UNIVERSITATEA POLITEHNICA TIMIȘOARA
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
SPECIALIZAREA: CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
SECȚIA: CALCULATOARE

INGINERIA ŞI TESTAREA SISTEMELOR DE CALCUL

# RISC-V
# Assembly Simulator

*Project Documentation*

Golya Carmen-Mihaela
Grupa 2.1
Anul 3

Timişoara, 2025

# Table of Contents

# I. Project Overview.

This project is an educational RISC-V Assembly Simulator written in the C programming language. It was created as part of a university course and focuses on demonstrating the internal steps through which a simplified RISC-V processor handles instructions, starting from raw assembly code and ending with the final execution results. The simulator aims to make the inner workings of a processor visible, showing how instructions are parsed, encoded, fetched, decoded, executed, and how their effects propagate through registers and memory.

The simulator supports a core subset of RISC-V operations that are sufficient for understanding the fundamentals of the architecture. Arithmetic and logical computations are implemented through R-type instructions such as ADD, SUB, MUL, and DIV. Immediate and control-flow instructions are represented by I-type operations like LW and JALR. Memory-storage functionality is included through the S-type instruction SW, while program control and jumping are illustrated through JAL, encoded in the standard RISC-V jump format. These operations together allow programs to perform calculations, interact with memory, and change execution flow, forming a minimal but coherent instruction set.

The program begins with the assembler component, which receives an input file containing RISC-V assembly code. This stage processes the code line by line, removes comments and unnecessary spacing, records labels and associates them with their correct addresses, and checks the structural correctness of each instruction. Once the input is cleaned and understood, the assembler transforms every instruction into a structured internal representation. This representation contains all information the simulator will need later: the operation type, involved registers, immediate values, and resolved label destinations. This preparation ensures that instructions are fully defined before execution begins.

After this construction phase, the simulator performs instruction encoding. In this step, each instruction is converted into its machine-code-like 32-bit binary form, respecting the layout of the RISC-V format. The opcode is placed in the appropriate bits, fields such as funct3 and funct7 are positioned correctly, registers are encoded into their designated sections, and immediate values are packed into their specific bit ranges. The result is a numerical instruction value that resembles real RISC-V machine code. These encoded instructions are stored sequentially in an internal instruction memory, forming the program that will be executed.

Once the program has been encoded, the simulator enters the processor execution phase. This phase implements a simplified version of the classical fetch–decode–execute cycle used by real CPUs. The cycle begins with the fetch stage, where the simulator reads the instruction located at the address pointed to by the program counter. After fetching, the simulator moves to decoding, a process in which the numerical instruction is interpreted and decomposed back into its fields, determining the operation type and extracting the operand registers and immediate values. With this decoded information, the simulator proceeds to the execution stage, where it carries out the action defined by the instruction. Arithmetic instructions compute new values and update the appropriate registers. Load and store operations access or modify the simulated memory region. Jump instructions modify the program counter and may save a return address depending on the operation's semantics.

Following execution, the simulator updates the global state to reflect the effects of the instruction. The program counter may advance sequentially or jump to a new address. Registers are modified according to the executed operation, and memory locations may change if the instruction required it. Once these updates are complete, the cycle repeats with the next instruction until the program has no further instructions to execute.

At the end of the simulation, the system produces an output that summarizes the final values of all registers and relevant memory sections. This output provides insight into how the program behaved and makes it possible to compare results with expected outcomes, particularly when running test cases or debugging assembly code.

Through this complete pipeline, the project offers a clear view of how assembly code is transformed step by step into actions performed by a simulated processor. It emphasizes the internal mechanisms behind instruction handling, making it a valuable learning tool for students who want to understand the operational principles of RISC-V architectures.

# II. Architecture.

## a. Supported Instruction Set.

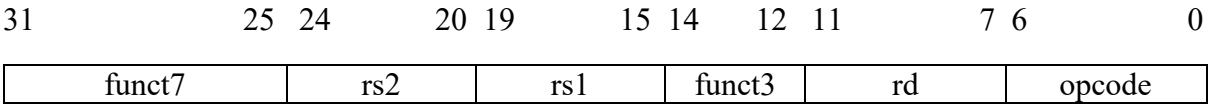
The simulator implements an educational subset of the RV32I instruction set architecture, covering arithmetic and logical operations, memory access, branching, and control transfer. Although intentionally reduced compared to the full ISA, the implemented instructions span all major RISC-V encoding formats, providing a faithful and technically representative execution environment. Each instruction is handled according to the official RISC-V specification, including operand extraction, immediate reconstruction, and control-flow semantics.

### 1. R-Type instructions:

The supported R-type instructions include ADD, SUB, MUL, and DIV, as well as XOR, OR, AND, SLL, SRL, and SRA. These operations perform register-to-register arithmetic and rely on the funct7, funct3, and opcode fields to distinguish between behaviors such as addition versus subtraction or signed multiplication versus division. Their execution updates the destination register based on ALU computations and forms the core of arithmetic processing in the simulator.

R-type instructions use only register operands and contain no immediate values. Their format places rd, rs1, rs2, and funct bits at fixed positions, allowing the decoder to identify the intended ALU operation purely based on bit patterns.

They follow the canonical RISC-V layout:



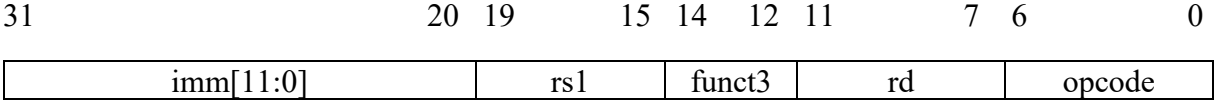

- **funct7** differentiates operations such as ADD vs. SUB, SRL vs. SRA, or MUL/DIV variants.
- **funct3** selects logical/shift/arithmetic groups.
- **opcode** is fixed as 0110011 (0x33).

All operands come directly from the register file (rs1, rs2), and the ALU writes the result into rd.

### 2. I-Type instructions:

The I-type instructions used in the simulator include LW and JALR, as well as ADDI and the pseudo-instruction LI. LW performs a memory read using a base address contained in a register and an immediate offset encoded in the instruction format. JALR provides an indirect jump mechanism by computing a new program counter from a register value and an immediate field, and by storing the return address into the destination register. These I-type operations illustrate the role of immediates and register-indexed addressing in the architecture.

I-type instructions introduce 12-bit sign-extended immediates, enabling immediate arithmetic, load addressing, and indirect jumps. The immediate occupies bits [31:20] and is extended at decode time.




- **LW** computes: address = rs1 + imm

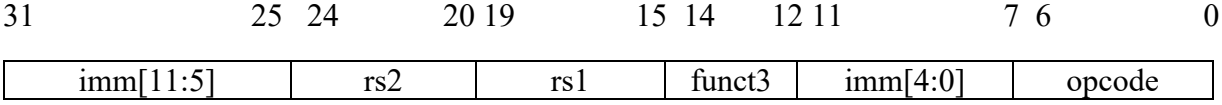
- **ADDI** computes: rd = rs1 + imm
- **LI** expands to ADDI or LUI+ADDI depending on immediate size
- **JALR** computes PC = (rs1 + imm) & ~1

This format supports register–immediate operations and enables more compact code.

