Course Project Report

Introduction to Processor Architecture

Design and Implementation of a 64-bit RISC-V Processor

Abstract:

This report describes the design and implementation of a 64-bit RISC-V processor with pipeline and hazard detection. The processor is implemented in Verilog and can execute a subset of the RISC-V instruction set. The processor is pipelined to improve performance, and it includes hazard detection logic to prevent data hazards and control hazards. The processor is also equipped with a branch predictor to reduce the performance impact of control hazards.

Submitted by:

Hrishikesh Milind Gawas	Vignesh Vembar	Krish Pandya
2024122006	2023102019	2023102026

Academic Year: 2024 - 2025

Contents

1 Sequential CPU Design **2**
1.1 Overview . . . 2
1.2 Working of the CPU . . . 2
1.3 Assembler . . . 3
1.3.1 Introduction . . . 3
1.3.2 Assembler Overview . . . 4
1.3.3 Testbench Generation . . . 4
1.4 Simulation . . . 5
1.5 Visualisation . . . 7

2 Pipelined CPU Design Documentation **8**
2.1 Overview . . . 8
2.2 Pipeline Register Implementation . . . 9
2.3 Architectural Rationale and Implementation Details . . . 9
2.3.1 Pipeline Register Design Strategy . . . 9
2.3.2 Control Signal Propagation . . . 10
2.3.3 Data Hazard Resolution System . . . 10
2.3.4 Branch Prediction Implementation . . . 10
2.3.5 Forwarding Implementation . . . 12
2.4 Critical Design Tradeoffs . . . 12
2.5 Key Implementation Nuances . . . 12
2.5.1 Instruction Injection Protocol . . . 12
2.5.2 Stall State Tracking . . . 13
2.5.3 End-of-Program Detection . . . 13
2.6 Architectural Validation Points . . . 13
2.7 Simulation . . . 13
2.7.1 Case I: Loop with Data Dependencies . . . 13
2.7.2 Case II: Branch Prediction Testing . . . 14
2.7.3 Case III: Branch Misprediction Recovery . . . 15
2.7.4 Case IV: Load-Use Hazard Resolution . . . 16
2.7.5 Case V: Data Forwarding Validation . . . 17
2.7.6 Case VI: Control Hazard Complex Case . . . 18

3 Contribution and Conclusion **19**

1 Sequential CPU Design

1.1 Overview

The `cpu_sequential` module is the final wrapper module which features a **basic 64-bit RISC-V processor** designed to execute instructions sequentially. It includes key components such as the **program counter (PC), instruction memory, control unit, register file, ALU, and data memory**, which work together to execute instructions in a step-by-step manner.

1.2 Working of the CPU

The CPU follows the standard **instruction execution cycle**, which consists of the following stages:

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Write-back

1. Instruction Fetch

- Features a 64-bit program counter which holds the address of the current instruction. This updates at every positive clock edge to point to the next instruction or branch target.
- Features separate memory for instruction and data storage. This prevents data hazards, discussed in later sections.
- Fetches the instruction from the **instruction memory** using the `pc_current` value.

2. Instruction Decode

- Features **control unit** which decodes the instruction and generates control signals. These determine the type of operation such as memory operations, arithmetic operations, and branch execution.
- Features a **register bank** which provides data stores on two source registers (`rs1` and `rs2`). This unit also writes the computed result to the destination register (`rd`).

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 1.1: Control Signals

3. Execute

- Features ALU (Arithmetic Logic Unit) which performs arithmetic and logical operations. It takes two inputs `reg_read_data1` (First operand) and `alu_operand2` (Second operand - register or immediate value) and produces `alu_result` and the `zero` flag (useful for branch instruction *beq*).

- This processor supports a subset of RISC-V instructions:
  - R-type: add, sub, or, and
  - I-type: addi, ld (load doubleword)
  - S-type: sd (store doubleword)
  - SB-type: beq (branch if equal)
  - Special: nop (no operation)
- If a branch instruction is executed, the `zero` flag determines whether to take the branch. The new PC address is calculated and provided to the multiplexer before `pc_next`. If no branch is taken, the PC increments sequentially (PC+4).

4. Memory Access

- Load instructions (`ld`) fetch data from **data memory** and Store instructions (`sd`) write data to memory.

5. Write-back

- The final result is written back to the register file. The value comes from either memory or the ALU result which is set using the `mem_to_reg` control signal.

