the earliest point where information about the current instruction and the subsequent instructions in the pipeline is available. By comparing the source registers (‘rs1‘, ‘rs2‘) of the instruction in the ID stage with the destination registers (‘rd‘) of instructions in the EX, MEM, and WB stages, it can anticipate data dependencies. Similarly, by monitoring control flow instructions, it manages jumps and branches.

3.1.1 Forwarding Logic

Data hazards of the type Read-After-Write (RAW) are the most common issue in a pipeline. They occur when an instruction needs to read a register whose value has not yet been written back by a preceding, in-flight instruction. For example:

```
add  x1, x2, x3  // Instruction 1 (result written to x1 in WB)
sub  x4, x1, x5  // Instruction 2 (needs x1 immediately in EX)
```

Without intervention, the ‘sub‘ instruction would read an old, stale value of ‘x1‘ from the register file before the ‘add‘ instruction has had a chance to write the new value back. Waiting for the write-back to complete would require stalling the pipeline for several cycles, nullifying the benefits of pipelining.

The solution implemented is **forwarding** (also known as *bypassing*). Instead of waiting for the result to travel through the MEM and WB stages to be written into the register file, the Hazard Unit routes the result directly from the output of the ALU (at the end of the EX stage) or from the output of the MEM stage back to the inputs of the ALU. This ensures the most recent value is always available just in time for the next computation.

The Hazard Unit implements this logic by generating two 2-bit control signals, ‘ForwardAE‘ and ‘ForwardBE‘, which control multiplexers placed at the inputs of the ALU. The logic accounts for two primary forwarding paths, with a clear priority system:

1. **EX/MEM to EX Forwarding:** If an instruction in the EX stage needs a result from the instruction immediately preceding it (which is now in the MEM stage), the result is forwarded from the ‘ALUResult‘ register in the EX/MEM pipeline register. This path has the highest priority as it provides the most recently computed value.

2. **MEM/WB to EX Forwarding:** If the result is not available from the MEM stage, the Hazard Unit checks if the instruction two cycles prior (now in the WB stage) is writing back the required value. If so, the result is forwarded from the MEM/WB pipeline register. This path is only taken if the first path is not active for the same register, preventing a newer value from being overwritten by an older one.

The conditions check that the write-enable signal for the source instruction (‘RegWriteM‘ or ‘RegWriteW‘) is active and that the destination register is not ‘x0‘. The logic for generating these forwarding signals is shown in Listing 4.

```
1 // --- Forwarding Logic for Rs1 (Operand A) ---
2 // Priority 1: Forward from MEM stage to EX stage.
3 if (RegWriteM && (RdM != 5'b0) && (RdM == Rs1E)) begin
4     ForwardAE = 2'b01; // Select data from MEM stage
5 end
6 // Priority 2: Forward from WB stage to EX stage.
7 else if (RegWriteW && (RdW != 5'b0) && (RdW == Rs1E)) begin
8     ForwardAE = 2'b10; // Select data from WB stage
```

```
9  end
10
11 // --- Forwarding Logic for Rs2 (Operand B) ---
12 // Priority 1: Forward from MEM stage to EX stage.
13 if (RegWriteM && (RdM != 5'b0) && (RdM == Rs2E)) begin
14     ForwardBE = 2'b01; // Select data from MEM stage
15 end
16 // Priority 2: Forward from WB stage to EX stage.
17 else if (RegWriteW && (RdW != 5'b0) && (RdW == Rs2E)) begin
18     ForwardBE = 2'b10; // Select data from WB stage
19 end
```

Listing 4: Forwarding logic within the Hazard Unit.

3.1.2 Stall and Flush Logic

While forwarding resolves most data hazards, it cannot handle every situation. Furthermore, control flow changes from branches and jumps introduce control hazards that require a different resolution strategy. The Hazard Unit is responsible for detecting these specific cases and asserting signals to stall or flush parts of the pipeline, ensuring program correctness.

Load-Use Hazard and Stalling A specific and critical data hazard is the *load-use* case, which occurs when an instruction immediately uses the result of a 'lw' (load word) instruction. The data from a load is only available at the end of the MEM stage. However, an instruction immediately following it would need that data at the beginning of the EX stage. This one-cycle gap cannot be bridged by forwarding alone.

To handle this, the pipeline must be stalled for one cycle. The Hazard Unit detects this condition by checking if the instruction in the EX stage is a load ('ResultSrcE == 2'b01') and its destination register ('RdE') is one of the source registers ('Rs1D' or 'Rs2D') of the instruction currently in the ID stage. This exact scenario is triggered in the provided test program:

```
lw x10, 0(x5)      // lw is in EX stage
add x11, x10, x10  // add is in ID stage, needs x10
```

When this dependency is detected, the Hazard Unit asserts three signals:

- `StallF` and `StallD`: These signals freeze the PC and the IF/ID pipeline register. This effectively holds the 'add' instruction in the ID stage for an extra cycle, preventing it from moving forward with a stale value of 'x10'.
- `FlushE`: This signal injects a "bubble" (equivalent to a NOP) into the ID/EX pipeline register. This bubble takes the place of the stalled 'add' instruction in the pipeline, giving the 'lw' instruction one cycle to complete the MEM stage and make its data available for the forwarding logic.