3. **S-Type instructions:**

The S-type instruction SW is supported to provide store capability. It writes a register value to memory at an address formed from a base register and a split immediate field. This demonstrates how the S-type format distributes immediate bits across non-contiguous segments of the instruction, requiring careful reconstruction during decoding.

S-type instructions encode the immediate value in two parts: imm[11:5] and imm[4:0]. These must be combined, sign-extended, and added to rs1 to compute the memory address.

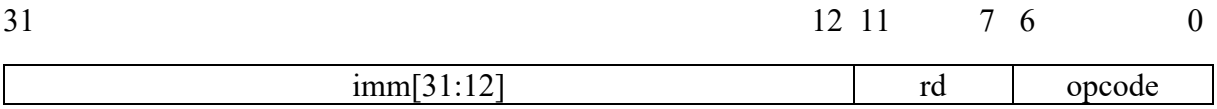



- **rs2** is the value being stored;
- **rs1 + imm** computes the target address;
- requires immediate reconstruction during decode.

4. **U-Type instructions:**

The simulator also implements the U-type instructions LUI and AUIPC. LUI loads a 20-bit immediate into the upper part of a register, while AUIPC adds such an immediate to the current program counter to support PC-relative addressing. These instructions enable efficient construction of large constants and form the basis for position-independent code.

U-type formats provide 20-bit upper immediates, enabling efficient manipulation of large values.




- **LUI**: rd = imm << 12;
- **AUIPC**: rd = PC + (imm << 12).

Both are crucial for generating 32-bit constants.

5. **B-Type instructions:**

The supported B-type instructions include BEQ, BNE, and BLT. These conditional branch instructions compare register values and update the program counter based on a sign-extended immediate offset encoded across multiple non-contiguous bit fields. They illustrate conditional control flow and the RISC-V branch encoding mechanism.

The B-type immediate is the most fragmented, requiring reconstruction from 4 separate bit regions before shifting and sign-extension.

| 31 | 30 29 | 25 24 | 20 19 | 15 14 | 12 11 | 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |

- Final immediate layout: { imm[12], imm[10:5], imm[4:1], imm[11], 0 };
- Added to PC for conditional branching.

**6. J-Type instructions:**

Finally, the simulator implements the JAL instruction. JAL performs an unconditional jump by adding a sign-extended 20-bit immediate to the current program counter, while also saving the return address in the destination register. This instruction illustrates long-range control flow and the particular bit arrangement of the J-type immediate, which must be reconstructed during the decode phase.

J-type immediates are spread across multiple fields but reconstruct into a 21-bit signed offset shifted left by 1.

| 31 | 30 21 | 20 | 19 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode |

- PC-relative jump;
- Return address saved in rd.

Together, these instructions provide a minimal but complete operational foundation for simulating computation, memory interaction, and program control within the RV32I model. Their diversity allows the simulator to cover all major stages of instruction processing, making the implementation both pedagogically valuable and technically representative of real RISC-V systems.

**Complete Set of Implemented Instructions:**

| Instruction | Format | Opcode/Funct | Notes |
|---|---|---|---|
| ADD | R-type | 0x33 (funct3=0x00, funct7=0x00) | Integer register addition |
| MUL | R-type | 0x33 (funct3=0x00, funct7=0x01) | Multiplication (M-extension subset) |
| SUB | R-type | 0x33 (funct3=0x00, funct7=0x20) | Integer subtraction |
| SLL | R-type | 0x33 (funct3=0x01, funct7=0x00) | Logical left shift |
| XOR | R-type | 0x33 (funct3=0x04, funct7=0x00) | Bitwise XOR |
| DIV | R-type | 0x33 (funct3=0x04, funct7=0x01) | Integer division |
| SRL | R-type | 0x33 (funct3=0x05, funct7=0x00) | Logical right shift |
| SRA | R-type | 0x33 (funct3=0x05, funct7=0x20) | Arithmetic right shift |
| OR | R-type | 0x33 (funct3=0x06, funct7=0x00) | Bitwise OR |
| AND | R-type | 0x33 (funct3=0x07, funct7=0x00) | Bitwise AND |
| LW | I-type | 0x03 (funct3=0x20) | Load word from memory |
| LI | I-type(Pseudo) | 0x13 (funct3=0x00, rs=0) | Expanded into ADDI |
| ADDI | I-type | 0x13 (funct3=0x00) | Add immediate |
| JALR | I-type | 0x67 (funct3=0x00) | Indirect jump |
| SW | S-type | 0x23 (funct3=0x20) | Store word to memory |
| LUI | U-type | 0x37 | Load upper immediate |
| AUIPC | U-type | 0x17 | Add upper immediate to PC |
| BEQ | B-type | 0x63 (funct3=0x00) | Branch if equal |

| | | | |
|---|---|---|---|
| BNE | B-type | 0x63 (funct3=0x10) | Branch if not equal |
| BLT | B-type | 0x63 (funct3=0x40) | Branch if less than |
| BGE | B-type | 0x63 (funct3=0x50) | Branch if greater or equal |
| JAL | J-type | 0x6F | Unconditional jump |

### b. Simulation Flow.

The simulation flow represents the sequence of stages through which a RISC-V assembly program is transformed from human-readable source code into executed operations within a simulated processor. In the context of this project, the simulation flow serves a primarily educational purpose, exposing each internal step of instruction handling and execution in a clear and traceable manner. By explicitly modelling these stages, the simulator allows students and developers to understand how RISC-V instructions are processed internally and how their effects propagate through registers, memory, and control flow.

At a high level, the simulator follows a linear pipeline that mirrors the behaviour of a real processor: assembly input parsing, instruction encoding, simulator initialization, repeated execution through a fetch–decode–execute cycle, and final state reporting. Each stage feeds its output into the next, forming a coherent and deterministic execution flow.

#### 1. Assembly Input Parsing.

The simulation begins with the parsing of an input file containing RISC-V assembly code (.asm). The assembler component reads the input line by line, processing textual instructions into a structured internal representation. During this stage, comments and unnecessary whitespace are removed, and each line is analyzed to identify opcodes, registers, immediate values, and labels.

Labels are collected and resolved by associating them with their corresponding instruction addresses, enabling correct handling of control-flow instructions such as branches and jumps. Pseudo-instructions are also identified and expanded into one or more real RISC-V instructions, ensuring compatibility with the rest of the execution pipeline. Throughout parsing, the simulator performs basic validation checks and reports errors such as unknown instructions, invalid operands, or missing labels, preventing execution of malformed programs.

#### 2. Instruction Encoding.

Once the assembly code has been parsed and validated, each instruction is converted into its machine-level representation. This encoding stage translates the high-level instruction description into a 32-bit binary format that follows the RV32I specification. The encoder assigns opcodes, funct3 and funct7 fields, register indices, and immediate values to their correct bit positions depending on the instruction format.

Different instruction types (R, I, S, B, U, and J) are handled according to their specific layouts, including immediate reconstruction for formats where the immediate is split across multiple bit fields. The resulting encoded instructions resemble real RISC-V machine code and are stored sequentially in the simulator's instruction memory, forming the program that will be executed.

#### 3. Simulator Initialization.

Before execution begins, the simulator initializes its core components. Instruction memory is populated with the encoded instructions generated during the encoding phase. The register file is initialized according to RISC-V conventions, with all general-purpose registers set to zero. Special registers, such as the stack pointer, may be assigned predefined values depending on the simulator configuration.