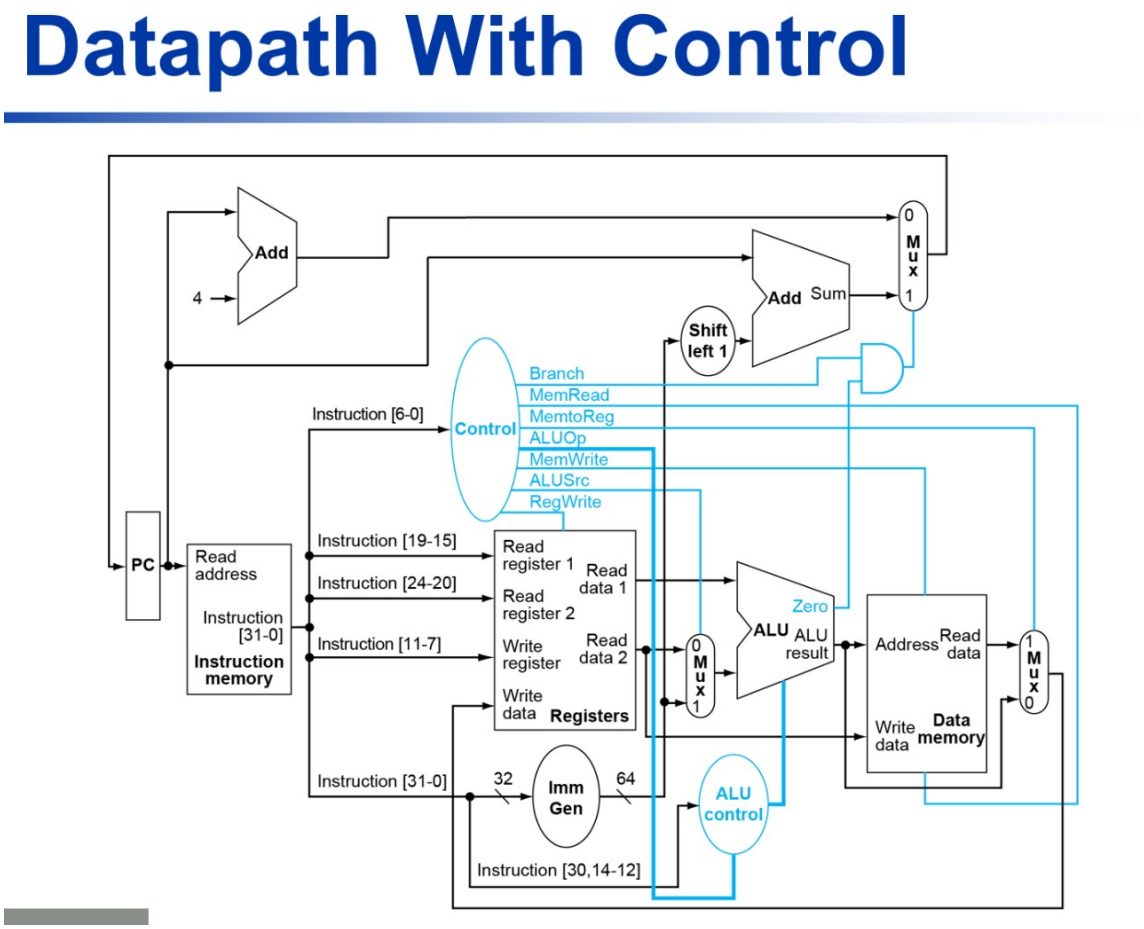

Figure 1.2: Sequential Datapath and Control Logic

1.3 Assembler

1.3.1 Introduction

An **assembler** is a program that translates assembly language instructions into machine code (binary format). The assembler processes RISC-V assembly instructions and generates corresponding binary instructions to be loaded into the instruction memory of the CPU.

1.3.2 Assembler Overview

The assembler operates in two passes:

1. **First Pass: Label Collection**
   - Reads the assembly file and scans for labels (used in branching instructions).
   - Assigns memory addresses to labels so they can be referenced later.
2. **Second Pass: Instruction Translation**
   - Converts each assembly instruction into its binary format based on the RISC-V instruction set.
   - Uses predefined opcode, function codes (`funct3`, `funct7`), and register encoding.
   - Replaces label references with corresponding memory addresses.

The output of this assembler is a list of **binary instructions** formatted in Verilog format, which is then inserted into the testbench.

1.3.3 Testbench Generation

The assembler generates a **testbench** in Verilog to verify the correctness of the `cpu_sequential` module. The testbench:

- Initializes a clock signal (`clk`) with a 10ns period.
- Generates a **reset** signal at the beginning to initialize the CPU.
- Loads the generated machine instructions into the **instruction memory** (`imem`).
- Simulates execution by stepping through the instruction cycle. The simulation runs until the CPU encounters an all zero bit **NOP (halt)** instruction.
- Monitors CPU behavior, registers, and memory operations at every cycle which can be viewed on terminal as shown below.

```
Time=605000
PC=0000000000000030
Instruction=000606b3
Executing: add/sub x13, x12, x0
Register values:
rs1(x12)=55
rs2(x0)=0
Writing to rd(x13)=55 [0x0000000000000037]
Control signals:
branch=0, mem_read=0, mem_to_reg=0, mem_write=0, alu_src=0, reg_write=1
ALU result=55 [0x0000000000000037]
```

Figure 1.3: Cycle wise monitoring

- After execution, the testbench prints the final values of **registers** and **memory**. It also writes memory contents to a file (`data_memory.hex`).
- This measures the **execution time** of the program in cycles.

Finally, a bash script is written which integrates all files such as assembler, testbench generated and all the modules to simulate the results in one go!

1.4 Simulation

Case I

Initially, a basic assembly sketch is implemented to test whether all functions work properly.

```
main:
    addi x1, x0, 3        # x1 = 3
    addi x2, x0, 7        # x2 = 7
    addi x3, x0, 1        # x3 = 1 (for AND operation)

loop:
    beq x1, x0, exit      # if x1 == 0, exit
    add x2, x2, x1        # x2 = x1 + x2
    sub x1, x1, x3        # x1 = x1 - 1 (same as addi x1, x1, -1)

    and x4, x1, x2        # x4 = x1 AND x2 (dummy operation)
    or x5, x1, x2         # x5 = x1 OR x2 (dummy operation)

    beq x0, x0, loop      # loop

exit:
    sd x2, 0(x0)     # Store x2 in memory
    ld x6, 0(x0)     # Load result into x6 (dummy operation)
    nop
```

Figure 1.4: Basic Assembly

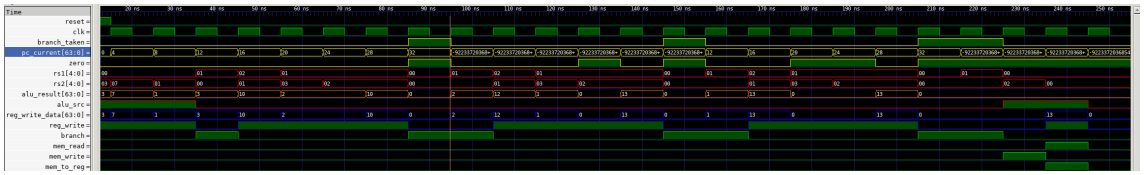

Figure 1.5: GTKWave Output

Case II

Some lengthy test cases were run to check edge cases.

```
start:
    addi x10, x0, 10    # Load n
    addi x11, x0, 0     # x11 = 0 (Fib(0))
    addi x12, x0, 1     # x12 = 1 (Fib(1))
    beq x10, x0, done   # If n == 0, return Fib(0)
    addi x10, x10, -1   # Decrement n by 1 to account for Fib(1)
    beq x10, x0, done1  # If n == 1, return Fib(1)

loop:
    add x13, x11, x12   # x13 = x11 + x12 (Fib(n) = Fib(n-1) + Fib(n-2))
    add x11, x12, x0    # x11 = x12 (shift Fib(n-1) to Fib(n-2))
    add x12, x13, x0    # x12 = x13 (shift Fib(n) to Fib(n-1))
    addi x10, x10, -1   # Decrement n
    beq x10, x0, done1  # Repeat until n == 0
    beq x0, x0, loop

done1:
    add x13, x12, x0    # Return Fib(1)

done:
    # x13 holds the Fibonacci result
    nop
```

Figure 1.6: n^{th} Fibonacci number

```
Time=605000
PC=0000000000000030
Instruction=000606b3
Executing: add/sub x13, x12, x0
Register values:
rs1(x12)=55
rs2(x0)=0
Writing to rd(x13)=55 [0x0000000000000037]
Control signals:
branch=0, mem_read=0, mem_to_reg=0, mem_write=0, alu_src=0, reg_write=1
ALU result=55 [0x0000000000000037]
```

Figure 1.7: Final Instruction Output

```
start:
    addi x10, x0, 20  # Load n
    addi x11, x0, 1   # x11 = 1 (accumulator for factorial)
    beq x10, x0, done # If n == 0, factorial is 1

loop:
    add x12, x0, x11  # x12 = x11 (initialize for multiplication)
    add x13, x0, x10  # x13 = x10 (multiplier counter)
    addi x11, x0, 0   # Reset x11 to 0

mul_loop:
    beq x13, x0, end_mul # If x13 == 0, end multiplication
    add x11, x11, x12    # x11 = x11 + x12 (multiplication by addition)
    addi x13, x13, -1    # Decrement multiplier counter
    beq x0, x0, mul_loop # Continue multiplication loop

end_mul:
    addi x10, x10, -1 # Decrement x10 by 1
    beq x10, x0, done # If x10 != 0, repeat loop
    beq x0, x0, loop

done:
    # x11 holds the factorial result
    nop               # End program (can replace with ecall to exit in real environment)
```