After one cycle of stalling, the 'lw' instruction has advanced to the WB stage, and its result can be forwarded to the 'add' instruction, which is now allowed to proceed into the EX stage.

Control Hazards and Flushing Control hazards arise from branch and jump instructions that change the normal flow of execution (‘PC = PC + 4‘). The processor’s default behavior is to speculatively fetch the next sequential instruction, a strategy known as *predict not-taken*. The actual outcome of a branch (taken or not taken) and the target address are not known until the instruction reaches the EX stage. If the prediction turns out to be wrong (i.e., a branch is taken or an unconditional jump occurs), the instructions that were speculatively fetched are incorrect and must be squashed.

This squashing is achieved by *flushing* the IF and ID stages. When the Hazard Unit detects that a branch or jump is taken (indicated by the ‘PcSrc‘ signal from the EX stage not being ‘2’b00‘), it asserts ‘FlushD‘ and ‘FlushE‘. The provided test code deliberately triggers this:

```
    beq x18, x18, IS_TAKEN // This branch is always taken. It is in EX.
    addi x20, x0, 1        // FLUSHED. This instruction is in ID.
    addi x21, x0, 1        // FLUSHED. This instruction is in IF.
IS_TAKEN:
    addi x29, x0, 5        // Correct target instruction.
```

When the ‘beq‘ is resolved as "taken" in the EX stage, the Hazard Unit flushes the two subsequent ‘addi‘ instructions by converting them into NOPs, and the PC is updated with the correct branch target address (‘IS_TAKEN‘). The same mechanism applies to unconditional jumps like ‘jal‘. The logic implementation for both stalling and flushing is shown in Listing 5.

```
// --- Stall and Flush Logic ---

// Load-Use Hazard Detection:
// Stalls the pipeline for one cycle if a load result is needed immediately.
if (ResultSrcE == 2'b01 && (RdE != 5'b0) && ((RdE == Rs1D) || (RdE == Rs2D))) begin
    StallF = 1'b1; // Stop the Program Counter.
    StallD = 1'b1; // Keep the same instruction in the IF/ID register.
    FlushE = 1'b1; // Convert the instruction in EX to a NOP (bubble).
end else begin
    StallF = 1'b0;
    StallD = 1'b0;
    // A Control Hazard might still require flushing.
    if (PcSrc != 2'b00) begin
        FlushE = 1'b1;
    end else begin
        FlushE = 1'b0;
    end
end

// Control Hazard Detection:
// If a branch/jump is taken, flush the speculatively fetched instruction
// in the Decode stage.
if (PcSrc != 2'b00) begin
    FlushD = 1'b1; // Convert the instruction in ID to a NOP (bubble).
end else begin
    FlushD = 1'b0;
end
```

Listing 5: Stall and Flush logic from the Hazard Unit.

3.2 Simulation and Verification

Verifying the functional correctness of a pipelined processor is a complex task due to the high degree of parallelism and the intricate interactions between in-flight instructions. A robust verification strategy was therefore essential to ensure that the design goals were met and that the hazard management logic operated correctly under all specified conditions. The verification approach was multifaceted, combining a custom-written assembly test program with a purpose-built SystemVerilog testbench and detailed waveform analysis.

3.2.1 Testbench Environment

A dedicated testbench was created in SystemVerilog to provide a controlled environment for simulating the processor. The testbench, as shown in the snippet in Listing 6, is responsible for three primary tasks:

1. **Clock and Reset Generation:** The testbench instantiates the processor as a "Device Under Test" (DUT) and provides it with the necessary clock and reset signals. A 100 MHz clock signal is generated, and an active-high reset is asserted for the first two clock cycles to ensure the processor starts from a known, stable state with all pipeline registers cleared.

2. **Test Execution and Termination:** After de-asserting the reset, the testbench allows the simulation to run for a fixed duration of 200 clock cycles. This duration was chosen to be sufficient for the execution of the entire test program loaded into the DUT's instruction memory, including the final infinite loop which signals the end of the test sequence.

