# Microarchitecture Design

## Introduction

### Processor Context and Target Application Domain

This processor is designed for low-cost, AI-enabled mobile phones targeting the Lesotho and broader African market. It specializes in executing key on-device AI workloads including voice recognition, biometric security, and intelligent connectivity management under stringent constraints of cost, power efficiency, and intermittent network reliability.

### Reference ISA

The microarchitecture implements a customized RISC-V ISA with domain-specific extensions optimized for AI workloads. The ISA features:

* 32-bit base instruction format with optional 16-bit compressed instructions
* 32 general-purpose registers (32-bit each)
* RISC-style load/store architecture with fixed-length instructions
* Custom extensions for AI acceleration
* Support for digital signal processing and pattern matching operations

### Supported Instructions Classes

1. Arithmetic Operations (ADD, SUB, ADDI, ANDI, XORI)
2. Load/Store Operations (LD\_W, ST\_W, LD\_B, ST\_B)
3. Control Flow (JUMP, JMP\_REG, BR\_EQ, BR\_NEQ)
4. Shift Operations (SHIFT\_L, SHIFT\_R)
5. Custom AI Acceleration

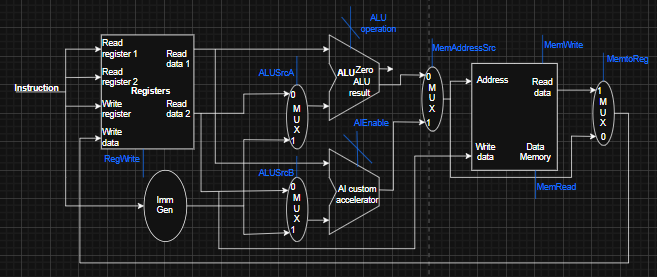
### Design Objectives

1. Minimize Power Consumption: Optimize energy efficiency to extend battery life in mobile devices, targeting operation below 100mW during typical usage.
2. Low CPI (Cycles Per Instruction**)**: Achieve near-1.0 CPI through efficient pipelining and minimal pipeline stalls, with specialized execution units for common AI operations.
3. Simple Pipeline Implementation: Employ a straightforward 5-stage pipeline (IF-ID-EX-MEM-WB) with minimal complexity to reduce hardware costs and improve reliability.
4. Domain-Specific Acceleration: Integrate custom execution units that provide single-cycle completion for critical AI operations that would otherwise require multiple instructions.
5. Cost-Effective Design: Minimize silicon area through simplified control logic and shared resources while maintaining performance for target applications.
6. Real-Time Responsiveness: Ensure low-latency execution for user-facing tasks like voice authentication (<100ms response time) through optimized memory hierarchy and execution units.

## Incremental Datapath Design

### R-Type Instruction Datapath

Block Diagram:



Data Flows and Control Signals:

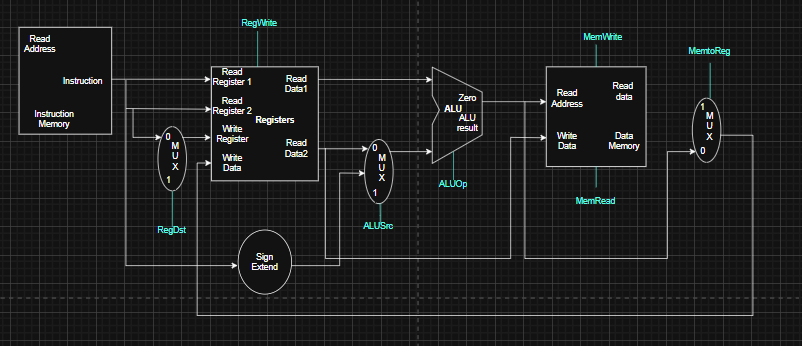
* Instruction [31:0]: Provides register addresses (Rsrc1, Rsrc2, Rdest) and function codes
* ALU Control: Funct3[2:0] + Funct7[5] → ALU operation (ADD, SUB, AND, OR, XOR, etc.)
* Register Read: Two 32-bit values from Register File
* ALU Result: 32-bit computation result
* Register Write: Result written back to Rdest register

Execution Explanation:

R-type instructions perform register-to-register operations. The instruction is fetched from memory and decoded to extract the source register addresses (Rsrc1, Rsrc2) and destination register (Rdest). The register file reads both source operands simultaneously while the control unit generates the appropriate ALU operation based on the Funct3 and Funct7 fields. The ALU computes the result, which is then written back to the destination register during the write-back stage. This format is used for arithmetic (ADD, SUB), logical (AND, OR), and shift operations.

