DESIGN AND IMPLEMENTATION OF A 2-WAY IN-ORDER SUPERSCALAR PROCESSOR

Author

Chinmay Kulkarni

Electrical engineering

3rd Year UG @IIT Gandhinagar

23110179@iitgn.ac.in

Table of Contents

1. Introduction

This document describes the design and implementation of a **32-bit, 2-way in-order superscalar processor** developed as an educational yet industry-aligned baseline microarchitecture. The processor can fetch, decode, issue, execute, and commit up to 2 instructions per cycle while preserving strict in-order semantics.

The design intentionally avoids aggressive optimisations (out-of-order execution, speculation, prediction) in favour of **clarity, correctness, and extensibility**. It serves as a solid foundation for further research and enhancement.

2. Architectural Overview

Key Features

- 32-bit datapath
- In-order issue and commit
- Dual-instruction fetch per cycle
- Two parallel execution pipelines
- Centralised hazard detection and control
- Single architectural register file

Pipeline Organization

The processor uses a **5-stage pipeline** shared across two issue slots:

1. IF – Instruction Fetch
2. ID/ISSUE – Decode and Issue
3. EX – Execute
4. MEM – Memory Access
5. WB – Write Back

Two non-overlapping clocks (clk1, clk2) are used to simplify pipeline latch timing and avoid write-read conflicts.

3. Instruction Set Architecture (ISA)

ISA Overview

The ISA is a **custom RISC-style ISA** inspired by MIPS/RISC-V, designed specifically to support superscalar in-order execution. All instructions are **32 bits wide** and aligned on word boundaries.

The ISA is divided into the following instruction classes: - Integer ALU (register-register), Integer ALU (register-immediate), Memory access (load/store), Control flow (branch and jump), No-operation (NOP)

Instruction	Type	Opcode (6 bits)	What it does
ADD	R	000000	Adds two registers: rd = rs1 + rs2
SUB	R	000001	Subtracts two registers: rd = rs1 - rs2
MUL	R	000010	Multiply: rd = rs1 * rs2
AND	R	000011	Bitwise AND: rd = rs1 & rs2
OR	R	000100	Bitwise OR: ` rd = rs1
XOR	R	000101	Bitwise XOR: rd = rs1 ^ rs2
SLL	R	000110	Shift left logical: rd = rs1 << shamt
SRL	R	000111	Shift right logical: rd = rs1 >> shamt
ADDI	I	001000	Add immediate: rd = rs1 + imm
SUBI	I	001001	Sub immediate rd=rs1-imm
ANDI	I	001010	Bitwise AND immediate: rd = rs1 & imm
ORI	I	001011	Bitwise OR immediate: ` rd = rs1
XORI	I	001100	Bitwise XOR immediate: rd = rs1 ^ imm
LW	MEM	010000	Load word: rt = MEM[base + offset]
SW	MEM	010001	Store word: MEM[base + offset] = rt
BEQ	B	011000	Branch if equal: if rs1 == rs2, jump to PC + offset
BNE	B	011001	Branch if not equal: if rs1 != rs2, jump to PC + offset
BLT	B	011010	Branch if less than (signed)
BGE	B	011011	Branch if greater or equal (signed)
J	J	100000	Jump to absolute/PC-relative address

NOP	—	111111	No operation

4. Instruction Encoding and Formats

All instructions share a **6-bit opcode field** located at bits [31:26]. The remaining bits are interpreted according to the instruction type.

4.1 R-Type (Register-Register ALU Instructions)

Format:

| 31:26 | 25:21 | 20:16 | 15:11 | 10:0 |

|opcode | rs1 | rs2 | rd |unused|

Description: - rs1, rs2: Source registers - rd: Destination register - Used for pure register-based ALU operations

Examples: - ADD rd, rs1, rs2 - SUB rd, rs1, rs2 - AND rd, rs1, rs2

The ALU result is written back to rd during the WB stage.

4.2 I-Type (Register-Immediate ALU Instructions)

Format:

| 31:26 | 25:21 | 20:16 | 15:0 |

|opcode | rs | rd | imm |

Description: - rs: Source register - rd: Destination register - imm: 16-bit signed immediate (sign-extended)

Examples: - ADDI rd, rs, imm - ANDI rd, rs, imm

The immediate value is sign-extended to 32 bits before execution.