3. **White-Box Verification and Debugging Support:** At the end of the 200-cycle period, the testbench performs a final "white-box" verification. It uses its privileged position to directly access the internal register file storage array within the DUT (‘dut.rf_u.rf_data‘) and prints the final hexadecimal and signed decimal values of all 32 registers to the simulation console. This provides an immediate and comprehensive summary of the test outcome. Furthermore, the testbench enables deep debugging by generating a Value Change Dump (‘.vcd‘) file, which captures the state of every signal within the design on every clock cycle, enabling detailed offline analysis with a waveform viewer.

```
1  ‘timescale 1ns/1ps
2  module RISCV_pipeline_tb;
3      logic clk;
4      logic rst;
5      // ... DUT instantiation ...
6
7      // Clock and Reset Generation
8      localparam CLK_PERIOD = 10;
9      initial begin
10         clk = 0;
11         forever #(CLK_PERIOD/2) clk = ~clk;
12     end
13
14     initial begin
15         // VCD generation for waveform analysis
16         $dumpfile("gtkwave/waveout.vcd");
17         $dumpvars(0, dut);
```

```
18
19          // Assert reset for 2 cycles
20          rst = 1'b1;
21          #(CLK_PERIOD * 2);
22
23          // Run the simulation for a fixed duration
24          rst = 1'b0;
25          #(CLK_PERIOD * 200);
26
27          // Final verification: Print register file state
28          $display("\n--- Final Register State ---");
29          for (integer i = 0; i < 32; i = i + 1) begin
30              $display("x%02d: 0x%08h (%11d)",
31                       i, dut.rf_u.rf_data[i], $signed(dut.rf_u.rf_data[i]));
32          end
33          $display("-------------------------\n");
34          $finish;
35      end
36  endmodule
```

Listing 6: Key components of the SystemVerilog testbench.

3.2.2 Test Program Analysis

To systematically validate the processor's functionality, a custom assembly program was carefully crafted. This code, written in RISC-V assembly language, was then converted into its corresponding 32-bit hexadecimal machine code using a web-based assembler and simulator [4]. The resulting machine code was used to pre-load the processor's instruction memory, providing the stimulus for the simulation. The program itself is not designed to perform a complex calculation, but rather to trigger specific, well-defined scenarios within the pipeline that test the correctness of the hazard management logic. Each sequence of instructions is designed to create a particular type of dependency or control flow change, allowing for targeted verification.

The analysis below breaks down the key parts of the test program, explaining the intended pipeline behavior and the expected final register values.

Data Hazard Verification (Forwarding) The first part of the test focuses on data hazards, ensuring that the forwarding logic works for all intended paths.

- **EX/MEM to EX Forwarding:** An immediate dependency is created to test the highest-priority forwarding path.

```
# Machine Code  // Assembly Instruction
#-------------------------------------------------
001002b7          // lui x5, 0x100       (x5 = 0x100000)
00a00337          // lui x6, 0xa00       (x6 = 0xa00000)
00500393          // addi x7, x0, 5      (x7 = 5)
40730333          // sub x6, x6, x7
00630433          // add x8, x6, x6

```

  Pipeline Behavior: The 'sub' instruction calculates the new value for 'x6'. The very next instruction, 'add', needs this result as both of its operands. When the 'add' instruction is in the EX stage, the 'sub' instruction is in the MEM stage. The hazard unit must detect this dependency ('RdM == Rs1E' and 'RdM == Rs2E')

and activate both forwarding multiplexers (‘ForwardAE‘ and ‘ForwardBE‘) to route the ALU result from the end of the EX stage back to both ALU inputs for the next cycle. No stall should occur.

Expected Result: ‘x6‘ is updated to ‘0xa00000 - 5 = 0x9ffffb‘. The ‘add‘ instruction must receive this forwarded value to correctly compute ‘x8 = 0x9ffffb + 0x9ffffb = 0x13ffff6‘.

- **MEM/WB to EX Forwarding:** The second test verifies the forwarding path from a more distant instruction. An independent instruction (‘addi‘) is inserted to create a one-cycle gap.

```
00100493          // addi x9, x0, 1
4084d4b3          // sra x9, x9, x8

```

Pipeline Behavior: The ‘sra‘ instruction needs the value of ‘x8‘ (calculated by the ‘add‘ two cycles prior). By the time ‘sra‘ enters the EX stage, the ‘add‘ instruction is in the WB stage. The hazard unit must detect this and forward the result from the MEM/WB pipeline register.

Expected Result: The final value of ‘x9‘ is less important than confirming the forwarding mechanism works; however, the successful execution of all prior tests is required for this stage to be valid.

Load-Use Hazard Verification (Stalling) This is a critical test, as this hazard cannot be resolved by forwarding alone.

```
0002a503          // lw x10, 0(x5)
00a505b3          // add x11, x10, x10
```

Pipeline Behavior: The ‘add‘ instruction needs the value of ‘x10‘ at the beginning of its EX stage. However, the ‘lw‘ instruction only retrieves this value from data memory at the end of its MEM stage. This timing mismatch requires a stall. The Hazard Unit must detect the load-use dependency (‘ResultSrcE == load‘ and ‘RdE == Rs1D‘), freeze the PC and the IF/ID register for one cycle, and inject a bubble into the pipeline.