The program counter (PC) is initialized to the address of the first instruction, establishing the starting point for instruction fetching. At this stage, the simulator is fully prepared to begin execution.

4. **Execution Loop.**

The core of the simulator consists of a loop that repeatedly executes instructions until the program terminates. Each iteration of this loop models the classical fetch–decode–execute cycle.

During the fetch stage, the simulator retrieves the instruction located at the address pointed to by the program counter. In the decode stage, the binary instruction is analysed to extract its opcode, source and destination registers, immediate values, and instruction type. This decoded information determines the operation that must be performed.

In the execute stage, the simulator performs the required computation. Arithmetic and logical instructions are handled by the ALU, memory instructions trigger read or write operations in the simulated memory, and control-flow instructions modify the program counter based on branch conditions or jump targets. After execution, the write-back stage updates the destination register, if applicable, with the computed result.

Finally, the program counter is updated. For most instructions, it advances sequentially to the next instruction, while branch and jump instructions override this behaviour by assigning a new target address. This cycle repeats until a termination condition is reached.

5. **Memory Model.**

The simulator employs a simplified memory model that represents the essential memory regions required for execution. Instruction memory stores the encoded program, while data memory is used for load and store operations. The simulator ensures that memory accesses remain within valid address ranges and that read and write operations behave consistently with RISC-V semantics.

Memory interactions are explicitly modelled to allow observation of how instructions affect stored values. This design makes it easier to debug programs and to understand the consequences of incorrect addressing or misaligned accesses.

6. **Debug and Logging Support.**

To improve traceability, the simulator provides logging mechanisms that record execution results. For each test program, the simulator generates output logs that include the final state of all registers and relevant memory locations. These logs are written to dedicated files, allowing users to inspect execution outcomes and compare them against expected results.

Error conditions such as invalid instructions, illegal memory accesses, or unexpected execution states are reported with descriptive messages, helping users identify and correct issues in their assembly programs.

7. **End of Simulation.**

The simulation terminates when the program reaches a defined stopping condition, such as the absence of further instructions to execute. Upon termination, the simulator performs final reporting, outputting the complete processor state. This final snapshot provides a clear summary of program behavior and serves as the primary verification artifact for testing and evaluation.

8. **Scalability and Extensibility.**

The simulation flow is designed to be modular and extensible. New instruction types or RISC-V extensions can be added by extending the parsing, encoding, and execution stages without redesigning the

entire pipeline. This structure also allows future enhancements, such as performance tracking, pipelined execution models, or support for additional architectural features, to be integrated incrementally.

### c. High-Level System Structure.

This section presents a high-level view of the internal structure of the RISC-V Assembly Simulator. It describes the main components of the system, their responsibilities, and the way they interact to simulate the execution of a RISC-V assembly program. The system is designed using a modular architecture, where each component represents a distinct functional unit commonly found in real processor implementations.

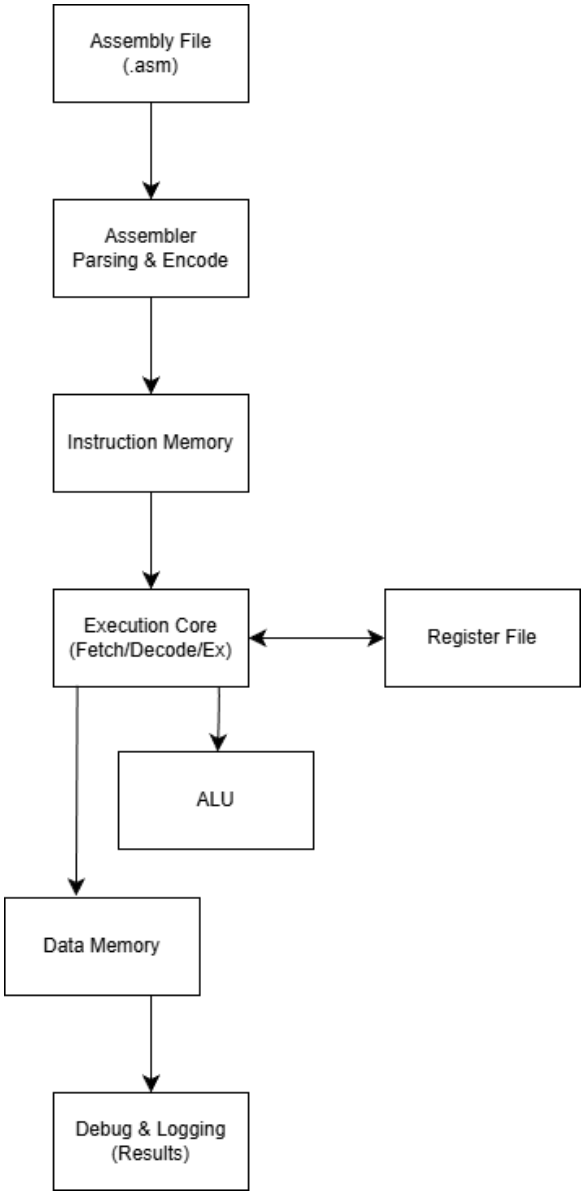
The primary objective of this structure is to clearly separate concerns such as input processing, instruction execution, memory handling, and result reporting. This separation improves readability, testability, and extensibility while making the simulator easier to understand from an architectural perspective.

#### 1. Key Components.

The simulator is composed of several interconnected components, each responsible for a specific aspect of program execution:

- **Assembler Module**: Processes the input assembly file, validates syntax, resolves labels and pseudo-instructions, and produces encoded machine-level instructions.
- **Instruction Memory**: Stores the encoded instructions in sequential order and provides read-only access during execution.
- **Data Memory**: Represents the simulated main memory used for load and store operations.
- **Register File**: Models the set of general-purpose RISC-V registers and stores intermediate computation results.
- **Program Counter (PC)**: Maintains the address of the currently executing instruction and controls execution flow.
- **Execution Core**: Implements the fetch–decode–execute cycle and coordinates decoding, ALU operations, and memory access.
- **Debug and Logging System**: Collects execution traces, register states, and error messages for analysis and testing.

The Figure below illustrates the high-level structure of the simulator and the interactions between the main components:




**2. Component Responsibilities.**

The **Assembler** serves as the entry point of the simulation flow. It transforms human-readable assembly code into a structured, machine-level representation by resolving labels, expanding pseudo-instructions, and encoding instructions according to the RV32I specification.

The **Instruction Memory** holds the encoded instructions and supplies them sequentially to the execution core based on the program counter value. It remains immutable during execution, reflecting real instruction memory behaviour.

The **Execution Core** orchestrates instruction execution. It retrieves instructions from instruction memory, decodes their fields, executes arithmetic or logical operations through the ALU, and triggers memory accesses or control-flow changes when required.

The **Data Memory** supports read and write operations requested by load and store instructions. It models address-based memory access and ensures that memory interactions follow correct semantics.

The **Register File** provides fast storage for operand values and execution results. Source registers are read during decoding, while destination registers are updated during the write-back phase.

The **Program Counter** determines the next instruction to be executed. It normally increments sequentially but may be updated explicitly by branch or jump instructions.

The **Debug and Logging System** records execution outcomes, detects invalid operations, and generates output logs that summarize the final processor state.