### I-Type Instruction Datapath

Block Diagram:



**Data Flows and Control Signals:**

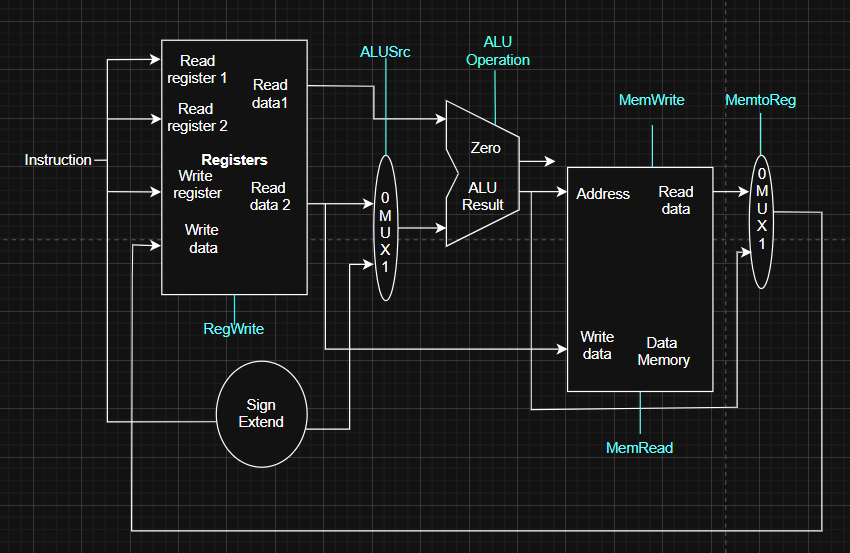
* **Immediate Generation**: Sign-extends 12-bit immediate from instruction[31:20]
* **ALU Control**: Funct3 determines operation (ADDI, ANDI, XORI, etc.)
* **Memory Control**: MemRead enabled for load operations, MemWrite for stores
* **Load Path**: Data memory output → write back to register

Execution Explanation:

I-type instructions handle operations with immediate values and load operations. The 12-bit immediate field is sign-extended to 32 bits, while the register file provides one source operand. For immediate arithmetic (ADDI, ANDI), the ALU combines the register value with the immediate. For load operations (LD\_W, LD\_B), the ALU computes the memory address by adding the base register to the offset, then reads from data memory, and the result is written back to the destination register. The control unit distinguishes between immediate operations and memory accesses using the opcode.

### S-Type Instruction Datapath

Block Diagram:



**Data Flows and Control Signals:**

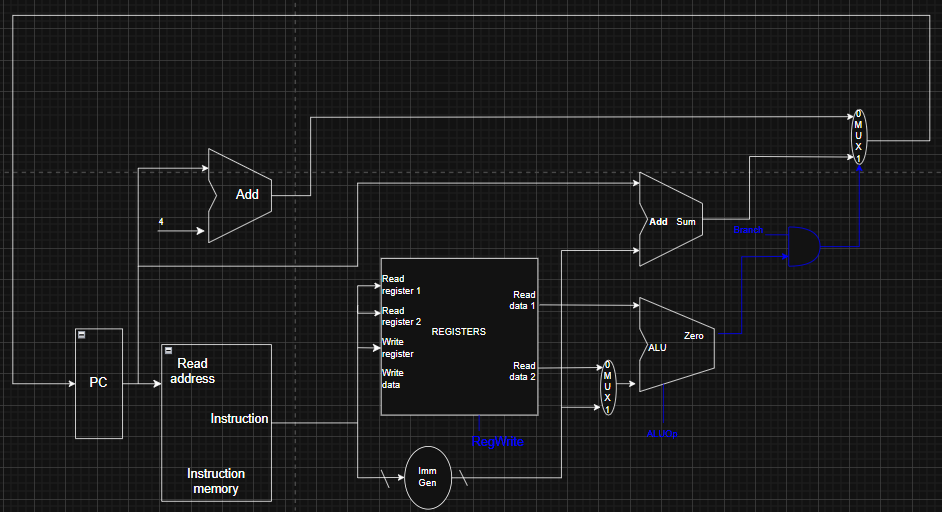
* **Immediate Generation**: Combines instruction[31:25] and [11:7] for store offset
* **ALU Operation**: Always addition for address calculation
* **Memory Control**: MemWrite enabled, MemRead disabled
* **Data Path**: Register Rsrc2 value → data memory input