Figure 1.8: n^{th} Factorial

```
Register file contents:
x0 = 0 [0x0000000000000000]
x1 = 0 [0x0000000000000000]
x2 = 0 [0x0000000000000000]
x3 = 0 [0x0000000000000000]
x4 = 0 [0x0000000000000000]
x5 = 0 [0x0000000000000000]
x6 = 0 [0x0000000000000000]
x7 = 0 [0x0000000000000000]
x8 = 0 [0x0000000000000000]
x9 = 0 [0x0000000000000000]
x10 = 0 [0x0000000000000000]
x11 = 3628800 [0x0000000000375f00]
```

Figure 1.9: Final Register Content (x11)

1.5 Visualisation

To enhance understanding of the CPU's operation, we developed a visualization tool that renders the processor's architectural components and data flow in real time. This interactive interface depicts the key components of the CPU including the register file, ALU, memory units, and control paths, allowing observation of instruction execution at each stage. Our visualization approach provides intuitive insights into the processor's behavior by highlighting active data paths and changing register values during program execution, making it especially valuable for debugging and educational purposes.

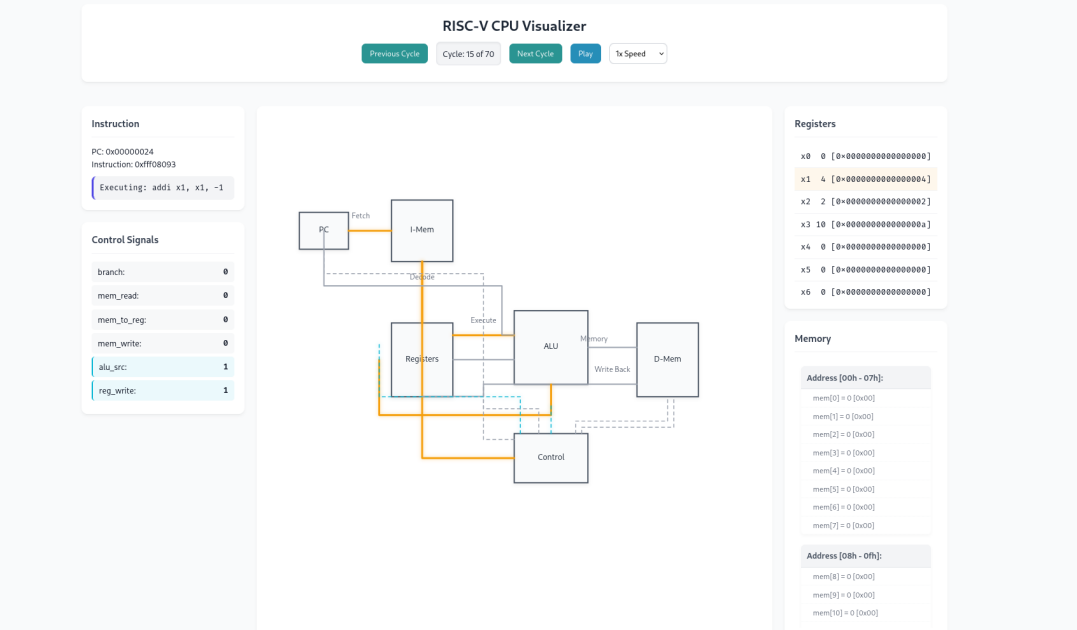

Figure 1.10: RISC-V CPU Interactive Visualization Interface

2 Pipelined CPU Design Documentation

2.1 Overview

This 5-stage pipelined RISC-V CPU implements a RISC-V subset with hazard handling and branch prediction. Key components are organized into Instruction Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB) stages connected through pipeline registers. This design is largely based on principles and architectures described in Harris and Harris [2012] and Patterson and Hennessy [2020]. The control sequence was designed keeping in mind the standard 5-stage pipeline operation.

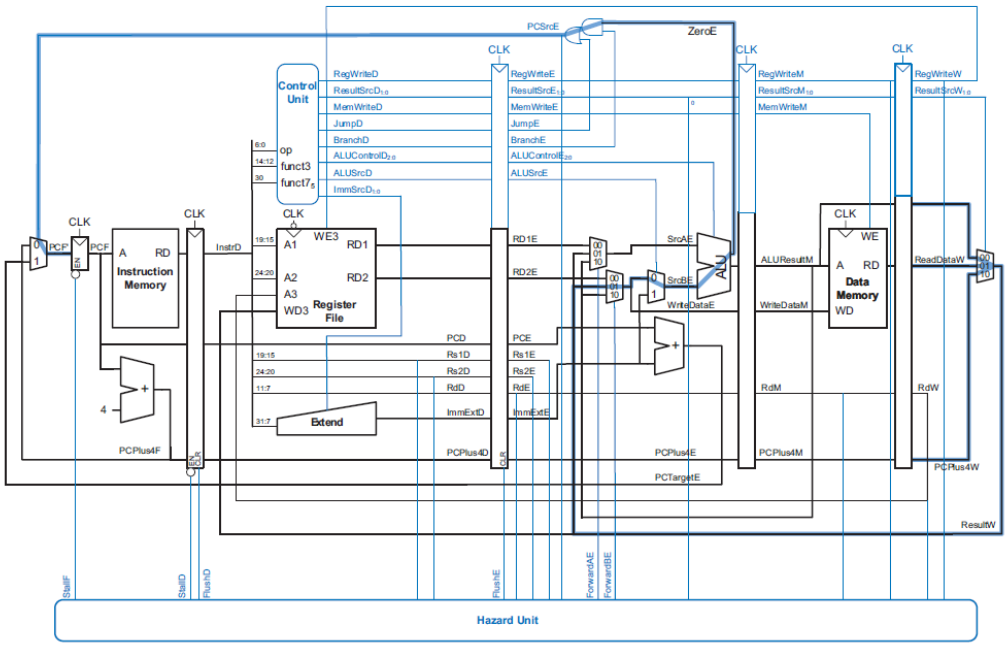

Figure 2.1: 5-Stage Pipeline Architecture with Forwarding Paths

2.2 Pipeline Register Implementation

Four pipeline registers manage intermediate results between stages which are:

- **IF/ID Register**: Stores PC value and fetched instruction

  Inserts NOPs during stalls/flushes. The enable signal `en` controls the register update, active when not stalled or during branch prediction, and also if stalled in the last cycle to ensure forward progress after a stall (this has to be done for load hazard) .

