Processor Architecture - Report

Quentin Meneghini, Lorenzo Pattaro Zonta

January 2026

1 Load Instructions

1.1 Fetch Stage

Load Instruction is fetched from memory using the current program counter (pc). The Instruction Memory Engine (IME) loads the instruction at address `i_pipe_D.pc`. The fetched instruction and its program counter are passed to the next stage via the `PIPE_D` pipeline register.

1.2 Decode Stage

The instruction is decoded by the `ide` module. Source register indices ("rs1", "rs2"), immediate value ("imm"), and control signals (including "`is_load`" in "`mem_control`") are extracted. The register file ("REGISTERS") outputs the value of "`rs1`" (base address for the load). All relevant data is packed into the "`i_pipe_A`" structure and passed to the next stage via "`PIPE_A`".

1.3 Address Calculation / Execute Stage

The ALU computes the effective address: `rs1 + imm`. The result is stored in `alu_result` and passed as "`execution_result`" to the next stage. The comparator and MDU are not used for loads, but bypass logic ensures the correct value for "`rs1`" if there are data hazards.

1.4 Memory Stage

The memory control signals `mem_control.is_load` indicate a load operation. The Data Memory ("DME") uses the computed address (`execution_result` to initiate a memory read. The value loaded from memory is placed in `i_pipe_W.mem_data` for the next stage.

1.5 Write Back

The pipeline register "`PIPE_W`" passes the loaded data and control signals. The "`mux_result`" in '`i_pipe_WC`' selects between the loaded data ("`mem_data`") and the ALU result, but for

loads, it chooses "mem_data". The register file writes the loaded value to the destination register ("rd"), as indicated by "wb_control".

2 Store Instructions

2.1 Fetch (F)

Store instruction is fetched from memory at the current pc using the Instruction Memory Engine (IME). The fetched instruction and its pc are passed to the next stage via PIPE_D.

2.2 Decode (D)

The instruction is decoded by the IDE module. Source register indices (rs1, rs2), immediate value (imm), and control signals (including is_store in mem_control) are extracted. The register file outputs the values of rs1 (base address) and rs2 (data to store). All relevant data is packed into i_pipe_A and passed to the next stage via PIPE_A.

2.3 Address Calculation / Execute (A)

The ALU computes the effective address: rs1 + imm. The result is stored in alu_result and passed as execution_result to the next stage. Bypass logic ensures correct values for rs1 and rs2 if there are data hazards.

2.4 Memory Access (M)

The memory control signals (mem_control.is_store) indicate a store operation. The Data Memory Engine (DME) uses the computed address (execution_result) and the value to store (rs2) to initiate a memory write. The store data is sent to memory via the arbiter and memory modules.

2.5 Write Back (W)

For store instructions, there is **no register write-back**. The pipeline passes through, but wb_control.is_write_back is not asserted. The store operation is complete once the memory acknowledges the write.

3 Branch/Jump Instructions

3.1 Fetch (F)

The branch/jump instruction is fetched from memory at the current pc using the Instruction Memory Engine (IME). The fetched instruction and its pc are passed to the next stage via PIPE_D.

3.2 Decode (D)

The instruction is decoded by the IDE module. Source register indices (rs1, rs2), immediate value (imm), and control signals (such as is_branch, is_jal, is_jalr) are extracted. The register file outputs the values of rs1 and rs2 for branch comparison or jump address calculation. All relevant data is packed into i_pipe_A and passed to the next stage via PIPE_A.

3.3 Address Calculation / Execute (A)

For branch instructions, the comparator checks the branch condition using rs1 and rs2. For jump instructions, the ALU computes the target address (pc + imm for jal, rs1 + imm for jalr). The result is stored in alu_result and control signals indicate if a branch/jump should be taken.

3.4 Memory Access (M)

Branch/jump instructions typically do not access data memory. The pipeline ensures correct control flow by flushing or updating pipeline registers if a branch/jump is taken.

3.5 Write Back (W)

For jal and jalr, the return address (pc + 1) is written to the destination register (rd). For branch instructions, there is no register write-back.

4 Arithmetic Instructions

4.1 Fetch (F)

The arithmetic instruction is fetched from memory at the current pc using the Instruction Memory Engine (IME). The fetched instruction and its pc are passed to the next stage via PIPE_D.

4.2 Decode (D)

The instruction is decoded by the IDE module. Source register indices (rs1, rs2), destination register (rd), immediate value (imm), and control signals are extracted. The register file outputs the values of rs1 and rs2 (or imm for immediate instructions). All relevant data is packed into i_pipe_A and passed to the next stage via PIPE_A.

4.3 Address Calculation / Execute (A)

The ALU performs the arithmetic operation (e.g., add, sub, and, etc.) using the values from rs1 and rs2 or imm. The result is stored in alu_result and passed as execution_result to the next stage.