### 3. Inter-Component Interactions.

The simulator operates as a data-flow system with clearly defined interactions between components. The assembler outputs encoded instructions that are loaded into instruction memory. During execution, the execution core fetches instructions using the program counter, reads operand values from the register file, performs computations, and updates registers or data memory accordingly. Any exceptional conditions or final execution results are captured by the logging system for later inspection.

### 4. Extensibility.

The modular design of the system supports future extensions with minimal structural changes. New instruction formats or RISC-V extensions can be incorporated by extending the assembler, encoder, and execution logic. Similarly, enhancements such as performance counters, pipelining models, or visualization tools can be added without altering the core architecture.

The high-level system structure of the RISC-V Assembly Simulator emphasizes modularity, clarity, and educational value. By separating functionality into well-defined components, the simulator closely mirrors real processor organization while remaining accessible and extensible for future development and experimentation.

# III. Core modules.

## a. Instruction Module.

The Instruction Module defines the low-level representation and manipulation of encoded RISC-V instructions within the simulator. This module is implemented as a header-only component, located in **riscv-simulator/include/instruction.h**, and does not have a corresponding source (.c) file. This design reflects its role as a utility layer that provides data structures and inline functions for instruction field extraction, rather than maintaining state or complex control logic.

The module operates directly on 32-bit encoded instruction values and provides a consistent interface for interpreting these values according to the RISC-V instruction formats. By encapsulating instruction decoding primitives in a dedicated module, the simulator ensures that instruction handling remains centralized, efficient, and reusable across the execution pipeline.

At the core of the implementation is a unified representation for encoded instructions. The module defines a union that aggregates all supported instruction formats into a single data type. Each member of the union corresponds to a specific RISC-V encoding format (R, I, S, B, U, and J), while an additional field allows direct access to the raw 32-bit instruction value. This approach enables the same encoded value to be interpreted in multiple ways, depending on the instruction type determined during decoding, while preserving a compact and efficient memory layout.

Each instruction format is represented by a dedicated **typedef struct** containing a single **uint32_t** value. Although these structures do not store decoded fields explicitly, they serve as semantic wrappers that distinguish instruction formats at the type level. This design choice improves code clarity and allows format-specific handling to be expressed explicitly in the decoder and CPU modules.

Field extraction is implemented through a collection of **static inline** functions. These functions use bit masking and shifting operations to retrieve specific instruction fields, such as the opcode, destination register (**rd**), source registers (**rs1, rs2**), function codes (**funct3, funct7**), and immediate values. Implementing these accessors as inline functions eliminates function call overhead and allows the compiler to optimize instruction decoding aggressively, which is particularly important given the frequency of decode operations during execution.

Immediate value extraction is handled carefully to respect the RISC-V encoding rules. For instruction formats where immediate values are stored contiguously (such as I-type instructions), the module extracts the relevant bit range and applies sign extension when required. For formats with fragmented immediates, such as B-type and J-type instructions, the implementation reconstructs the immediate value by combining bits from multiple non-contiguous fields, followed by sign extension to produce the correct signed offset. This logic closely follows the RISC-V specification and ensures correct control-flow behavior during execution.

By providing a clean and well-defined interface for instruction field access, the Instruction Module serves as the foundation for the decoding and execution stages of the simulator. The Decoder Module relies on these inline getters to interpret encoded instructions, while the CPU Module uses the extracted fields to drive ALU operations, memory access, and program counter updates. As a result, this module plays a critical role in maintaining correctness and consistency throughout the fetch–decode–execute pipeline.

## b. CPU Module.

The CPU Module implements the execution core of the RISC-V Assembly Simulator and is responsible for simulating the behaviour of a processor during program execution. This module is implemented in **riscv-simulator/src/cpu.c** and maintains the complete architectural state of the simulated system, including the

register file, the program counter, and execution control flags. As the central coordinating component, the CPU module orchestrates the interaction between instruction decoding, arithmetic execution, memory access, and control-flow management.

The processor state is encapsulated within a dedicated CPU structure that aggregates all runtime data required for execution. This structure contains an array of 32 general-purpose registers, corresponding to the RISC-V register file, and a program counter that stores the address of the next instruction to be executed. In addition to core architectural state, the structure maintains pointers to the memory subsystem and the loaded assembly program, enabling direct access to instruction memory and data memory during execution. Execution progress is tracked through an instruction counter, while dedicated flags are used to signal halted or error states.

Initialization of the CPU is performed through explicit setup routines that prepare the processor for execution. During initialization, all registers except register x0 are reset to zero, preserving the RISC-V convention that x0 is hardwired to zero. The program counter is initialized to the start of instruction memory, execution counters are reset, and all control flags are cleared. A secondary initialization routine associates the CPU with a memory instance and a parsed assembly program, binding the processor to its execution context and enabling instruction fetch and data access.

Instruction execution within the CPU module follows a structured fetch–decode–execute pipeline. During the fetch phase, the CPU retrieves a 32-bit encoded instruction from instruction memory at the address specified by the program counter. The fetched value is stored as an encoded instruction representation, allowing it to be passed directly to subsequent stages without modification. Basic validation is performed to ensure that the CPU and memory references are valid before proceeding.

In the decode phase, the CPU examines the opcode field of the fetched instruction and determines its instruction format. Decoding is delegated to format-specific logic, which extracts register indices, immediate values, and function codes using the utilities provided by the Instruction Module. If an unsupported or invalid opcode is encountered, the CPU records an error state and terminates execution to prevent undefined behavior.

During the execute phase, the CPU performs the operation specified by the decoded instruction. Arithmetic and logical operations are delegated to the ALU module, while load and store instructions interact with the memory subsystem. Control-flow instructions compute branch conditions or jump targets and update the program counter accordingly. Execution logic is organized by instruction format to maintain clarity and allow incremental extension of supported instructions.

After execution, the CPU performs the write-back stage, updating the destination register with the computed result when applicable. The module explicitly prevents writes to register x0, ensuring correct architectural behaviour. The program counter is then updated either sequentially or explicitly, depending on whether the executed instruction modifies control flow. This mechanism accurately models both linear instruction execution and non-sequential jumps or branches.

Instruction execution is driven by a step-based control mechanism, where each step performs a single fetch–decode–execute cycle and increments the instruction counter upon successful completion. A higher-level execution loop repeatedly invokes this mechanism until the program terminates normally, an error condition is detected, or a predefined instruction limit is reached. This limit acts as a safeguard against infinite loops during testing and debugging.

For debugging and analysis purposes, the CPU module provides utilities to print the current processor state, including register values and execution metadata. These facilities allow users to inspect the effects of program execution and verify correctness against expected results. By centralizing execution control and state management, the CPU Module plays a critical role in ensuring the correctness, traceability, and educational value of the RISC-V Assembly Simulator.

### c. ALU Module.

The ALU Module implements the Arithmetic Logic Unit of the RISC-V Assembly Simulator and provides the core computational functionality required during instruction execution. This module is defined by the interface declared in **riscv-simulator/include/alu.h** and implemented in **riscv-simulator/src/alu.c**. Its role is to perform arithmetic, logical, and shift operations on 32-bit signed integer operands, as dictated by decoded RISC-V instructions.