- **ID/EX Register**: Preserves control signals and decoded operands

  This register captures the decoded instruction information and control signals, forwarding them to the EX stage. The enable signal `en` is similar to IF/ID, ensuring data is held during stalls but progresses otherwise .

- **EX/MEM Register**: Carries ALU results and memory addresses

  This register stores the results from the EX stage, such as ALU results, data for memory operations, and control flags, for use in the MEM stage. It is always enabled (`en` is 1'b1) as it needs to pass data every cycle when the pipeline is not flushed by reset. (Why enable always you might ask? Because we need to pass the data every cycle and we don't want to stop the data flow in the pipeline and in our design choice we decided to pass (addi x0 x0 0) which is actually the instruction that RISC-V Uses.)

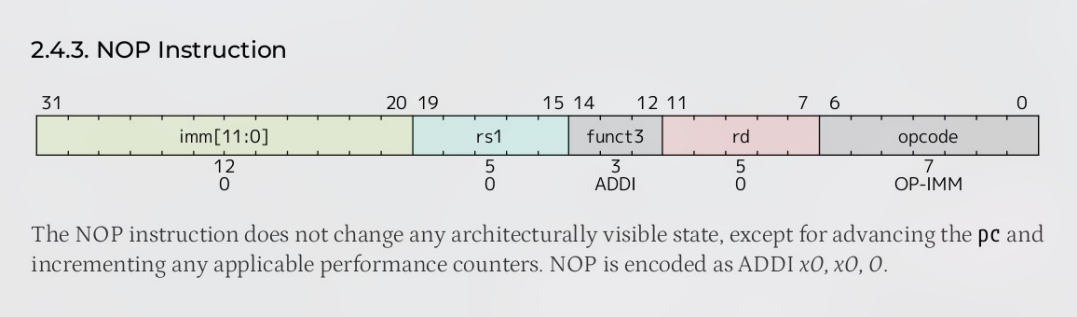

Figure 2.2: Stalling Instruction in RISC-V

- **MEM/WB Register**: Holds data for write-back operations

  This register carries data from the MEM stage to the WB stage, including memory read data and ALU results, for the final write-back to the register file. Similar to EX/MEM, it's also always enabled to ensure data progresses in the pipeline.

2.3 Architectural Rationale and Implementation Details

2.3.1 Pipeline Register Design Strategy

- **Naming Convention**: Registers named `<stage1>_<stage2>_register` indicate inter-stage boundaries (e.g., IF/ID separates Fetch/Decode)

- **Data Preservation**: Each register stores complete stage output:

```
// EX/MEM register contents
input [63:0] id_ex_pc, alu_result, operandB, branch_target;
input [31:0] id_ex_instruction;
input id_ex_zero, id_ex_branch, id_ex_mem_read, ... ;
```

  As illustrated, the `ex_mem_register` module (and similarly for others) bundles all necessary outputs from the EX stage needed by subsequent stages. This includes the program counter, ALU result, operand B, branch target, instruction itself, zero flag, and control signals related to branch, memory read/write, memory to register transfer and register write operations.

- **Flush/Stall Handling**: Reset or inject NOPs during control hazards. The `reset` and `flush` inputs are used to clear pipeline registers during control hazards or at system reset. For data hazards, specifically load-use hazards, NOP instructions are injected into the pipeline to create a stall cycle, as evident in the IF/ID register instantiation:

```
wire [31:0] instr_to_use = flush ? 32'h00000013 : instruction;  // NOP when flush
if_id_register if_id(
    // ...
    .d({
        pc_current,
        stall & ~stalled_last_cycle ? 32'h00000013 : instr_to_use,
        stall & ~stalled_last_cycle ? 1'b1 : nop_instruction,
        branch_predicted,
        predicted_pc
    }),
    // ...
);
```

Here, when a stall condition is detected and it's not a continued stall from the last cycle, a NOP instruction (32'h00000013, which is addi x0, x0, 0) is inserted into the IF/ID pipeline register. As discussed before in 2.2, this instruction is used to stall the pipeline for one cycle to resolve load-use hazards.

2.3.2 Control Signal Propagation

- **Stage-Specific Generation**: Control signals originate in ID stage. The `control_unit` module is instantiated in the ID stage and generates control signals based on the decoded instruction from the IF/ID register:

```
control_unit ctrl(
    .instruction(if_id_instruction),
    .branch(branch),
    .mem_read(mem_read),
    .mem_to_reg(mem_to_reg),
    .mem_write(mem_write),
    .alu_src(alu_src),
    .reg_write(reg_write)
);
```

- **Pipeline Carry-Through**: Signals propagate with instruction flow. The control signals generated in the ID stage, such as `branch`, `mem_read`, `mem_write`, `mem_to_reg`, `reg_write`, and `alu_src`, are then passed into the ID/EX pipeline register:

```
id_ex_register id_ex(.d({..., branch, mem_read, ...}));
```

These control signals then move down the pipeline with the instruction, being registered in subsequent pipeline registers (EX/MEM, MEM/WB if needed for later stages).

2.3.3 Data Hazard Resolution System

Table 2.1: Hazard Handling Mechanisms

Hazard Type	Detection Method	Resolution
Load-Use	ID stage register dependency check	1-cycle stall
Data Hazards (EX MEM)	Forwarding logic in EX stage	Forwarding
Control	Branch misprediction in EX stage	Pipeline flush

2.3.4 Branch Prediction Implementation

- **Static Prediction**: Our processor implements a simple static branch prediction strategy.

```
assign branch_predicted = is_branch;
```

This line implements an "always-taken" prediction policy. When an instruction is identified as a branch (by checking if its opcode matches the RISC-V branch opcode `7'b1100011`), we predict it will be taken. This detection happens early in the Instruction Fetch stage:

```
wire is_branch = (instruction[6:0] == 7'b1100011);
```

- **Target Calculation**: For predicted-taken branches, we calculate the target address immediately:

```
wire [63:0] if_branch_target = pc_current + if_branch_offset;
```

This calculation occurs during the IF stage, allowing us to redirect the PC without waiting for later stages. The branch offset is extracted and sign-extended from the instruction according to the RISC-V B-type format:

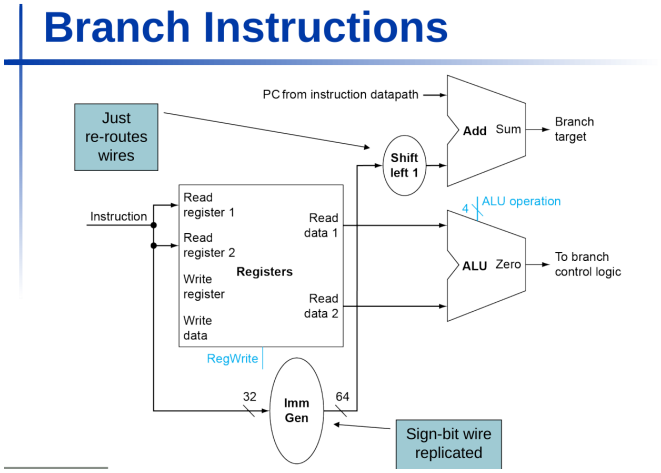

Figure 2.3: Branch Offset Calculation

- **Misprediction Recovery**:

```
assign flush = branch_mispredicted & ~id_ex_branch_predicted;
```

This critical line determines when to flush the pipeline after a branch misprediction.