4.3 MEM-Type (Load / Store Instructions)

Format:

| 31:26 | 25:21 | 20:16 | 15:0 |

|opcode | base | rt |offset|

Description: - base: Base address register - rt: Load destination or store source - offset: Signed byte/word offset

Examples: - LW rt, offset(base) - SW rt, offset(base)

Operation: - Effective address = base + sign-extended offset - Load writes to register in WB stage - Store writes to memory in MEM stage

4.4 B-Type (Conditional Branch Instructions)

Format:

| 31:26 | 25:21 | 20:16 | 15:0 |

|opcode | rs1 | rs2 |offset|

Description: - Two registers are compared - Branch target is PC-relative

Examples: - BEQ rs1, rs2, offset - BNE rs1, rs2, offset - BLT rs1, rs2, offset

The branch decision is resolved in the EX stage.

4.5 J-Type (Jump Instructions)

Format:

| 31:26 | 25:0 |

|opcode |target|

Example: - J target

Operation: - J: PC is updated with the target All jump instructions cause pipeline flushing.

5. Register File Organisation

Architectural Registers

- 32 general-purpose registers (R0–R31)
- Each register is 32 bits wide
- R0 is hardwired to zero

Access Characteristics

- Dual read ports (for two issued instructions)
- Dual write ports (one per pipeline)
- Writes occur in the WB stage

Register Usage by Instruction Type

Instruction Type	Source Registers	Destination Register
R-Type ALU	rs1, rs2	rd
I-Type ALU	rs	rd

Load (LW)	base	rt
Store (SW)	base, rt	none
Branch	rs1, rs2	none

The decode logic extracts register indices directly from instruction fields and accesses the register file or forwarded values accordingly.

6. Instruction Storage and Flow Through Pipeline

- Instructions are stored in a word-addressed instruction memory array
- Two consecutive instructions are fetched per cycle
- Instructions retain their original binary encoding throughout the pipeline
- Field extraction (opcode, registers, immediate) is done in the ID stage

No instruction rewriting or micro-op decomposition is used.

7. Detailed Pipeline Stage Description

7.1 Instruction Fetch (IF)

The Instruction Fetch stage is responsible for supplying up to **two instructions per cycle** to the pipeline. The Program Counter (PC) is word-addressed and increments by two on each successful fetch, enabling dual-issue capability. Two instruction registers, IR0 and IR1, latch consecutive instructions from instruction memory.

Control behaviour in IF: - On normal execution: PC ← PC + 2 - On stall: PC and IF/ID registers are held constant

On branch or jump taken: PC is redirected to the computed target and younger instructions are invalidated

The IF stage itself is kept simple; no branch prediction or speculative fetch is implemented.

7.2 Instruction Decode and Issue (ID / ISSUE)

The ID stage is the **most critical and complex stage** of the processor. It performs instruction decoding, register operand fetch, functional unit classification, hazard checks, and issue arbitration.

7.2.1 Instruction Decode

Each fetched instruction is decoded combinationally. The opcode field [31:26] determines: - Instruction class (ALU, MEM, BRANCH/JUMP) - Subtype (RR, RI, LOAD, STORE, etc.)

From the instruction word, the following fields are extracted: - Source register indices (rs1, rs2) - Destination register index (rd or rt) - Immediate field (sign-extended for I-type and MEM-type)

No micro-op translation is performed; the original instruction encoding is preserved.

7.2.2 Register File Access

The register file provides: - Two read ports per issued instruction - Up to two write ports in the WB stage

Operands may be sourced directly from the register file. In the current design, RAW hazards result in stalling rather than forwarding.

Register zero (R0) is hardwired to zero and is never modified.

7.2.3 Functional Unit Classification

Each instruction is classified into one of three functional categories: - ALUtype – Integer arithmetic and logical operations - MEMtype – Load and store operations - B_Jtype – Branch and jump operations

This classification drives the issue logic and determines which pipeline (A or B) an instruction may enter.