4.4 Memory Access (M)

Arithmetic instructions do not access data memory. The pipeline simply forwards the `execution_result` and control signals to the next stage.

4.5 Write Back (W)

The result of the arithmetic operation (`execution_result`) is written to the destination register (`rd`) in the register file, as indicated by `wb_control.is_write_back`.

5 Exceptions

The pipeline stalls either when the memory is full, the data memory (translation lookaside buffer or cache) has a miss in M stage, the store buffer is full in M stage, the Multiply Divide Unit (MDU) is still working, or there is a TLB or cache miss in F stage.

Moreover the system detect load-use hazards, that happen when an instruction that loads data from memory is immediately followed by an instruction that uses the loaded data as a source operand. Since the data from memory may not be available yet, the pipeline must stall to avoid using incorrect or old data. The signal `stall_load_bypass` is on when the current instruction in the A (execute) stage is a load and the destination register of the load instruction is equal to at least one between the source registers of the instruction in the decode stage, so when the instruction in the decode stage needs the value to being loaded.

For stage F and D, the pipes are enabled when neither `stall_pipeline` nor `stall_load_bypass` are turned on, and for the remaining stages only `stall_pipeline` needs to be off.

Lastly, while for stages M, W and WC there is no flush for the respective pipes, for stage D and A the flush happen when there is a branch misprediction or there is a `jalr` instruction or we have the `stall_load_bypass` equal to one or a bad instruction has been detected.

6 Performance Results

we tested the processor performance with Buffer_sum program. The assembly is shown in listing below. In particular we pre-loaded the array of random values in the main memory at a known address. the instruction are in reverse order because of how the processor read bytes, but the line numbering is correct. A screenshot of the rtl is shown in Figure 1.

```
// 0C: lw x4, 0(x2)
// 08: addi x3, x2, 128
// 04: addi x2, x0, 160
// 00: addi x1, x0, 0
dut.MEM.mem[0] = {32'h00012203,32'h08010193, 32'h0A000113,32'h00000093};

// 1C: sw x1, 128(x2)
// 18: bne x2, x3, -12
// 14: addi x2, x2, 4
// 10: add x1, x1, x4
dut.MEM.mem[1] = {32'h08112023,32'hFE311AE3, 32'h00410113,32'h004080B3};
```

This snippet takes 322 cycles to be completed.

6.1 Loop Metrics

Loop bounds

$$x_2^{start} = 160, \quad x_3 = 288, \quad \text{step} = 4$$
$$N = \frac{288 - 160}{4} = 32 \text{ iterations}$$

Dynamic instruction count

- Setup: 3 instructions
- Loop body: $4 \times 32 = 128$ instructions
- Epilogue: 1 instruction

$$I_{tot} = 3 + 128 + 1 = 132$$

Memory operations

$$\text{Loads} = 32, \quad \text{Stores} = 1, \quad \text{Total} = 33$$
$$\text{Memory fraction} = \frac{33}{132} = 25\%$$

Branch behavior

$$\text{Branches} = 32, \quad \text{Taken} = 31, \quad \text{Not taken} = 1$$
$$\text{Taken rate} = \frac{31}{32} \approx 96.9\%$$

Performance (measured)

$$C_{tot} = 322 \text{ cycles}$$

$$CPI = \frac{322}{132} \approx 2.44$$

$$C_{\text{iter}} = \frac{322}{32} \approx 10.1 \text{ cycles/iteration}$$

Ideal baseline

$$\text{Ideal } C_{\text{iter}} \approx 4 \Rightarrow \text{Overhead} \approx 6 \text{ cycles/iteration}$$

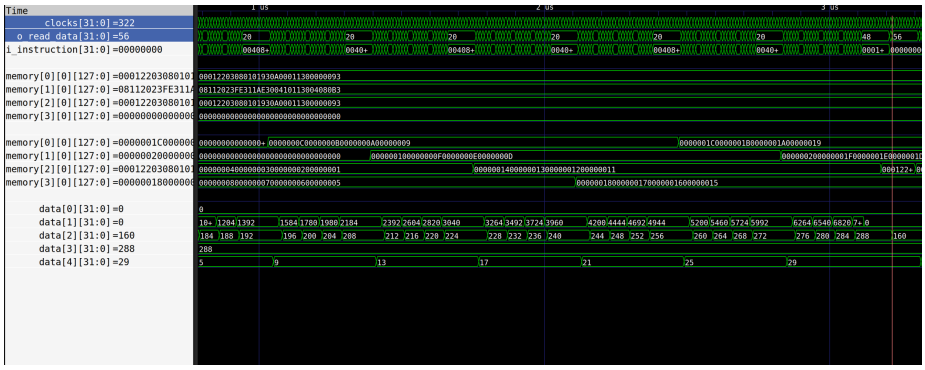

Figure 1: RTL of Buffer_sum test