The module is designed as a stateless component that operates exclusively on input operands and returns computed results. It does not maintain internal state and does not interact directly with memory or registers. This design mirrors the behaviour of a real hardware ALU and allows the CPU module to delegate computation cleanly and deterministically.

ALU operations are identified using a dedicated enumeration type that defines all supported arithmetic and logical operations. This enumeration includes entries for addition, subtraction, multiplication, division, bitwise operations, logical shifts, arithmetic shifts, and a fallback value representing unknown or unsupported operations. Using an enumeration provides a clear and extensible mapping between decoded instruction semantics and executable ALU functionality.

All ALU functionality is routed through a single execution interface. The execution function accepts an operation identifier and two signed 32-bit operands, then dispatches the computation using a structured selection mechanism. Each supported operation is implemented explicitly, ensuring predictable behavior and constant-time execution for all arithmetic and logical cases.

Arithmetic operations such as addition, subtraction, multiplication, and division are performed directly on signed integers, reflecting the semantics of RISC-V integer instructions. Logical operations operate at the bit level and include conjunction, disjunction, and exclusive OR. Shift operations are implemented in accordance with the RISC-V specification, with the shift amount constrained to the lower five bits of the second operand to ensure valid shift ranges for 32-bit values.

Logical right shifts are implemented by casting the input operand to an unsigned integer prior to shifting, ensuring that zero bits are shifted into the most significant positions. Arithmetic right shifts preserve the sign bit of the operand. This distinction accurately models the difference between logical and arithmetic shift instructions at the architectural level.

The ALU module includes basic error reporting for unsupported or invalid operations. If an unknown operation identifier is encountered, the module emits a diagnostic message and returns a neutral result. This behaviour prevents undefined execution while providing clear feedback during debugging and testing.

Within the simulator pipeline, the ALU Module is invoked exclusively by the CPU Module during the execute stage of instruction processing. The CPU determines the required ALU operation based on decoded instruction fields, retrieves operand values from the register file, and passes them to the ALU for computation. The resulting value is then returned to the CPU, which handles write-back to the destination register and any required program counter updates.

By isolating all arithmetic and logical functionality within a dedicated module, the simulator achieves a clear separation of concerns. This structure simplifies execution logic in the CPU module, improves readability, and ensures that computational behaviour remains consistent across all instruction types that rely on ALU operations.

### d. Memory Module.

The Memory Module implements the main memory subsystem of the RISC-V Assembly Simulator and is responsible for modelling byte-addressable memory used during program execution. This module is defined by the interface declared in **riscv-simulator/include/memory.h** and implemented in **riscv-simulator/src/memory.c**. Its primary role is to provide safe and controlled access to instruction and data memory while supporting program loading and debugging utilities.

The memory system is implemented as a dynamically allocated, contiguous block of bytes, allowing flexible memory sizing at runtime. Memory is represented internally by a dedicated structure that stores a pointer to the allocated byte array and the total size of the memory region. This abstraction enables the simulator to treat memory uniformly while enforcing bounds checks on all accesses.

Memory initialization is performed through an explicit allocation routine that reserves the requested number of bytes and initializes the entire memory region to zero. This ensures a clean and deterministic memory state prior to program execution. A corresponding cleanup routine releases the allocated memory and resets the internal state, preventing memory leaks and allowing reuse of the memory subsystem across simulation runs.

The Memory Module provides controlled access to memory through read and write operations that operate on 32-bit values. These operations implement little-endian encoding, where the least significant byte is stored at the lowest memory address, matching the conventions of modern RISC-V systems. Each read or write operation performs bounds checking to ensure that the requested address range lies entirely within the allocated memory region, preventing illegal memory access and undefined behaviour.

Instruction fetching and data access are unified through the same memory interface. The CPU module uses 32-bit read operations to fetch encoded instructions based on the program counter value, while load and store instructions rely on the same interface to read from or write to data memory. This unified approach simplifies integration and ensures consistent access semantics throughout the simulator.

In addition to basic read and write functionality, the Memory Module provides mechanisms for loading programs and static data into memory. Encoded instruction sequences are loaded into memory starting at a specified base address, allowing flexible placement of instruction memory. Similarly, data segments extracted from the assembly program are placed at configurable offsets, enabling separation between instruction and data regions within the simulated address space.

For debugging and verification purposes, the module includes utilities for inspecting memory contents. These functions allow dumping memory regions in word-sized increments, printing both addresses and values in a human-readable format. This capability is particularly useful for validating program loading, diagnosing incorrect memory accesses, and analysing execution results.

Error handling is an integral part of the Memory Module design. All memory operations verify pointer validity and address bounds before proceeding. When invalid access is detected, the module emits descriptive error messages and prevents the operation from being performed. This defensive approach improves simulator robustness and simplifies debugging by clearly identifying faulty memory interactions.

By encapsulating all memory-related behaviour within a dedicated module, the simulator achieves a clean separation between computation, control flow, and storage. The Memory Module provides a reliable foundation for instruction execution while closely modelling the behaviour of real processor memory systems in an educational context.

### e. Assembler Module.

The Assembler Module is responsible for reading, parsing, and translating RISC-V assembly source files into a structured representation that can be executed by the simulator. This module is implemented in **riscv-simulator/src/assembler.c**, with its interface defined in **riscv-simulator/include/assembler.h**. It serves as the bridge between human-readable assembly code and the internal data structures used by the simulator during instruction encoding and execution.

The module processes assembly source files line by line, identifying executable instructions, static data declarations, and symbolic labels. It organizes the parsed content into separate sections corresponding to executable code and static data, reflecting the conventional .text and .data segments used in assembly programming. This separation enables correct memory placement and simplifies later stages of program loading and execution.

At the core of the implementation is a set of structured data types that model the contents of an assembly program. Individual instructions are represented by a structure that stores the instruction label, opcode, operand list, operand count, source line number, and assigned memory address. Static data entries are represented separately, storing optional labels, literal values, and assigned memory addresses. Symbol definitions are captured using a dedicated structure that maps label names to resolved addresses, enabling efficient lookup during branch and jump resolution.

All parsed components are aggregated into a central program structure that maintains arrays of instructions, data entries, and symbols, along with counters that track the number of valid entries in each category. This structure represents the complete parsed assembly program and is passed to subsequent stages of the simulation pipeline for encoding and execution.

File parsing is initiated through a dedicated entry function that opens the assembly file and processes its contents sequentially. The parser identifies section directives such as .data and .text and switches parsing behaviour accordingly. Lines belonging to the data section are interpreted as static data declarations, while lines in the text section are parsed as executable instructions. The parser enforces a structured flow by associating addresses with instructions and data as they are encountered.

Comment handling and whitespace normalization are integral parts of the parsing process. The module removes both line-based and block-based comments before further processing, ensuring that only meaningful instruction content is analysed. Leading and trailing whitespace is eliminated to simplify tokenization and prevent parsing errors caused by formatting inconsistencies.

Operand parsing is performed by tokenizing operand lists using comma delimiters. Each operand is trimmed of extraneous whitespace and stored as a distinct string. This approach allows the simulator to support instructions with varying operand counts while maintaining a uniform representation for later decoding and execution.

Symbol resolution is handled by constructing a symbol table after parsing all instructions. Each label encountered during parsing is associated with the memory address of the corresponding instruction or data entry. This table enables branch and jump instructions to resolve symbolic targets during execution, ensuring correct control-flow behaviour.