7.2.4 Issue Arbitration Logic

The processor supports **at most one instruction per functional class per cycle**. The following rules apply: -

Branch and jump instructions are issued only to **Pipe A** Load and store instructions are issued only to **Pipe B** Two ALU instructions may issue simultaneously (one per pipe)

If both fetched instructions compete for the same functional resource, one instruction is issued while the other is stalled or converted to a NOP. Identical instruction types targeting the same pipeline cause a controlled stall (stall_II).

The ID stage ensures that all issued instructions maintain in-order semantics.

7.2.5 RAW Hazard Handling Policy

Read-After-Write (RAW) hazards are resolved in the ID/ISSUE stage using strict in-order rules.

If a RAW dependency exists with an **older instruction already ahead in the pipeline** (EX/MEM/WB), the dependent instruction is **stalled** until the producing instruction completes write-back.

If a RAW dependency exists **between the two instructions fetched in the same cycle**, the **older instruction (Instruction 0) is issued first**, while the **younger instruction (Instruction 1) is stalled by inserting a NOP**. In the following cycle, the stalled instruction is reissued after the hazard condition is cleared.

This ensures correctness while preserving in-order semantics and deterministic superscalar issue behaviour.

7.3 Execute (EX)

The Execute stage contains two independent ALUs, one per pipeline. Depending on instruction type, the ALU performs: - Arithmetic and logical operations - Effective address calculation for memory instructions - Branch condition evaluation - Jump target computation

Branch decisions are resolved in EX. On a taken branch or jump, control signals are generated to redirect the PC and flush older instructions.

7.4 Memory Access (MEM)

The MEM stage is active only for Pipe B. Supported operations: - Load: data is read from memory and forwarded to WB - Store: data is written to memory, unless suppressed by a branch flush

Pipe A bypasses the MEM stage entirely.

7.5 Write Back (WB)

The WB stage commits results to the architectural register file: - ALU results are written for RR and RI instructions - Load data is written for LW instructions

Dual write-back is supported, with one result per pipeline. Writes are disabled during branch flush cycles to preserve correctness.

8. Hazard Detection and Handling

The processor employs a **conservative but correct hazard handling strategy**, prioritizing correctness over performance.

8.1 Data Hazards (RAW)

Read-After-Write (RAW) hazards are detected by comparing: - Destination registers of instructions in ID/EX and EX/MEM stages - Source registers of instructions in IF/ID stage
If a match is detected a hazard stall is asserted. The pipeline is stalled until the producing instruction completes write-back.
No data forwarding is implemented in the current design.

8.2 Write-After-Write (WAW)

WAW hazards are inherently avoided due to: - In-order issue - In-order write-back
When two instructions target the same destination register, the younger instruction is stalled or dropped according to issue rules.

8.3 Structural Hazards

Structural hazards arise from limited shared resources: - Only one memory operation is allowed per cycle - Only one branch/jump is allowed per cycle
Such conflicts are resolved in the ID stage through controlled stalling or NOP insertion.

8.4 Control Hazards

Control hazards occur due to branches and jumps. Handling strategy: - Branch outcome is resolved in EX stage - All younger instructions are flushed on a taken branch - Instruction fetch resumes from the correct target

9. Verification Strategy

Verification is performed using **directed, cycle-accurate testing**, focusing on functional correctness and hazard behaviour.

9.1 Test Categories

The verification plan includes: - Individual opcode validation (RR and RI types) - Load and store correctness - Back-to-back memory operations - Branch taken and not-taken scenarios - Jump and jump-and-link behaviour - RAW hazards at varying distances (1-cycle, 2-cycle) - Dual-issue and single-issue scenarios

9.2 Hazard-Focused Tests

Dedicated tests verify: - RAW hazard stalling across pipelines - Load-use hazards - Branch flush correctness - Suppression of memory writes during flush

Waveform inspection is used to ensure that control signals (stall, branched) behave as expected.

9.3 Limitations of Verification

The current verification strategy does not include: - Randomised testing - Formal verification - Performance benchmarking

However, it is sufficient to establish architectural correctness for a baseline design.

10. Conclusion

This processor represents a clean and correct implementation of a 2-way in-order superscalar architecture. The detailed ID stage logic, conservative hazard handling, and structured verification approach make it suitable as a baseline for research, teaching, and further enhancement.