  - `branch_mispredicted` indicates that the actual branch outcome (determined in EX stage) differs from what was predicted
  - `~id_ex_branch_predicted` means the branch was predicted as NOT taken

The combination `branch_mispredicted & ~id_ex_branch_predicted` means: "We have a misprediction AND the branch was predicted NOT taken but should have been taken." This specific condition matters because:

1. If we predicted a branch as taken but it shouldn't be , we've fetched instructions from the wrong path (honestly our skill issue), but we only need to redirect the PC without flushing.
2. If we predicted a branch as NOT taken but it should be , we've already started executing instructions after the branch that should never execute!!, so we must flush these from the pipeline. THIS IS NUCLEAR and we do it with flush remember from pipeline registers we mentioned we have flush going on all of them hahahahah it gets used now. Indeed Beautiful.

When the `flush` signal is asserted, it resets the IF/ID and ID/EX pipeline registers, effectively discarding any incorrectly fetched instructions and allowing execution to continue from the correct path.

2.3.5 Forwarding Implementation

```
// EX hazard (ALU result not yet written)
if (ex_mem_reg_write && ex_mem_rd != 0 && ex_mem_rd == id_ex_rs1 && !
    ex_mem_mem_to_reg)
forwardA = 2'b10;
// MEM hazard (completed execution)
if (mem_wb_reg_write && mem_wb_rd != 0 && mem_wb_rd == id_ex_rs1)
forwardA = 2'b01;
```

This Verilog snippet illustrates the forwarding logic for operand A in the EX stage. It checks for RAW hazards by comparing the destination register (`ex_mem_rd`, `mem_wb_rd`) of instructions in the EX/MEM and MEM/WB stages with the source register (`id_ex_rs1`) of the instruction in the EX stage.

- **EX Hazard Forwarding (2'b10)**: If there's a match with the EX/MEM stage's destination register and it's not a load instruction (indicated by `!ex_mem_mem_to_reg`), forward from the EX/MEM ALU result. The condition `ex_mem_rd != 0` ensures that we don't forward if the destination register is x0.
- **MEM Hazard Forwarding (2'b01)**: If there's a match with the MEM/WB stage's destination register, forward from the MEM/WB stage's result. This could be either from memory (`mem_wb_mem_read_data` if `mem_wb_mem_to_reg` is true) or ALU result. This forwarding path takes precedence over the EX hazard forwarding, ensuring the most up-to-date data is used, especially after a load-use stall is resolved.
- **No Forwarding (default 2'b00)**: If none of the forwarding conditions are met, operand A is read from the register file (`id_ex_reg_read_data1`).

Literally the same logic is implemented for operand B using `forwardB`.

2.4 Critical Design Tradeoffs

- **Branch Prediction Simplicity vs Accuracy**: Static always-taken reduces complexity but may increase mispredictions. The choice of always-taken branch prediction is a trade-off favoring hardware simplicity over prediction accuracy. We will further continue to improve this project and add a better branch prediction.
- **Forwarding Complexity vs Performance**: Full forwarding network eliminates stalls for Data hazards (except load-use) but increases logic complexity. Implementing a full forwarding network adds complexity to the EX stage and control logic (which we admit was a skill issue could have implemented a full fledged forwarding network). However, it significantly reduces pipeline stalls due to Data hazards as well, improving overall performance by allowing dependent instructions to proceed without waiting for write-back.
- **Stall Mechanism vs Throughput**: Single-cycle load stalls balance pipeline depth with hazard frequency. Introducing a single-cycle stall for load-use hazards is a compromise. It simplifies the hazard detection and resolution logic but may slightly reduce throughput compared to more complex hazard avoidance techniques like load bypassing or more sophisticated stall mechanisms. We looked into those but couldn't implement.

2.5 Key Implementation Nuances

2.5.1 Instruction Injection Protocol

```
// NOP insertion during flush/stall
wire [31:0] instr_to_use = flush ? 32'h00000013 : instruction;
if_id_register if_id(.d({..., stall ? 32'h00000013 : instr_to_use, ...}));
```

To handle pipeline flushes (due to branch mispredictions) and stalls (due to load-use hazards), NOP (No Operation) instructions are injected into the pipeline. The code shows that during a flush or stall, the `instr_to_use` signal is assigned the NOP instruction (32'h00000013, which is `addi x0 x0 0`). This NOP instruction as discussed in Figure 2.2 is then fed into the IF/ID register, effectively bubbling through the pipeline and ensuring no unintended operations are performed during hazard resolution.

2.5.2 Stall State Tracking

```
reg stalled_last_cycle;
always @(posedge clk) stalled_last_cycle <= stall;
```

A register `stalled_last_cycle` is used to track if a stall occurred in the previous clock cycle. This is crucial for the stall mechanism to work correctly for load-use hazards. When a load-use hazard is detected (indicated by the `stall` signal), the pipeline stalls for one cycle. In the next cycle, even if the hazard condition is still theoretically present (though now resolved by the stall cycle), the pipeline must proceed. The `stalled_last_cycle` flag ensures that the stall is only for one cycle, allowing the pipeline to resume in the subsequent cycle, as reflected in the enable condition of pipeline registers like IF/ID and ID/EX: `.en( stall | stalled_last_cycle)`.

2.5.3 End-of-Program Detection

```
assign end_program = mem_wb_nop_instruction &
~branch_mispredicted &
~branch_predicted;
```

The `end_program` signal is asserted to indicate the program's termination. It is designed to detect when a NOP instruction reaches the Write-Back stage (`mem_wb_nop_instruction`) and there are no ongoing branch mispredictions or predictions (`branch_mispredicted & ~branch_predicted`). This condition ensures that the program is considered finished only after all instructions, including any NOPs injected due to stalls or flushes, have completed the pipeline and there are no pending control flow changes from branch handling.