Expected Result: Assuming the data memory at address ‘0x100000‘ holds ‘0‘, ‘x10‘ should be updated to ‘0‘. If the stall is successful, ‘x11‘ will correctly be calculated as ‘0 + 0 = 0‘. If the stall fails, ‘x11‘ would be calculated using a stale, incorrect value of ‘x10‘. Therefore, the final value ‘x11=0‘ is proof of a functional stall mechanism.

Control Hazard Verification (Flushing) The final tests validate the handling of control flow changes.

```
00999463          // bne x19, x9, NO_TAKEN     (Branch Not Taken)
00a009b7          // lui x19, 0xa00            (This instruction MUST
    be executed)
01290663          // beq x18, x18, IS_TAKEN    (Branch TAKEN)
00100a13          // addi x20, x0, 1           (FLUSHED)
00100a93          // addi x21, x0, 1           (FLUSHED)
...
IS_TAKEN:
00500e93          // addi x29, x0, 5
...
00c000ef          // jal x1, END_LOOP          (JUMP)
00100b13          // addi x22, x0, 1           (FLUSHED)
```

Pipeline Behavior: The ‘bne‘ branch is predicted "not-taken", and since the condition is false, the pipeline proceeds normally. The ‘beq‘ branch, however, is always taken. The processor, having already fetched the next two ‘addi‘ instructions speculatively, must now squash them. The Hazard Unit asserts flush signals to turn the instructions for ‘x20‘ and ‘x21‘ into NOPs. A similar flush must occur for the ‘jal‘ instruction.

Expected Result: The key indicators of success are that the flushed instructions had no effect. Therefore, ‘x20‘ and ‘x21‘ must remain ‘0‘. The instruction at the branch target must execute, setting ‘x29‘ to ‘5‘. Finally, the ‘jal‘ instruction must execute, saving its return address in ‘x1‘ (in this specific implementation, the value is 72).

Final Verification The combined result of all these tests can be verified in a single view of the final register state. Figure 2 shows the output from the testbench at the end of the simulation. All the expected values discussed above are present: `x6=0x9ffffb`, `x8=0x13ffff6`, `x11=0`, `x20=0`, `x21=0`, `x29=5`, and `x1=72`. This comprehensive match between expected and actual results provides high confidence that all hazard and control mechanisms of the processor are implemented correctly.

```
op);. Defaulting to 1364-2005 behavior.
WARNING: Code/instruction_mem.sv:33: $readmemh(Code/instructions.txt): Not enough words in the file for the requested
range [0:127].
VCD info: dumpfile gtkwave/waveout.vcd opened for output.

----------------------------------------------------------------
x00 .------ hex: 0x00000000 .------ dec (signed):          0
x01 .------ hex: 0x00000048 .------ dec (signed):         72
x02 .------ hex: 0x00000000 .------ dec (signed):          0
x03 .------ hex: 0x00000000 .------ dec (signed):          0
x04 .------ hex: 0x00000000 .------ dec (signed):          0
x05 .------ hex: 0x00100000 .------ dec (signed):    1048576
x06 .------ hex: 0x009ffffb .------ dec (signed):   10485755
x07 .------ hex: 0x00000005 .------ dec (signed):          5
x08 .------ hex: 0x013ffff6 .------ dec (signed):   20971510
x09 .------ hex: 0x00000000 .------ dec (signed):          0
x10 .------ hex: 0x00000000 .------ dec (signed):          0
x11 .------ hex: 0x00000000 .------ dec (signed):          0
x12 .------ hex: 0x00000000 .------ dec (signed):          0
x13 .------ hex: 0x00000000 .------ dec (signed):          0
x14 .------ hex: 0x00000000 .------ dec (signed):          0
x15 .------ hex: 0x00000000 .------ dec (signed):          0
x16 .------ hex: 0x00000000 .------ dec (signed):          0
x17 .------ hex: 0x00000000 .------ dec (signed):          0
x18 .------ hex: 0x00a00000 .------ dec (signed):   10485760
x19 .------ hex: 0x00a00000 .------ dec (signed):   10485760
x20 .------ hex: 0x00000000 .------ dec (signed):          0
x21 .------ hex: 0x00000000 .------ dec (signed):          0
x22 .------ hex: 0x00000000 .------ dec (signed):          0
x23 .------ hex: 0x00000000 .------ dec (signed):          0
x24 .------ hex: 0x00000000 .------ dec (signed):          0
x25 .------ hex: 0x00000000 .------ dec (signed):          0
x26 .------ hex: 0x00000000 .------ dec (signed):          0
x27 .------ hex: 0x00000000 .------ dec (signed):          0
x28 .------ hex: 0x00000000 .------ dec (signed):          0
x29 .------ hex: 0x00000005 .------ dec (signed):          5
x30 .------ hex: 0x00000040 .------ dec (signed):         64
x31 .------ hex: 0x00000000 .------ dec (signed):          0
----------------------------------------------------------------

Testbench/testbench.sv:64: $finish called at 2020000 (1ps)
```

Figure 2: Final state of the register file as displayed in the simulation console by the testbench. The values confirm the correct execution of the entire test program.