Execution Explanation:

S-type instructions handle store operations to memory. The instruction contains two register operands - a base address register and a data register to be stored. The immediate generator combines fields from different parts of the instruction to form the 12-bit offset, which is sign-extended. The ALU adds the base register value to this offset to compute the memory address. The data from the second source register is then written to this computed address in the data memory. Store operations do not write back to the register file.

### B-Type Instruction Datapath

Block Diagram:



**Data Flows and Control Signals:**

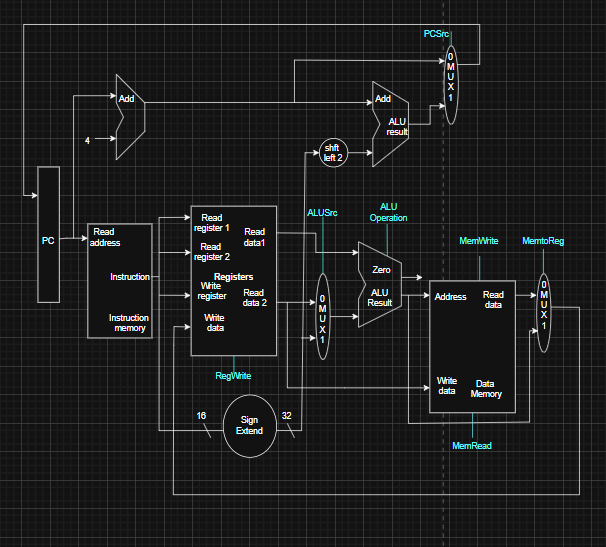
* **Branch Immediate**: Sign-extended offset for PC-relative branching
* **Comparator Control**: Funct3 determines branch condition (BEQ, BNE, etc.)
* **PC Update**: Selects between PC+4 (no branch) or PC+offset (branch taken)
* **Branch Taken**: Signal from comparator to PC update logic

Execution Explanation:

B-type instructions handle conditional branching based on register comparisons. Both source registers are read from the register file and compared in the branch comparator. The immediate generator creates a sign-extended branch offset from the B-format immediate fields. Based on the branch condition (BEQ, BNE, etc.) specified by Funct3 and the comparison result, the PC update logic either continues sequential execution (PC+4) or takes the branch (PC+offset). This enables efficient control flow for loops and conditional execution in the target AI applications.

### J-Type Instruction Datapath

Block Diagram:



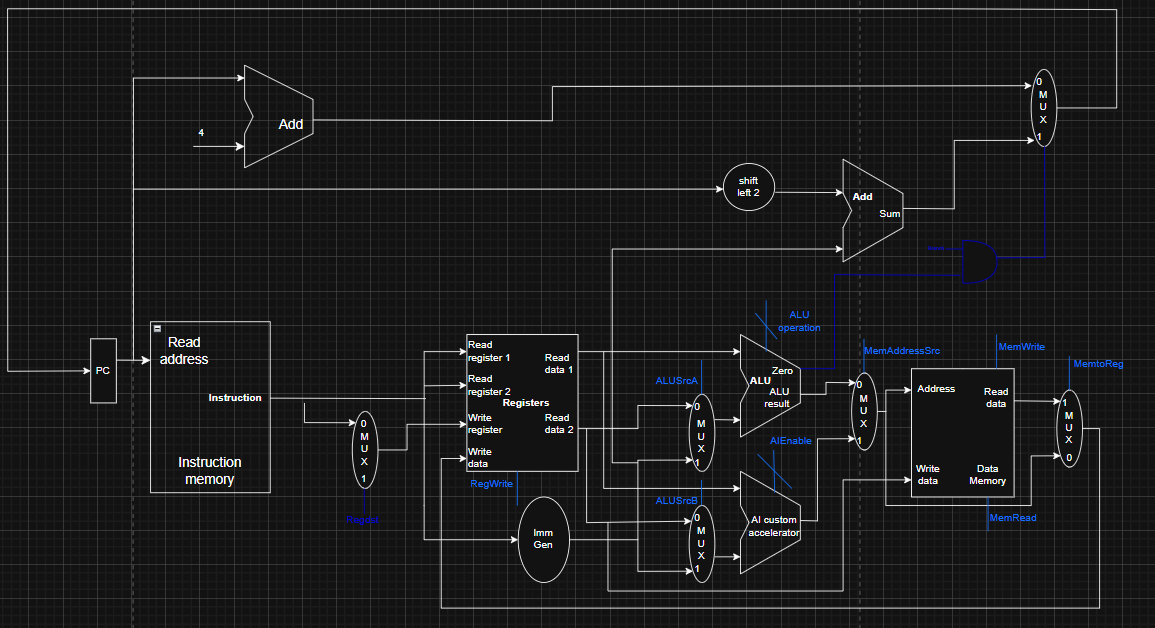
**Data Flows and Control Signals:**

* **Jump Immediate**: 20-bit immediate from instruction[31:12] sign-extended and shifted left by 1 bit
* **PC Source Select**: Control signal chooses between sequential execution (PC+4) and jump target
* **Jump Enable**: Opcode-based signal that forces PC update to jump target
* **Immediate Calculation**: jump\_target = PC + (sign\_extend(imm20) << 2)

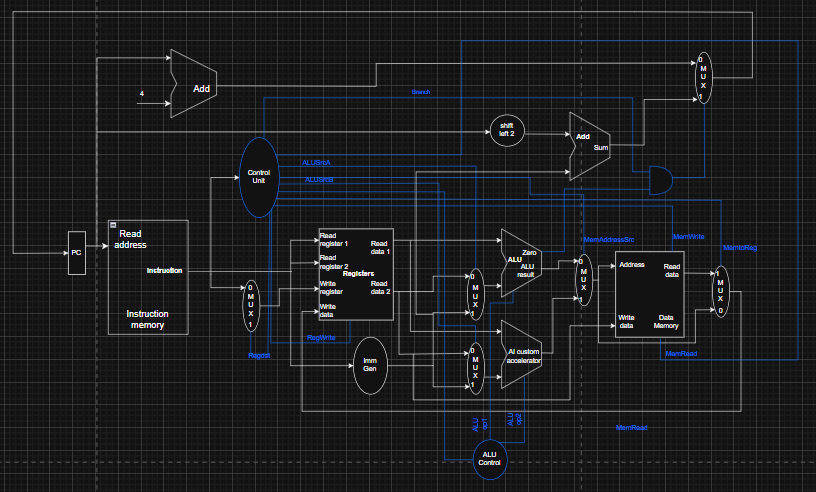
Execution Explanation:

J-type instructions handle unconditional jump operations to PC-relative addresses. The 20-bit immediate field from the instruction is sign-extended to 32 bits and then left-shifted by 1 bit (since instructions are 2-byte aligned in the compressed ISA variant). This offset is added to the current PC value to compute the jump target address. Unlike branch instructions, J-type jumps are always taken and do not require register comparisons. The control unit generates a Jump Enable signal based on the opcode, forcing the PC update logic to select the computed jump target rather than the sequential PC+4. This format is essential for implementing function calls, long-range jumps, and exception handling in the processor.