For debugging and verification purposes, the Assembler Module provides utility functions that print the parsed program contents, including instructions, data entries, and symbol mappings. These functions allow developers to inspect the intermediate representation produced by the assembler and validate correctness before execution.

By encapsulating all parsing, symbol management, and program organization logic within a dedicated module, the simulator achieves a clean separation between source-level processing and execution-level behaviour. The Assembler Module ensures that input programs are transformed into a consistent and well-structured format that integrates seamlessly with the encoding, memory, and CPU components of the simulator.

### f. Encoder Module.

The Encoder Module is responsible for translating parsed RISC-V assembly instructions into their corresponding 32-bit binary machine code representation, as defined by the RISC-V Instruction Set Architecture. This module is implemented in **riscv-simulator/src/encoder.c**, with its interface declared in **riscv-simulator/include/encoder.h.** It serves as the final transformation stage between the assembler's structured representation of instructions and the binary format required by instruction memory and execution.

The module operates on instruction structures produced by the Assembler Module and encodes them according to their instruction type. It supports all standard RISC-V instruction formats used by the simulator, including R-type, I-type, S-type, B-type, U-type, and J-type instructions. Each format follows a fixed bitfield layout defined by the RISC-V specification, and the encoder ensures that all fields are placed at their correct positions within the 32-bit instruction word.

Encoding logic is organized around a set of format-specific builder functions. Each builder function is responsible for assembling a complete instruction word by combining opcode values, register indices, function codes, and immediate fields using bit masking and shifting operations. This modular approach isolates format-specific details and improves readability while ensuring correctness and consistency across instruction types.

The main encoding routine determines the appropriate instruction format based on the opcode of the parsed instruction. It extracts operand information, resolves register indices, parses immediate values, and validates all fields before invoking the corresponding builder function. This step ensures that only well-formed instructions are encoded and that invalid input does not propagate into the execution pipeline.

Immediate value handling is a critical aspect of the encoder implementation. The module includes explicit validation logic to ensure that immediate values fall within the allowed bit-width ranges dictated by the instruction format. For example, I-type immediates are validated against 12-bit signed limits, while branch and jump offsets are checked for both range and alignment constraints. These checks prevent incorrect encoding and provide early error detection during program translation.

Branch and jump instructions require special handling due to their fragmented immediate layouts. The encoder computes relative offsets based on resolved symbol addresses and encodes these offsets by distributing immediate bits across non-contiguous fields within the instruction word. This logic mirrors the RISC-V specification and ensures that control-flow instructions behave correctly during execution.

Error handling is integrated throughout the encoding process. The module detects and reports unsupported opcodes, missing or invalid operands, out-of-range immediates, and invalid register indices. When an error is encountered, descriptive diagnostic messages are printed, and encoding is aborted for the affected instruction. This approach simplifies debugging and prevents invalid machine code from entering instruction memory.

Within the simulator pipeline, the Encoder Module is invoked after assembly parsing and symbol resolution. The resulting encoded instructions are passed to the Memory Module, where they are stored in instruction memory and later fetched by the CPU during execution. By isolating instruction encoding in a dedicated module, the simulator maintains a clean separation between source-level parsing and binary-level execution.

### g. Decoder Module.

The Decoder Module implements the operand decoding layer of the RISC-V Assembly Simulator and is responsible for translating textual assembly operands into structured values usable by the rest of the pipeline. This module is defined by the interface declared in **riscv-simulator/include/decoder.h** and implemented in **riscv-simulator/src/decoder.c**. Its primary purpose is to convert register names and aliases, numeric immediates, and memory-style operands into normalized numeric representations that can be consumed by the assembler, encoder, and CPU execution logic.

The module focuses on decoding assembly-level syntax rather than decoding 32-bit machine instructions. It interprets human-readable operand strings such as x10, sp, -12, 0x1F, or 12(x1) and produces canonical forms such as register indices (0–31), signed immediate values, and base-register-plus-offset pairs. By centralizing operand parsing and validation in a dedicated module, the simulator avoids duplicated parsing logic and ensures consistent behaviour across all components that depend on operand interpretation.

Register decoding is implemented through a function that maps textual register identifiers to their numeric indices. The decoder supports both explicit register names of the form xN and conventional RISC-V aliases such as sp, ra, and argument or temporary registers. For explicit xN registers, the implementation extracts and validates the numeric suffix and rejects values outside the legal range of 0–31. For aliases, the decoder consults a predefined alias mapping and returns the corresponding architectural register index. When an unknown alias or invalid register name is encountered, the module reports an error and returns an invalid result to prevent incorrect downstream encoding or execution.

Immediate decoding is implemented through a parsing routine that converts numeric literals from their string form into a signed 32-bit integer value. The implementation supports multiple common representations, including decimal and hexadecimal forms, and handles negative values where applicable. Basic validation is performed to ensure the input is not empty and that conversion produces a meaningful numeric result. This immediate parsing routine is reused by multiple instruction encoders and by memory operand parsing, ensuring consistent handling of literal values throughout the simulator.

Memory operand decoding is implemented to support addressing expressions of the form offset(base), such as 12(x1) or -4(sp). The decoding logic splits the string around the parentheses, parses the offset as an immediate value, and resolves the base register using the register decoding routine. The results are returned as a pair consisting of a signed offset and a base register index, allowing load and store instructions to compute effective addresses correctly during execution. Invalid formats, such as missing parentheses or malformed components, are detected and reported through explicit error messages.

Error handling is integrated throughout the Decoder Module. For invalid register names, malformed immediates, or incorrectly formatted memory operands, the module prints descriptive diagnostic messages and returns invalid values. This behaviour prevents silent failures and improves debugging by clearly identifying operand-level issues before they propagate into encoding or execution.

Within the simulator pipeline, the Decoder Module is used primarily during assembly processing and instruction encoding. The Assembler module produces tokenized operands, the Decoder module converts those operands into numeric representations, and the Encoder module then places the resulting fields into correct bit positions when generating the 32-bit machine instruction word. By providing a consistent operand decoding interface, the module contributes directly to correctness and maintainability across the simulator.

# IV. Main Entry Point.

The **main.c** file serves as the entry point of the RISC-V Assembly Simulator and is responsible for orchestrating the entire simulation process. It integrates all major modules of the system, including the Assembler, Encoder, Memory, and CPU modules, and coordinates their interaction in a well-defined, sequential workflow. The main program does not implement simulation logic itself but acts as a controller that drives each stage of the pipeline from input parsing to program execution and cleanup.

Execution begins by validating command-line arguments and ensuring that a single assembly source file is provided as input. If the required argument is missing or invalid, the program terminates immediately with an informative error message. This early validation prevents undefined behaviour later in the execution flow.

The first operational stage consists of parsing the input assembly file. The main program invokes the assembler interface to read and process the source file, producing an **AssemblyProgram** structure that contains parsed instructions, static data entries, and symbol definitions. If parsing fails due to file I/O errors or malformed assembly syntax, the program reports the failure and terminates gracefully.

After successful parsing, the memory subsystem is initialized. A contiguous block of memory is dynamically allocated to serve as the simulator's main memory. The memory size is currently fixed within the entry point, ensuring a deterministic and controlled execution environment. Once initialized, the memory module is ready to accept both instructions and data.

