Design and Implementation of a Pipelined RISC-V Processor

Weiyuan Du

January 2026

1 Analysis of System Architecture and Performance

1.1 System Architecture

1.1.1 Hazard Detection and Handling Logic

The Hazard Unit manages pipeline dependencies by monitoring register addresses (rf_a1, rf_a2, rf_a3) and control states ($we\_rf, sel\_result0, pcsrc$). According to the implemented Verilog module, RAW hazards are bypassed using a priority-based forwarding multiplexer, while Load-Use and Control hazards are resolved through a combination of stalls and flushes.

Table 1: Hazard Detection and Control Logic

Signal	**Logic Condition**	**Description / Result**
E_fd_A	$(E\_rf\_a1 == M\_rf\_a3) \land M\_we\_rf \land (E\_rf\_a1 \neq 0)$	10 (Fwd from MEM)
	else if $(E\_rf\_a1 == W\_rf\_a3) \land W\_we\_rf \land (E\_rf\_a1 \neq 0)$	01 (Fwd from WB)
E_fd_B	$(E\_rf\_a2 == M\_rf\_a3) \land M\_we\_rf \land (E\_rf\_a2 \neq 0)$	10 (Fwd from MEM)
	else if $(E\_rf\_a2 == W\_rf\_a3) \land W\_we\_rf \land (E\_rf\_a2 \neq 0)$	01 (Fwd from WB)
$lwStall$	$((D\_rf\_a1 == E\_rf\_a3) \lor (D\_rf\_a2 == E\_rf\_a3)) \land E\_sel\_result0$	$F\_stall, D\_stall, E\_flush = 1$
E_pcsrc	Branch Taken signal from Execute stage	$D\_flush, E\_flush = 1$

1.1.2 Load-Use Flushing Concept

In the provided example (Figure 3b), the register values ($x2, x3, x4$) remain correct regardless of whether $PLR2$ is flushed because the $Stall$ signal freezes the PC and $PLR1$. This causes the dependent instruction to be re-dispatched and re-executed in the following cycle with the correct forwarded data, effectively overwriting any temporary incorrect results in the register file.

However, flushing is mandatory to prevent **side effects** during the stall cycle. If an instruction performs a non-idempotent operation, such as writing to memory, a lack of flushing would lead to architectural errors.

Demonstration Program: The following sequence demonstrates a scenario where flushing $PLR2$ is critical:

```
lw  x1, 0(x2)  # Load value into x1
sw  x1, 8(x2)  # Store x1 to memory (depends on lw)
```

Analysis:

- **With Flushing:** When the $lwStall$ is detected, E_flush is asserted. This clears the `sw` instruction's control signals (setting `MemWrite` to 0) in the Execute stage. The memory remains unchanged during the stall cycle.
- **Without Flushing:** The `sw` instruction would remain in the Execute stage with its `MemWrite` signal active. It would erroneously write the *stale* value of $x1$ to memory address `8(x2)` during the stall cycle, leading to data corruption before the correct value is eventually written in the next cycle.

1.1.3 Data Forwarding, Flushing, and Stalling

- **Data Forwarding:** Bypasses the Register File by routing M_alu_o or W_result directly to the ALU inputs via E_fd_A/B muxes.
- **Stalling:** Disables stage updates ($EN = 0$) to hold instructions, specifically used to wait for lw data.
- **Flushing:** Clears pipeline registers ($CLR = 1$) to invalidate instructions following a taken branch or during a load-use stall.

1.2 Performance Analysis

1.2.1 CPI Estimation for Test Assembly

Based on the provided assembly code and the implemented Hazard Unit logic, the performance is analyzed as follows:

- **Instruction Count (I):** 20 instructions are executed to reach the final loop.
- **Control Hazards:** Two jumps occur (one conditional `beq` and one `jal`). Each jump asserts `E_pcsrc`, leading to 2 flush cycles per jump (Total: 4 cycles).
- **Data Hazards:** One Load-Use hazard occurs between `lw x2` and `add x9`, triggering `lwStall` and adding 1 stall cycle.
- **Total Cycles (C):** I + Base Overhead (4) + Flushes (4) + Stall (1) = 29 cycles.

The estimated average CPI for this program is:

$$CPI_{avg} = \frac{29}{20} = 1.45$$

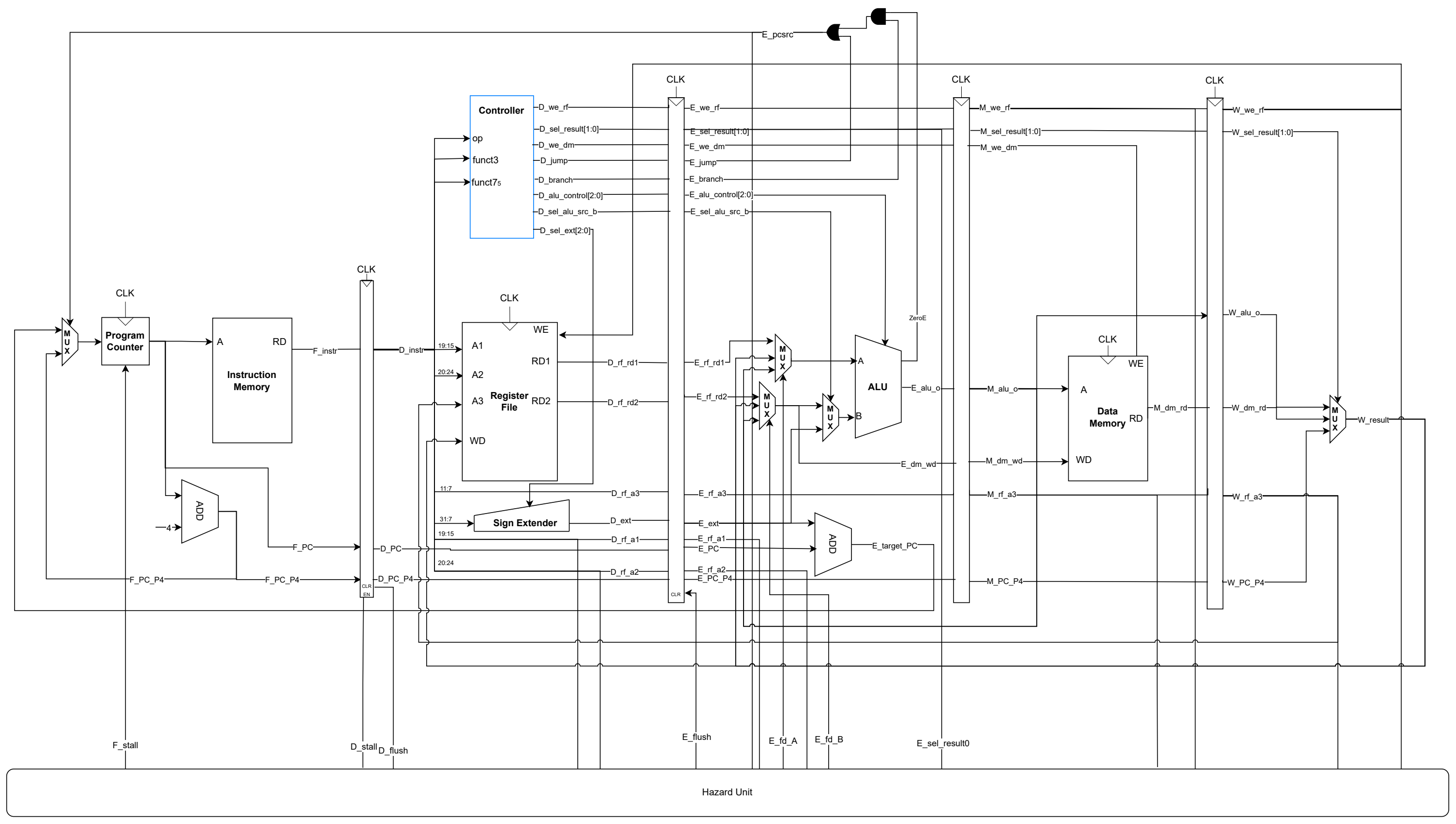
1.2.2 Pipelined Processor Performance Comparison

Based on the component delays provided in the lab manual:

- **Single-Cycle Critical Path (T_{sc}):** $T_{clk-Q} + T_{MemRead} + T_{Dec} + T_{RegRead} + T_{ALU} + T_{MemRead} + T_{Mux} = 710$ ps.
- **Multi-Cycle Critical Path (T_{mc}):** Bounded by the Memory Read stage: $T_{clk-Q} + T_{MemRead} + T_{Mux} = 270$ ps.
- **Pipelined Critical Path (T_{pl}):** Bounded by the slowest stage plus register overhead: 270 ps.

Conclusion: The maximum frequency of the pipelined version is not five times higher than the single-cycle version. This limitation exists because the cycle time is determined by the bottleneck stage (Memory Read) and the inescapable overhead introduced by the setup and propagation delays (T_{clk-Q}) of the pipeline registers.

Source Code Repository:
https://github.com/SinYita/riscv-single/tree/RV_PL



E_pcsrc
CLK
Controller
op
funct3
funct7₅
D_we_rf
D_sel_result[1:0]
D_we_dm
D_jump
D_branch
D_alu_control[2:0]
D_sel_alu_src_b
D_sel_ext[2:0]
E_we_rf
E_sel_result[1:0]
E_we_dm
E_jump
E_branch
E_alu_control[2:0]
E_sel_alu_src_b
M_we_rf
M_sel_result[1:0]
M_we_dm
W_we_rf
W_sel_result[1:0]
MUX
Program Counter
A
RD
Instruction Memory
F_instr
D_instr
19:15
20:24
11:7
31:7
A1
A2
A3
WD
WE
Register File
RD1
RD2
D_rf_rd1
D_rf_rd2
E_rf_rd1
E_rf_rd2
ZeroE
ALU
B
E_alu_o
M_alu_o
W_alu_o
Data Memory
M_dm_rd
W_dm_rd
W_result
E_dm_wd
M_dm_wd
D_rf_a3
E_rf_a3
M_rf_a3
W_rf_a3
Sign Extender
D_ext
E_ext
ADD
4
F_PC
D_PC
E_PC
D_rf_a1
E_rf_a1
D_rf_a2
E_rf_a2
E_target_PC
F_PC_P4
D_PC_P4
E_PC_P4
M_PC_P4
W_PC_P4
CLR
EN
F_stall
D_stall
D_flush
E_flush
E_fd_A
E_fd_B
E_sel_result0
Hazard Unit