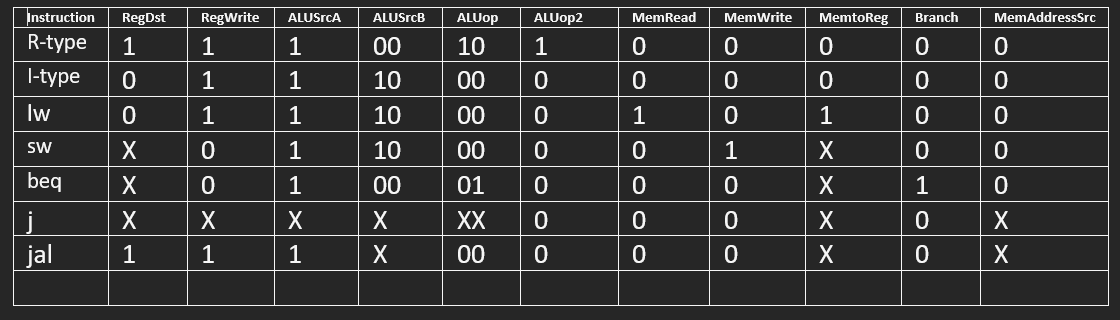
### Unified Single-Cycle Datapath



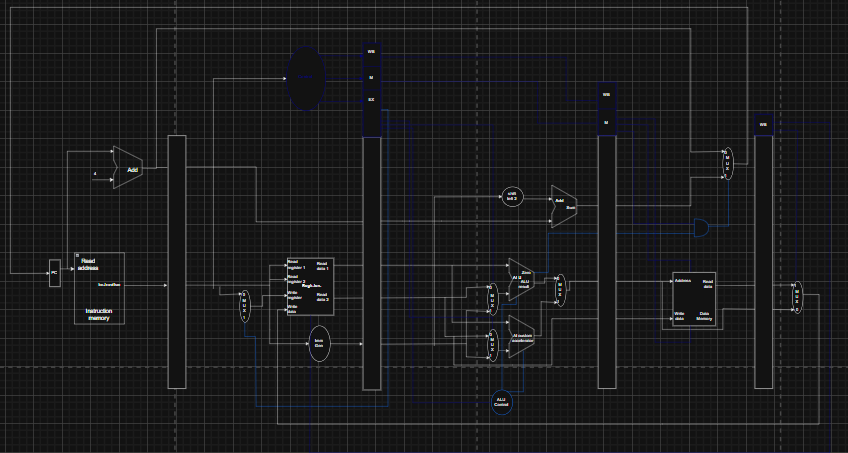
### Control Unit Design



### Truth Table



### Pipelined Implementation



### Pipeline Hazards and Optimizations

1. Data Hazards (Structural, Read-After-Write)

A. Forwarding

This is the primary optimization technique used to resolve most RAW hazards without stalling.

* Mechanism: Instead of waiting for the data to be written back to the Register File (WB stage), the result is forwarded (or bypassed) directly from the output of an earlier stage (like the EX or MEM stage) back to the input of a later stage (the ALU inputs in the EX stage).
* Implementation: Requires a Forwarding Unit in the EX stage and two dedicated Forwarding MUXs at the ALU inputs.
  + Forwarding Paths:
    1. EX/MEM Register → EX Stage ALU Input: For results calculated in the current instruction's EX stage (e.g., an R-type instruction) that are needed by the next instruction's EX stage.
    2. MEM/WB Register → EX Stage ALU Input: For results calculated two instructions ago (e.g., an R-type instruction two cycles prior) or for the result of a load instruction that has just completed the memory access.

B. Stalling (Pipeline Interlock)

Forwarding cannot resolve all data hazards, specifically the Load-Use Hazard.

* Hazard Type: A load instruction (lw) reads data in the MEM stage. If the very next instruction immediately tries to use that data in the EX stage, forwarding from MEM/WB is too late. The data is not available until the MEM stage completes.
* Mechanism: A Hazard Detection Unit (placed in the ID stage) detects this specific dependency.

It stalls the pipeline by:

Disabling the write to the PC and the IF/ID register (stopping the fetching of new instructions).

Inserting a NOP (No-Operation) bubble into the ID/EX register. This is done by zeroing out all the control signals for the instruction currently in the ID/EX stage.

* Result: The dependent instruction is delayed by one cycle, creating a "bubble," allowing the load instruction to complete its MEM access and providing the data to the EX stage via forwarding in the next cycle.

2. Control Hazards (Branch Hazards)

A control hazard occurs when a branch or jump instruction is executed, and the pipeline has already fetched one or more instructions from the sequential path, which now might be incorrect.

A. Simple Optimization: Branch Prediction with Stall

The most basic and essential mechanism is to always predict the branch is Not Taken.

* Mechanism:
  1. Continue fetching the next sequential instruction (PC + 4).
  2. The branch condition is calculated in the EX stage.
  3. If the branch is determined to be Taken in the EX stage, the instruction(s) in the ID and IF stages are flushed (turned into NOPs by clearing the IF/ID and ID/EX registers), and the PC is updated with the branch target address.
* Cost: This simple approach results in a 2-cycle stall every time a branch is taken.

B. Optimization: Move Branch Decision to ID Stage (Proposed)

To reduce the branch penalty, the decision (and calculation of the branch target address) should be moved earlier in the pipeline.