Instruction encoding is performed next. The main program iterates over all parsed instructions and invokes the Encoder Module to translate each instruction into its 32-bit binary representation. Encoded instructions are stored in a temporary array, preserving their original order. During this stage, the program validates encoding results and detects errors such as invalid operands, unsupported instructions, or out-of-range immediates. If an encoding failure occurs, execution is aborted and all allocated resources are released before termination.

Once encoding is complete, the binary instruction sequence is loaded into memory starting at address zero. This establishes the instruction memory region used during execution. Static data entries extracted from the .data section are then loaded into memory at a calculated offset following the instruction block, ensuring that instruction and data regions do not overlap.

With memory populated, the CPU module is initialized and bound to the memory subsystem and parsed program. Register state is reset, the program counter is initialized to the beginning of instruction memory, and internal execution flags are cleared. At this stage, the simulator is fully prepared to begin execution.

Program execution is initiated by invoking the CPU's execution routine. The CPU repeatedly performs fetch–decode–execute cycles until the program terminates normally, an execution error occurs, or a predefined execution limit is reached. Throughout execution, the CPU interacts with instruction memory, data memory, and the ALU as required by each instruction.

After execution completes, the main program prints the final processor state. This includes register contents, execution status flags, and summary information such as whether the CPU halted normally or encountered an error. This final state provides a clear snapshot of the program's outcome and is particularly useful for debugging and test validation.

Finally, the program performs cleanup by releasing all dynamically allocated resources, including encoded instruction buffers and memory allocations. A successful cleanup ensures that the simulator terminates cleanly without resource leaks.

Through this structured control flow, the main program provides a clear and robust integration layer that connects all simulator components into a coherent execution pipeline. Its sequential design improves readability, simplifies debugging, and ensures that failures at any stage are detected and handled consistently.

# V. Build System.

## a. CMake Lists.

The project uses CMake as the primary build system generator. CMake provides a platform-independent way to describe how the simulator is compiled and linked, allowing the project to be built consistently across different operating systems and toolchains. In this project, the **CMakeLists.txt** file defines the project language, the required C standard, the include paths, the set of source modules, and the final executable target.

The configuration starts with **cmake_minimum_required(VERSION 3.10)**, which enforces a minimum CMake version to ensure that all commands used in the build script are available and behave predictably. Next, **project(RISCV_Simulator C)** declares the project name and specifies that the project is written in C, enabling CMake's C toolchain detection and compilation rules.

The simulator is written in C and relies on a consistent language standard; therefore, the build configuration explicitly sets the standard using **set(CMAKE_C_STANDARD 99)** and **set(CMAKE_C_STANDARD_REQUIRED ON)**. This forces compilation in C99 mode and prevents fallback to older language modes, ensuring that the code compiles deterministically even when using different compilers.

The project headers are stored in a dedicated include/ directory, so **include_directories(include)** is used to add that directory to the compiler's header search path. This allows all modules to include project headers using a clean form such as **#include "cpu.h"** without requiring relative paths.

The source code is split into multiple modules to mirror the simulator's architecture. These compilation units are listed in the **SRC_FILES** variable, which groups all implementation files (**alu.c, assembler.c, cpu.c, decoder.c, encoder.c, memory.c**) together with the entry point (**main.c**). This structure ensures that each major subsystem is compiled independently and linked into a single executable, improving maintainability and reflecting the modular design described in the architecture and core modules sections.

Finally, the executable target is defined using **add_executable(riscv_simulator ${SRC_FILES})**. This command instructs CMake to compile the listed source files and link them into the final program named riscv_simulator. When built, this target produces the simulator binary that is then used by the Makefile targets for running automated tests and generating execution logs.

Overall, this **CMakeLists.txt** provides a minimal but clear and portable build definition: it enforces the correct C language standard, keeps include paths organized, compiles the simulator as a set of modular components, and generates a single executable artifact suitable for both manual usage and automated testing.

## b. Makefile.

Although CMake is used as the primary build system generator, the project also includes a Makefile that acts as a convenience interface for building, testing, and managing the simulator. The Makefile does not replace CMake, instead, it standardizes common development workflows into short, easy-to-remember commands.

This approach allows developers to interact with the project using simple **make** commands, without needing to remember the full set of CMake invocations or directory-specific details. Internally, all compilation and configuration steps are still performed through CMake.

The Makefile serves several goals. It enforces an out-of-source build structure by always using a dedicated build/ directory, provides a single entry point for configuring, building, cleaning, and rebuilding the project,

automates test discovery and execution including structured log generation, and ensures consistent behavior across different environments and build types. By separating configuration, build, and test concerns into named targets, the Makefile improves usability while keeping the underlying build system portable and robust.

The simulator is built into the build/ directory, and the resulting executable is named riscv_simulator. The build type can be selected at invocation time, with Release being the default optimized build and Debug enabling debug symbols. This behavior is controlled through the BUILD_TYPE variable, which is passed directly to CMake during configuration, for example by invoking **make BUILD_TYPE=Debug**.

During the configuration phase, the Makefile internally executes the equivalent of the **cmake -DCMAKE_BUILD_TYPE=<type> ..** command from inside the build/ directory. This step reads the project configuration, detects compiler and system settings, and prepares the build environment for compilation. Users typically do not need to run this command manually, as it is handled automatically.

Invoking **make** or **make all** configures the project if necessary and builds the simulator executable using CMake. This is the standard way to compile the project after cloning the repository. The **make sim** command performs the same build step explicitly, ensuring that the simulator target is compiled and linked.

Test management is also integrated into the Makefile. The **make list-tests** command scans the tests/ directory for all .asm files and prints the discovered test programs, allowing users to verify that test discovery is functioning correctly. Running **make test** builds the simulator if needed, executes all discovered test programs, stores their output in tests/results/<test>_out.log files, prints a summary of passed and failed tests, and returns a non-zero exit code if any test fails. This behavior makes the command suitable for both local validation and automated testing environments.

For focused debugging, the **make run TEST=<file.asm>** command executes a single test program, verifies that the specified file exists, runs the simulator on that input, and writes the output to the corresponding log file. The **make logs** command runs all tests and generates log files for each of them without producing a final summary, which is useful when only raw execution output is required.

The Makefile also provides commands for cleaning build artifacts. Invoking **make clean** removes compiled objects and intermediate files while preserving the build/ directory and the CMake cache, enabling a clean recompilation without reconfiguration. The **make distclean** command removes the entire build/ directory, forcing a complete reconfiguration on the next build. For convenience, **make rebuild** combines these steps by performing a full cleanup followed by a fresh build from scratch.

Finally, the **make help** command prints a concise overview of available targets and supported variables, serving as a quick reference for users unfamiliar with the Makefile.

Overall, the Makefile complements the CMake-based build system by providing a clear, task-oriented interface that simplifies common development operations while preserving portability and configurability.

In addition to the Makefile-based workflow, the project can also be built by invoking CMake directly. This approach may be useful for users who prefer working with CMake commands or for environments where the Makefile wrapper is not used. In such cases, an out-of-source build is still recommended. The typical sequence consists of creating a build directory, configuring the project, and then compiling the executable by running **mkdir build**, followed by **cd build**, **cmake ..**, and finally **cmake --build .**. This sequence produces the same **riscv_simulator** executable as the Makefile-based build process and relies on the same configuration defined in the **CMakeLists.txt** file.