2.6 Architectural Validation Points

- **Forwarding Path Verification**: Validation of all EX/MEM and MEM/WB bypass scenarios confirms the correct data flow and hazard resolution through forwarding.
- **Branch Recovery Sequence**: Testing the flush signal timing and PC correction ensures that mispredicted branches are correctly handled, and the pipeline recovers to the correct execution path.
- **Load-Use Hazard Handling**: Verification of single-cycle stall insertion confirms that load-use hazards are detected and resolved by stalling the pipeline appropriately, preventing incorrect data usage.
- **Control Signal Propagation**: Checking signal integrity through pipeline stages validates that control signals are accurately generated in the ID stage and correctly propagated through subsequent pipeline registers, maintaining control over instruction execution throughout the pipeline.
- **Memory Alignment**: Validation of 64-bit data access boundaries ensures that memory operations are correctly aligned, which is critical for data integrity in a 64-bit architecture.

2.7 Simulation

To validate our pipelined CPU design, we performed extensive testing using a variety of assembly programs. Each test case was designed to verify specific aspects of pipeline functionality, including hazard handling, forwarding, and branch prediction. Ofcourse the Assembler was the same as sequential.

2.7.1 Case I: Loop with Data Dependencies

Our first test case exercised both branch prediction and data forwarding capabilities:

```
begin:
    addi x1, x0, 3      # x1 = 3
    addi x2, x0, 7      # x2 = 7

loop:
    beq x1, x0, exit    # if x1==0, end program
```

```
    add x2, x2, x1       # x2 = x1 + x2
    addi x1, x1, -1      # x1--
    beq x0, x0, loop     # loop

exit:
    nop
```

It validates:

- Register forwarding for the `add x2, x2, x1` instruction which uses updated x1 values
- Branch prediction for both conditional and unconditional branches
- Control hazard handling when the branch to exit is finally taken

Check how the final value in x2 is 13 as expected.

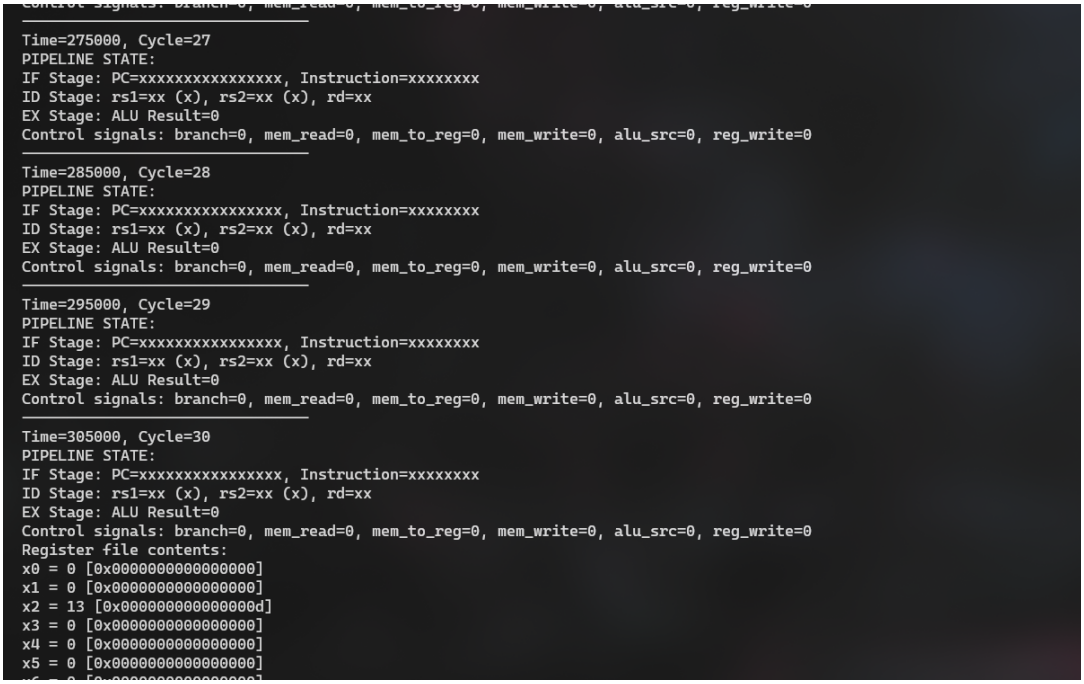

Figure 2.4: Loop with Data Dependencies - Pipeline Execution

2.7.2 Case II: Branch Prediction Testing

A very simple branch testing:

```
begin:
    beq x0, x0, L1
    addi x6, x0, 3

L1:
    addi x5, x0, 4

end:
    nop
```

This program includes an always-taken branch at the beginning that skips an instruction. It tests:

- Early branch detection in the IF stage
- Static "always-taken" prediction implementation check
- Correct execution path after branch prediction

See how x5 is 4 as expected.

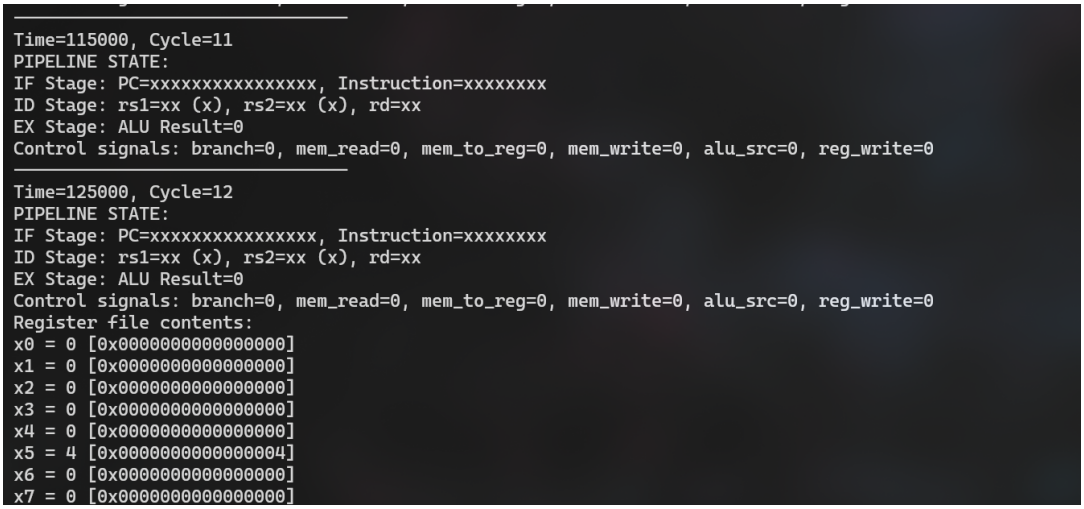

Figure 2.5: Branch Prediction Test - Pipeline Execution

2.7.3 Case III: Branch Misprediction Recovery

Our third test examines pipeline recovery after branch mispredictions:

```
begin:
    beq x0, x0, end

L1:
    addi x5, x0, 4

L2:
    addi x6, x0, 4

L3:
    addi x7, x0, 4

end:
    nop
```

This code challenges the pipeline (literally challenged me bro this stuff wasn't working lmao) with multiple branch targets and validates:

- Pipeline flush operation when branches are mispredicted
- NOP insertion during recovery
- PC redirection to the correct execution path
- As expected the output shows everything is 0.