3.2.3 Simulation Waveforms

While the final register file state confirms the overall correctness of the program execution, it does not provide insight into the internal, cycle-by-cycle behavior of the pipeline. To

verify the dynamics of the hazard management system, the simulation output was captured in a VCD file and analyzed using the GTKWave waveform viewer. This allows for a microscopic inspection of the hardware's behavior at critical moments.

EX/MEM to EX Forwarding in Action The first critical mechanism to verify is the highest-priority forwarding path. This occurs when an instruction in the EX stage requires the result of the instruction immediately preceding it, which is currently in the MEM stage. The test program triggers this with a 'sub' instruction followed by an 'add' that uses the 'sub''s result.

Figure 3 captures this exact event. The vertical cursor marks the rising edge of the clock where the 'add' instruction is in the EX stage and the 'sub' instruction is in the MEM stage. The waveform shows:

- **The Condition:** At the cursor, the instruction in MEM has a destination register 'RdM' of '0x06' (x6) and its write enable signal 'RegWriteM' is high. The instruction in EX needs this register as both of its sources, with 'Rs1E' and 'Rs2E' both being '0x06'.
- **The Action:** The Hazard Unit correctly detects this dependency. As a direct result, it asserts both forwarding signals, 'ForwardAE' and 'ForwardBE', to '2'b01'. This value instructs the ALU input multiplexers to select the result coming from the EX/MEM pipeline register.
- **The Result:** The data inputs to the ALU, 'alu_src_a' and 'alu_src_b', receive the correct, forwarded value of '0x009FFFFB' (the result of the 'sub' operation). This successfully bypasses the stale data that would have otherwise been read from the register file.

This visual evidence proves conclusively that the EX/MEM to EX forwarding path is implemented correctly, operating within a single clock cycle and preventing any unnecessary pipeline stalls.

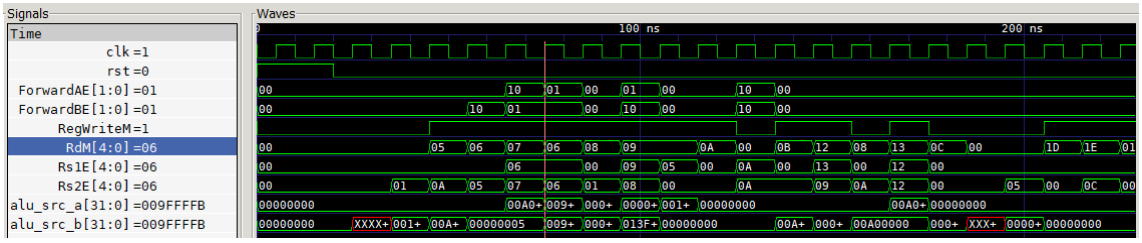

Figure 3: Waveform demonstrating EX/MEM to EX forwarding. At the rising edge marked by the cursor, the Hazard Unit detects the dependency between the `sub` instruction (in MEM) and the `add` instruction (in EX). It asserts `ForwardAE` and `ForwardBE` to '01', routing the fresh result ('0x009FFFFB') to the ALU inputs.

Load-Use Hazard and Pipeline Stall The most complex data hazard is the load-use case, where an instruction requires a value that is being loaded from memory by the instruction immediately preceding it. This dependency cannot be resolved by forwarding alone and necessitates a one-cycle pipeline stall. The test program triggers this exact scenario with a 'lw' instruction followed by an 'add'.

Figure 4 provides a detailed view of the moment the Hazard Unit detects and handles this event. The cursor marks the rising clock edge where the 'lw' instruction is in the

EX stage and the dependent ‘add‘ instruction is in the ID stage. The waveform clearly demonstrates the detection and resolution logic:

- **The Condition:** At the cursor, the control signal ‘ResultSrcE‘ is ‘01‘, indicating that the instruction in the EX stage is a load. Its destination register, ‘RdE‘, is ‘0x0A‘ (register x10). Concurrently, the instruction in the ID stage requires this same register as its source operand, as shown by ‘Rs1D‘ also being ‘0x0A‘.
- **The Action:** The Hazard Unit correctly identifies this critical dependency. In response, it immediately asserts its three primary control signals for this situation: ‘StallF‘, ‘StallD‘, and ‘FlushE‘ all go high.
- **The Consequence:** These signals orchestrate a perfect one-cycle stall. ‘StallF‘ and ‘StallD‘ freeze the PC and the IF/ID pipeline register, holding the ‘add‘ instruction in the Decode stage. ‘FlushE‘ injects a bubble (a NOP) into the EX stage, giving the ‘lw‘ instruction the crucial extra cycle it needs to fetch its data from memory. As shown in the waveform, these signals remain asserted for exactly one clock cycle before returning to zero, after which the pipeline resumes normal operation.

This visual confirmation is definitive proof that the stall and flush logic is correctly implemented, preventing data corruption in the critical load-use scenario.