* Proposal: Move the branch comparison logic (zero-detection for beq/bne) from the EX stage back to the ID stage. The branch target address is already calculated in ID.
* Benefit: The branch decision is known after the ID stage, reducing the penalty for a mispredicted branch from 2 cycles to 1 cycle. Requires an additional comparator and minor register access logic in the ID stage.

3. Pipeline Optimizations (Beyond Hazards)

A. Optimizing Register Write-Back

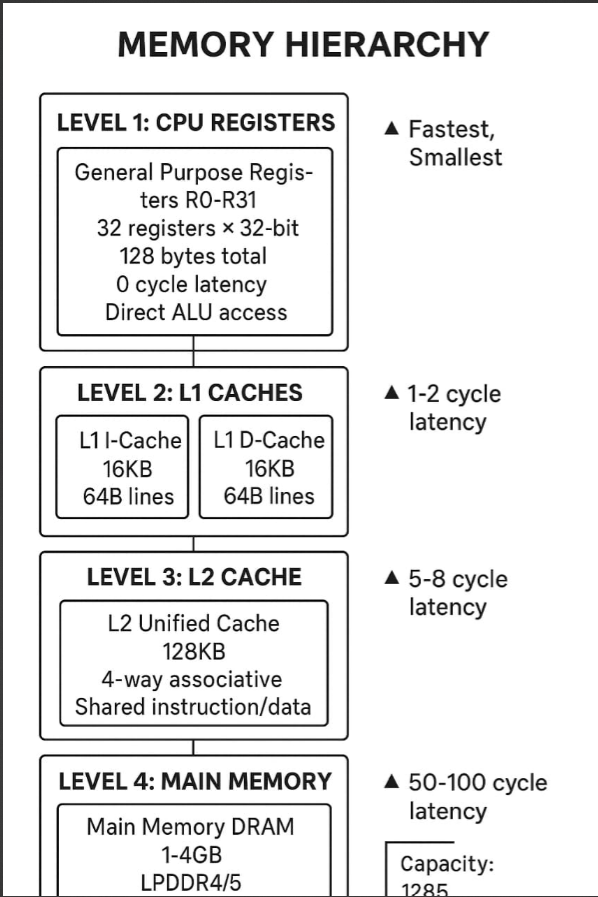
* Proposal: The Register File is typically written in the first half of the clock cycle and read in the second half. This allows the instruction in the WB stage to write to the register file and the instruction in the ID stage to read the *new* value in the same clock cycle. This eliminates an additional potential 1-cycle stall for RAW hazards that complete in the WB stage.

B. Branch Delay Slot (Compiler Optimization)

* Mechanism: A compiler optimization where the instruction immediately following a branch instruction (the branch delay slot) is always executed, regardless of whether the branch is taken or not.
* Benefit: Converts a hardware stall cycle into a useful work cycle by placing an independent, non-dependent instruction from either before the branch or the branch target into the delay slot. This hides the branch penalty entirely.

### Memory Hierarchy Design

The memory hierarchy is designed to balance performance, power efficiency, and cost for AI-enabled mobile devices in the Lesotho market. The design prioritizes energy efficiency and real-time responsiveness for voice recognition, biometric security, and connectivity management workloads.



### Conclusion

Designing this processor revealed that true innovation lies not in raw performance, but in thoughtful constraint-driven design. By focusing exclusively on the specific needs of African mobile users—voice interfaces for low literacy, biometric security for financial inclusion, and intelligent connectivity for unreliable networks—we created an architecture where every transistor serves a purpose. The most challenging aspect was balancing the simplicity needed for low-cost manufacturing with the specialized capabilities required for AI workloads. This experience underscored that effective engineering for emerging markets isn't about stripping down existing technology but about building up fundamental local needs. The resulting design proves that limitations can breed creativity, and that targeted solutions often outperform generic ones when context matters most.

### References

1. CS3520 – Computer Organization & Architecture I, Processor Design & Implementation.
2. CS3520 – Computer Organization & Architecture I, Instruction Set Architecture
3. RISC-V International (2024). RISC-V Specification Documents. [online] Available at: <https://riscv.org/technical/specifications/> .
4. Waterman, A., & Asanović, K. (2019). The RISC-V Instruction Set Manual, Volume I: User-Level ISA. RISC-V Foundation.
5. Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design: The Hardware/Software Interface. RISC-V Edition. Morgan Kaufmann.