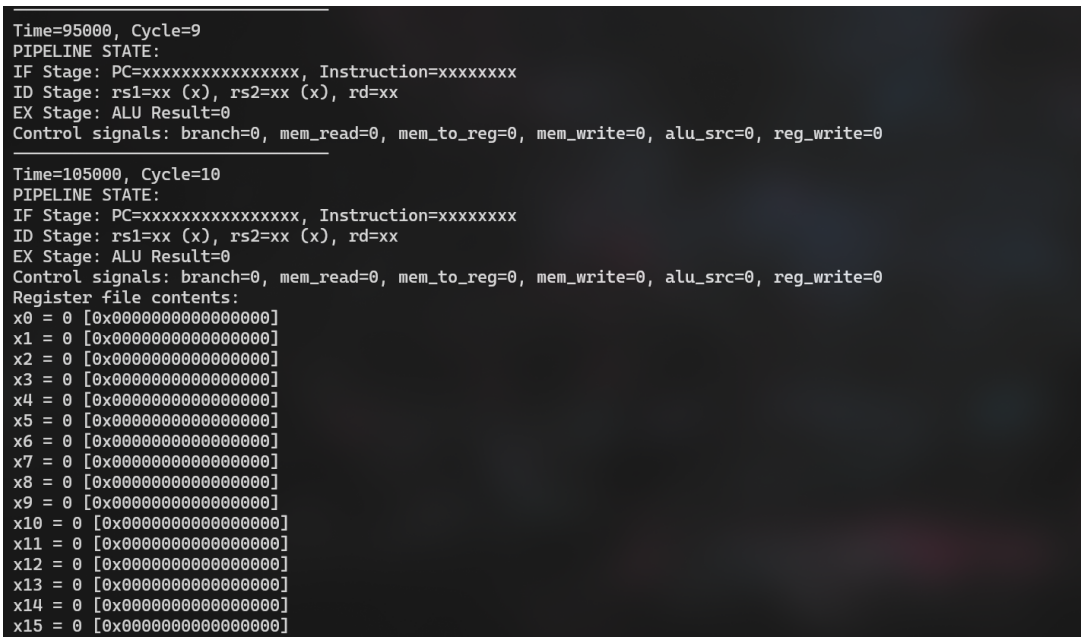

Figure 2.6: Branch Misprediction Recovery Test

2.7.4 Case IV: Load-Use Hazard Resolution

Tests the Load Hazard:

```
begin:
    ld x2, 0(x0)
    addi x4, x2, 1
    add x8, x6, x2
    add x9, x4, x2

end:
    nop
```

This code creates a classic load-use hazard where the value loaded into x2 is immediately needed by the next instruction. It confirms:

- Accurate detection of load-use dependencies
- Proper insertion of a single-cycle stall
- Correct resumption of pipeline flow after stall resolution

As you can see x0 was loaded with 5 before hand to see this output:

```
Time=145000, Cycle=14
PIPELINE STATE:
IF Stage: PC=xxxxxxxxxxxxxxxx, Instruction=xxxxxxxx
ID Stage: rs1=xx (x), rs2=xx (x), rd=xx
EX Stage: ALU Result=0
Control signals: branch=0, mem_read=0, mem_to_reg=0, mem_write=0, alu_src=0, reg_write=0
Register file contents:
x0 = 0 [0x0000000000000000]
x1 = 0 [0x0000000000000000]
x2 = 5 [0x0000000000000005]
x3 = 0 [0x0000000000000000]
x4 = 6 [0x0000000000000006]
x5 = 0 [0x0000000000000000]
x6 = 0 [0x0000000000000000]
x7 = 0 [0x0000000000000000]
x8 = 5 [0x0000000000000005]
x9 = 11 [0x000000000000000b]
x10 = 0 [0x0000000000000000]
x11 = 0 [0x0000000000000000]
x12 = 0 [0x0000000000000000]
x13 = 0 [0x0000000000000000]
x14 = 0 [0x0000000000000000]
x15 = 0 [0x0000000000000000]
x16 = 0 [0x0000000000000000]
x17 = 0 [0x0000000000000000]
x18 = 0 [0x0000000000000000]
x19 = 0 [0x0000000000000000]
x20 = 0 [0x0000000000000000]
x21 = 0 [0x0000000000000000]
x22 = 0 [0x0000000000000000]
x23 = 0 [0x0000000000000000]
x24 = 0 [0x0000000000000000]
x25 = 0 [0x0000000000000000]
x26 = 0 [0x0000000000000000]
x27 = 0 [0x0000000000000000]
x28 = 0 [0x0000000000000000]
x29 = 0 [0x0000000000000000]
x30 = 0 [0x0000000000000000]
x31 = 0 [0x0000000000000000]

Memory contents:
mem[0] = 5 [0x05]
mem[1] = 0 [0x00]
mem[2] = 0 [0x00]
mem[3] = 0 [0x00]
mem[4] = 0 [0x00]
```

Figure 2.7: Load-Use Hazard Resolution Test

2.7.5 Case V: Data Forwarding Validation

Double Forwarding Test:

```
begin:
    addi x1, x0, 5
    addi x2, x1, 1
    add x3, x0, x1
    addi x4, x1, 1

end:
    nop
```

The program creates multiple data dependencies in quick succession, ensuring that:

- EX-to-EX forwarding works correctly for the result of `addi x1, x0, 5`
- MEM-to-EX forwarding functions for later uses of x1
- Forwarding logic properly prioritizes the most recent value as seen by output:

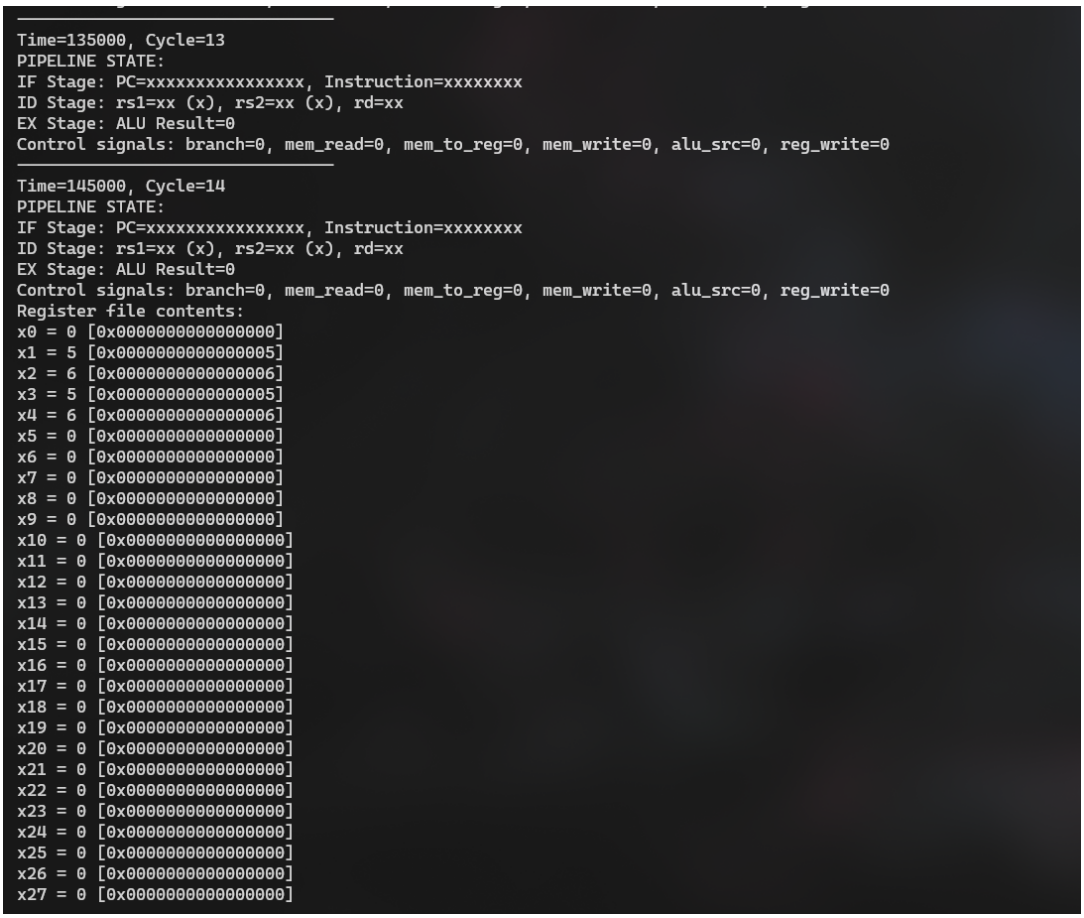

Figure 2.8: Data Forwarding Validation Test

2.7.6 Case VI: Control Hazard Complex Case

ok last test combines branch prediction with data dependencies:

```
begin:
    beq x0, x0, L1
    addi x7, x0, 4
    addi x8, x0, 5
    addi x9, x0, 6
    addi x10, x0, 10

L1:
    addi x5, x0, 4
    addi x6, x0, 5

end:
    nop
```

This test validates:

- Branch target calculation in the presence of multiple potential paths
- Instruction squashing after branch resolution
- Resumption of correct execution sequence

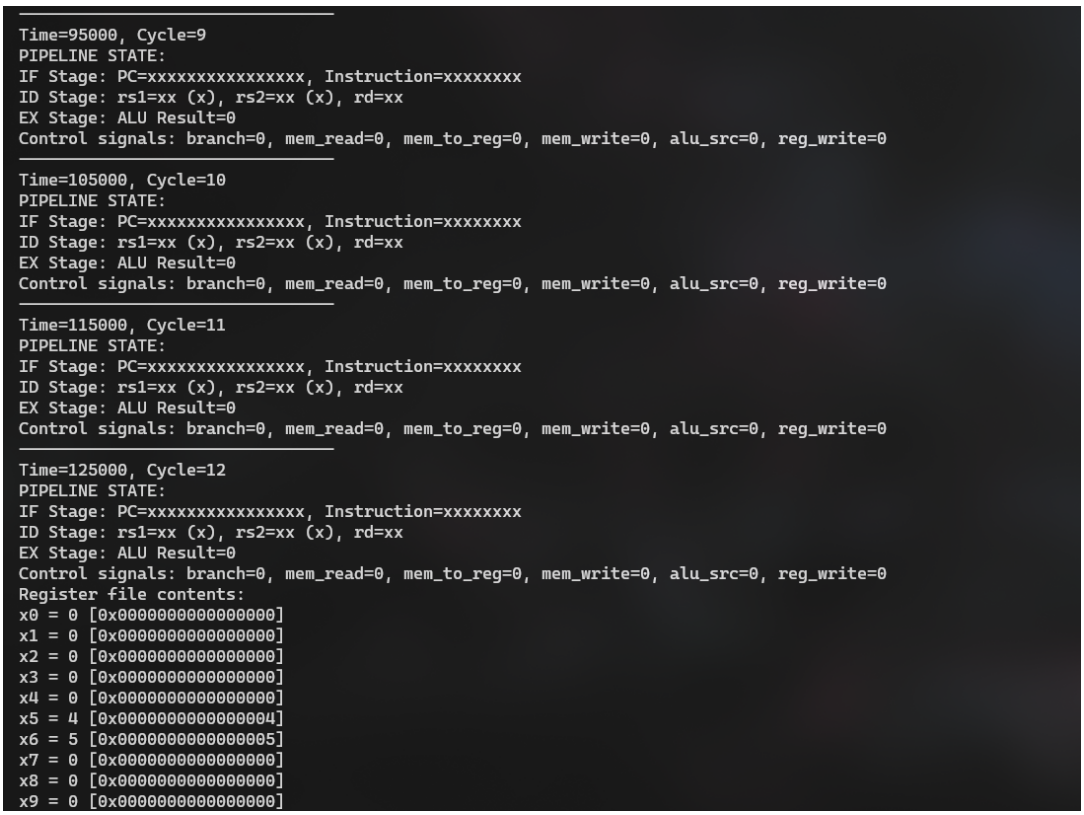

Figure 2.9: Control Hazard Complex Test

These tests confirm that our pipelined CPU correctly handles the key challenges of pipelined execution: data hazards through forwarding and stalling, and control hazards through prediction and recovery mechanisms.

3 Contribution and Conclusion

Well mostly it wasn't a 1/3 split but it was a 9/3 team effort, Hrishikesh worked on report writing , basic structure of Pipelining, desining registers and stuff along with creating beautiful test cases that we have right now , Vignesh the goat made the assembler and ALU , fixed a lot of bugs that we found out on the later stages of testing on even his amazing test cases , he handled the sequential fixes along with Hrishikesh the Data Hazards in pipelining, and for Krish, I finished (which I am writing right now) this last parts of report writing and all the hazards in pipelining along with web view, made the integration for Sequential and all the initial design blocks.

All in all it was more of a team effort so contributing any part to a person feels unjust as all of us worked on almost everything. We would also like to thank Prof Deepak Gangadharan and all the TAs for this amazing of a course, this was really a fun project to work on and we learnt a lot!

We will further work on this on our free time and extend the ISA as much as possible along with better hazard handling strategies that we will come up with.

Concluding here by saying that our implementation was sophisticated and aligned with the project's best requirements. Hazard handling was graceful, testing will be smoother due to the assembler, and the provided README would make the job much easier. The sequential design had really big programs running while maintaining speed. Future work includes extending the ISA and improving hazard handling strategies.

Adios!! Thanks for Reading

References

D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2012.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition*. Morgan Kaufmann, 2020.