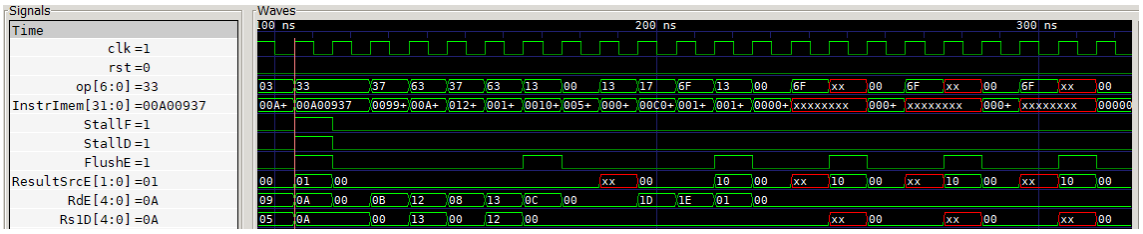

Figure 4: Waveform of the load-use hazard stall. At the cursor, the processor detects that the `lw` instruction in EX (identified by `ResultSrcE=01`, destination `RdE=0A`) is a source for the instruction in ID (`Rs1D=0A`). This triggers a one-cycle assertion of `StallF`, `StallD`, and `FlushE` to resolve the hazard.

Control Hazard and Pipeline Flush The final verification step focuses on control hazards. When a branch is taken, the instructions that the processor has speculatively fetched based on the default sequential flow (‘PC+4‘) are incorrect and must be squashed from the pipeline. This is achieved by "flushing" the affected pipeline stages. The test program forces this scenario with a ‘beq‘ instruction that is guaranteed to be taken.

Figure 5 captures the exact moment this control hazard is resolved. The cursor is placed on the rising edge of the clock immediately after the ‘beq‘ instruction has been evaluated in the EX stage. The waveform provides clear evidence of the entire process:

- **The Trigger:** In the cycle marked by the cursor, the ‘PCSrcE‘ signal, which determines the source for the next program counter value, transitions to ‘2’b01‘. This indicates that the branch condition was met and the processor must jump to the target address, abandoning the speculative path.
- **The Response:** The Hazard Unit detects this change in control flow. Its immediate response is to assert both ‘FlushD‘ and ‘FlushE‘. These signals are responsible for converting the incorrect instructions, which at this point are in the ID and EX stages,

into NOPs. As can be seen, the signals remain high for one full clock cycle to ensure the incorrect instructions are completely cleared.

- **The Recovery:** A few cycles after the flush, the ‘InstrImem‘ signal shows the processor fetching the instruction from the correct branch target address (‘00500E93‘, which corresponds to ‘addi x29, x0, 5‘). This confirms that the PC was successfully updated to the branch target and that the pipeline has resumed correct execution.

This waveform provides a definitive, visual confirmation that the control hazard mechanism is functioning as designed, ensuring the integrity of the program's execution path even after abrupt changes in control flow.

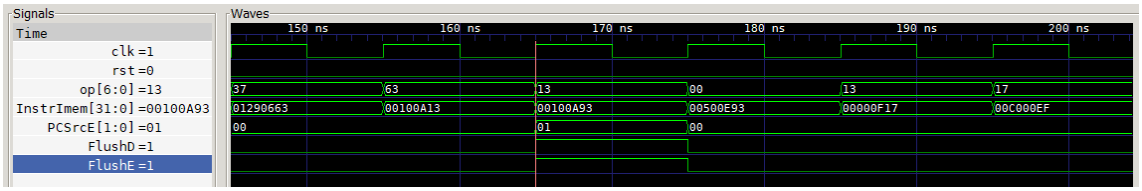

Figure 5: Waveform of a control hazard due to a taken branch. At the cursor, the `beq` instruction is resolved, causing `PCSrcE` to go high. The Hazard Unit reacts by asserting `FlushD` and `FlushE` to squash the two speculatively fetched instructions, allowing the pipeline to correctly fetch from the target address.

4 Results and Discussion

The verification process detailed in the previous chapter confirmed that the implemented processor is functionally correct and successfully handles the complex data and control hazards inherent in a pipelined architecture. This section moves beyond simple functional correctness to discuss the qualitative aspects of the project. It will analyze the challenges encountered during the design and debugging phases, reflect on the performance characteristics of the final design, and consider its limitations and potential avenues for future improvement.

4.1 Challenges Encountered

The design and implementation of a pipelined processor, while being a well-documented academic problem, presented a number of practical challenges that moved well beyond theoretical understanding. Overcoming these hurdles was a fundamental part of the learning experience of this project.

- **Managing Pipeline Complexity and Hazard Logic:** The most significant and persistent challenge was the implementation of the Hazard Unit. While the theory behind forwarding and stalling is straightforward, translating it into precise, cycle-perfect control logic proved to be highly complex. The ‘load-use‘ hazard was particularly difficult to manage, as it required a delicate orchestration of stalling the early stages of the pipeline while simultaneously flushing a subsequent stage to insert a bubble. An incorrect signal at the wrong clock edge could lead to subtle data corruption that was difficult to trace.

- **Bridging Theory and Practice in Pipelining:** A surprisingly difficult aspect was moving from the abstract, block-diagram-level understanding of a pipeline to its concrete implementation in SystemVerilog. Ensuring that every necessary control and

data signal was correctly defined, passed between stages through the pipeline registers, and utilized at the correct time required meticulous planning. Early iterations of the top-level module struggled with integrating all the sub-modules, highlighting the significant gap between understanding the function of each component and making them work together as a cohesive whole.

- **A Subtle Bug in Immediate Forwarding:** One of the most challenging bugs to debug occupied several days of work. It was discovered that 'ADDI' instructions were not producing the correct result. The issue was initially mistaken for a problem in the ALU or the Immediate Extender module. However, after extensive waveform analysis, the root cause was traced back to an error in the forwarding logic for the ALU's second operand. The multiplexer that selects between the register value ('rs2') and the immediate value was not being controlled correctly in the presence of other, unrelated forwarding conditions. This bug underscored the intricate dependencies within the datapath and the importance of exhaustive testing.
- **Implementing Complex Control Flow:** While standard branches ('beq', 'bne') were relatively straightforward, implementing the 'JALR' (Jump and Link Register) instruction presented its own set of challenges. Correctly calculating the target address (a sum of a register value and an immediate) and ensuring this address was fed back to the program counter at the right time, all while managing the pipeline flush, required careful integration between the ALU, the Hazard Unit, and the PC control logic.

4.2 Performance Considerations

The primary motivation for implementing a pipelined architecture is to improve performance by increasing instruction throughput. In an ideal scenario, the processor could approach a throughput of one instruction completed per clock cycle (a CPI of 1.0). However, the real-world performance is invariably lower due to the very hazards this project set out to manage. The final performance of this processor is therefore a direct function of how efficiently it handles these hazards.

The Positive Impact of Forwarding Forwarding is the single most important performance-enhancing feature in this design. Without it, almost every sequence of dependent instructions would require the pipeline to stall for one or two cycles, waiting for a result to be written back to the register file. Given the high frequency of such dependencies in typical programs, the processor's performance would degrade catastrophically, likely performing no better than a simpler, non-pipelined multi-cycle design.

The implemented forwarding logic for both ALU-to-ALU (EX to EX) and Memory-to-ALU (MEM to EX) dependencies successfully mitigates the vast majority of data hazards without any cycle penalty. This allows long sequences of arithmetic and logical operations to execute at the ideal throughput of one instruction per cycle, maintaining the core benefit of the pipelined approach.

The Necessary Cost of Stalls and Flushes While forwarding is highly effective, it cannot resolve all hazards. The performance of the processor is therefore limited by two key scenarios that incur mandatory cycle penalties:

- **Load-Use Hazard Stalls:** As demonstrated in the verification chapter, the 'load-use' hazard imposes an unavoidable one-cycle stall. Every time a program loads a

value from memory and attempts to use it in the very next instruction, the pipeline loses one cycle of work. The overall performance impact is therefore directly proportional to the frequency of this specific instruction pattern in the executed code. Well-scheduled code that inserts independent instructions after a load can mitigate this, but it cannot always be avoided.

- **Control Hazard Flushes:** Changes in control flow are the second major source of performance degradation. This processor uses a "predict not-taken" strategy for branches. While this is effective for loops and sequential code, every *taken* branch or unconditional jump (‘JAL‘, ‘JALR‘) forces the pipeline to flush the two instructions that were speculatively fetched. This effectively costs two cycles of lost work for every taken branch or jump. In programs with frequent branching or complex control flow, this "branch penalty" can become a significant bottleneck, noticeably reducing the processor's overall instruction throughput.

In conclusion, the performance of this 5-stage pipeline is a balance. It achieves near-ideal throughput for linear sequences of register-based operations thanks to aggressive forwarding, but its performance is inevitably reduced by the necessary, penalty-inducing stalls and flushes required to maintain functional correctness in the face of memory access and control flow changes.

4.3 Limitations and Future Improvements

This project successfully achieved its primary goal of implementing a functional, 5-stage pipelined processor for the base integer (RV32I) instruction set. However, as an academic project, it was designed with a constrained scope. There are several clear limitations that offer exciting avenues for future work and improvement.

- **Instruction Set Extensions:** The processor currently implements only the base 32-bit integer instruction set (RV32I). A significant area for future work would be to incorporate the standard RISC-V extensions, which would dramatically increase the processor's capabilities:
  - The **M Extension** for integer multiplication and division.
  - The **A Extension** for atomic memory operations, crucial for synchronization in multi-threaded or multi-core environments.
  - The **F and D Extensions** for single- and double-precision floating-point arithmetic.
  - The **C Extension** for compressed instructions, which improves code density and can reduce pressure on the instruction fetch stage.
- **Absence of a Privilege and CSR System:** A major limitation of the current design is the lack of a system for privileged operations and Control and Status Registers (CSRs). Consequently, a number of important instructions from the RV32I base set are not implemented. These include:
  - Memory barrier instructions like ‘FENCE‘ and ‘FENCE.I‘, which are essential for managing memory consistency.
  - System call (‘ECALL‘), breakpoint (‘EBREAK‘), and environment-related instructions.

  - The entire suite of CSR instructions (‘CSRRW‘, ‘CSRRS‘, etc.), which are the standard mechanism for managing system state, such as timers, interrupts, and performance counters.

  Implementing a minimal privilege model and the CSR register file would be a critical next step toward making the processor capable of running an operating system.

- **Lack of Exception and Interrupt Handling:** The processor does not currently handle exceptions (e.g., illegal instructions, misaligned memory accesses) or external interrupts. In its current state, such an event would lead to unpredictable behavior. A future improvement would be to add logic to detect these events, save the processor state (e.g., the current PC), and jump to a predefined exception handler routine. This is a fundamental feature for any robust computing system.

- **Memory System and Performance Enhancements:** The current memory system is simplistic, consisting of single-cycle instruction and data memories. This is unrealistic for a real system. The most impactful performance improvement would be the addition of a **memory hierarchy**, including separate instruction and data caches. Adding caches would dramatically reduce the average memory access time but would also introduce new complexities, such as cache misses that require pipeline stalls. Furthermore, the performance penalty from control hazards could be reduced by implementing more advanced **branch prediction** logic instead of the static "predict not-taken" scheme.

5 Conclusion

This project chronicled the complete design, implementation, and verification of a 5-stage pipelined RISC-V processor for the RV32I instruction set. From the initial architectural design to the final verification through simulation, the project successfully navigated the core complexities of modern processor design, particularly the critical challenges of data and control hazard management. The resulting processor correctly executes a purpose-built test program by employing a combination of forwarding, stalling, and flushing to ensure functional correctness without sacrificing the performance benefits of a pipelined datapath.

5.1 Key Achievements and Lessons Learned

The primary achievement of this project is the creation of a working and verifiable pipelined processor, a task that required the integration of numerous hardware components into a single, cohesive system. Beyond the final artifact, however, the project yielded several invaluable lessons that transcend the specific implementation details.

First and foremost was the profound realization of the gap between theory and practical application. While concepts like pipelining and hazard resolution are clear on paper, implementing them in hardware description language, with cycle-by-cycle precision, revealed a much higher level of complexity. This led to a deep appreciation for the meticulous planning required in digital design.

A second critical lesson was the central role of verification in the design process. It became immediately apparent that a well-designed testbench is not an afterthought but the most powerful tool for initial debugging. The ability to quickly check the final state of the register file provided a high-level "pass/fail" metric. However, the most crucial lesson learned was the power of deep, visual debugging. Analyzing the processor's behavior

through waveform simulations proved to be the definitive method for isolating and resolving subtle bugs, transforming abstract problems into concrete, visible phenomena and making it possible to pinpoint the exact source of an error.

5.2 Relevance to Real-World Applications

While this processor is an academic project, the principles applied and skills developed are directly relevant to real-world digital electronics and semiconductor design. The entire project mirrors the standard industry workflow for developing digital hardware, from architectural specification and HDL implementation (in SystemVerilog) to functional verification and debugging.

The experience gained in managing pipeline logic, resolving hazards, and designing complex control units is fundamental to the development of high-performance CPUs, microcontrollers, and specialized accelerators (e.g., for AI or graphics). The systematic debugging skills, particularly the analysis of waveforms, are an essential competency for any digital design or verification engineer working on FPGAs. In essence, this project served as a practical, hands-on introduction to the core challenges and methodologies that define the field of modern digital hardware design.

Acknowledgments

I would like to acknowledge the broader open-source community. Several publicly available RISC-V processor projects on platforms such as GitHub provided valuable inspiration, particularly for the structural organization of the top-level module and for demonstrating different implementation patterns. Furthermore, I wish to acknowledge the use of an AI-powered language model in this project. The AI served as an instrumental tool for structuring and refining the text using LaTeX of this report, and acted as a valuable resource for clarifying complex concepts that were not immediately resolvable through standard documentation alone.

References

[1] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture, RISC-V Edition*, 2nd. Morgan Kaufmann, 2021.

[2] RISC-V International, *The risc-v instruction set manual, volume i: Unprivileged isa*, https://riscv.org/technical/specifications/, Document Version 20191213. Accessed on: June 30, 2025, 2019.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd. Morgan Kaufmann, 2021.

[4] L. Teske. "Riscv-asm: Online risc-v assembler." (), [Online]. Available: https://riscvasm.lucasteske.dev/.