# VI. Testing.

This chapter presents the testing strategy used to validate the correctness and robustness of the RISC-V Assembly Simulator. The tests included in the repository are designed to exercise a wide range of instruction types, control-flow patterns, and arithmetic behaviors supported by the simulator. Rather than focusing solely on isolated instructions, the test programs implement complete algorithmic tasks, thereby demonstrating correct interaction between multiple instructions across iterative and conditional execution paths.

All tests are written in RISC-V assembly and executed using the automated testing infrastructure provided by the project's Makefile. Each test is executed independently, and its output is captured in a dedicated log file, allowing both functional validation and detailed inspection of execution behaviour.

The test suite includes programs that cover arithmetic computation, conditional branching, looping constructs, memory access, and interaction between registers and memory. Together, these tests provide strong evidence that the simulator correctly implements the fetch–decode–execute pipeline and maintains architectural state consistently across complex execution scenarios.

One of the more computationally intensive tests is the **Fibonacci sequence program**. This test computes the *n*-th Fibonacci number iteratively, using multiple registers to track intermediate values and a loop controlled by conditional branch instructions. The program exercises arithmetic operations, register updates, conditional branching, and unconditional jumps. It also validates correct loop termination and data consistency across multiple iterations, making it an effective stress test for both control flow and arithmetic correctness.

The **sum of the first *n* natural numbers** test provides a simpler but fundamental validation scenario. This program uses a loop to accumulate values from 1 to *n*, relying on addition, register incrementation, and branch-equal instructions for loop control. Although straightforward, this test confirms correct handling of iterative execution, comparison operations, and memory storage of results.

The **factorial** computation test introduces repeated multiplication and loop-based accumulation. By computing the factorial of a given input value, this test validates multiplication instructions, loop termination conditions, and proper sequencing of arithmetic updates. It also verifies that the simulator correctly handles base cases and termination logic using branch instructions.

The **conditional sum** test introduces more nuanced control flow by combining arithmetic, branching, and bitwise logic. This program determines whether the input value is even or odd using a bitwise AND operation and then conditionally executes one of two distinct computation paths. This test demonstrates correct evaluation of conditional expressions, branching based on computed values, and selective execution of code blocks.

The **modulo** computation test validates arithmetic precision and the correct interaction between division, multiplication, and subtraction instructions. By computing the remainder of a division using basic arithmetic operations, this program ensures that division results are handled correctly and that dependent arithmetic instructions produce accurate results. This test is particularly important for verifying the correctness of multiplication and division instructions and their integration within larger arithmetic expressions.

An additional test focuses on computing **powers of two** using bit manipulation. This program calculates $2^n$ by applying the Shift Left Logical (SLL) instruction to an initial value of one. By shifting the binary representation of the value left by n positions, the test efficiently computes powers of two without using multiplication. This scenario demonstrates correct implementation of logical shift operations and highlights

the simulator's ability to support efficient bit-level arithmetic. The test also verifies correct interaction between shift instructions and memory load/store operations.

The **counting set bits** test, also known as a Hamming weight computation, evaluates the simulator's ability to manipulate and analyse binary data. This program iteratively examines each bit of a 32-bit integer by masking the least significant bit using a bitwise AND operation and accumulating the result. A division operation is used to effectively shift the value right between iterations. This test exercises bitwise logic, arithmetic accumulation, division, and loop control, demonstrating correct handling of low-level binary operations and iterative processing.

A **memory swap** test validates correct memory access semantics by exchanging the values stored at two fixed memory locations. The program loads both values into registers, swaps them using register-level operations, and writes them back to memory at their original locations. This test demonstrates proper usage of load and store instructions and confirms that memory writes correctly update the underlying memory model.

The **vector-like sum** test simulates traversal of an array stored in memory by summing values located at sequential memory offsets. The program iteratively loads each element, accumulates the total sum in a register, and advances through memory using fixed offsets. Loop termination is controlled using comparison and branch instructions. This test demonstrates sequential memory access patterns, loop control, and cumulative arithmetic, providing confidence in the simulator's handling of memory traversal scenarios.

Across all test programs, the simulator demonstrates correct handling of arithmetic instructions such as addition, subtraction, multiplication, and division, as well as logical operations, conditional and unconditional branching, memory load and store instructions, and register state updates. The tests also confirm correct symbol resolution, memory addressing, and program counter updates during execution.

Test execution is automated using the project's Makefile, which discovers all assembly test files in the designated test directory and executes them sequentially. For each test, the simulator's output is redirected to a corresponding log file stored in the results directory. These logs contain detailed execution output and final processor state, enabling precise validation without cluttering the console output.

In conclusion, the test suite provides comprehensive coverage of the simulator's core functionality. By combining algorithmic correctness tests with control-flow and arithmetic stress cases, the tests demonstrate that the simulator faithfully executes RISC-V assembly programs and maintains architectural correctness across complex execution paths.

# VII. Conclusions.

This project presented the design and implementation of an educational RISC-V Assembly Simulator written in C, with the primary goal of modelling the execution of RISC-V programs in a clear, modular, and analysable manner. The simulator was developed to illustrate the internal stages of instruction processing, from assembly parsing and binary encoding to instruction execution and architectural state updates, closely following the principles of the RV32I instruction set.

A key strength of the simulator lies in its modular architecture. Each major subsystem, such as the assembler, encoder, decoder, instruction representation, memory model, arithmetic logic unit, and CPU execution core, was implemented as a distinct component with well-defined responsibilities. This separation of concerns improves code readability, simplifies debugging, and enables individual modules to be extended or replaced without affecting the overall structure of the system. The architecture mirrors real processor organization, reinforcing the conceptual link between software simulation and hardware design.

The correctness of the simulator was validated through an extensive and carefully constructed test suite. The included tests cover a wide range of execution scenarios, including arithmetic computation, looping constructs, conditional branching, bitwise manipulation, and memory access patterns. More complex programs, such as Fibonacci sequence generation, factorial computation, modulo calculation, and vector-like memory traversal, demonstrate correct interaction between multiple instruction types across iterative and conditional execution flows. The successful execution of all test programs, combined with automated testing and structured log generation, provides strong evidence that the simulator faithfully implements the intended execution semantics.

Beyond functional correctness, the simulator offers significant educational value. By exposing each stage of the fetch–decode–execute pipeline and maintaining explicit control over registers, memory, and control flow, the project enables users to observe how high-level assembly programs are translated into low-level machine behaviour. This makes the simulator a useful learning tool for students studying computer architecture, low-level programming, or instruction set design, as well as for developers seeking a deeper understanding of RISC-V execution principles.

While the simulator focuses on correctness and clarity rather than performance or completeness, this choice was intentional. The current implementation models a single-cycle, in-order execution model without pipelining, caching, or advanced exception handling. These limitations help keep the system accessible and easier to reason about, while still providing a realistic architectural foundation.

The modular design of the simulator naturally supports future extensions. Potential improvements include support for additional RISC-V extensions (such as RV64I or the full M extension), enhanced error reporting and diagnostics, configurable memory layouts, performance counters, and more advanced execution models such as pipelining or step-by-step interactive debugging. These extensions could be implemented incrementally without fundamental changes to the existing architecture.

In conclusion, this project successfully demonstrates a complete and well-structured RISC-V assembly simulation environment. Through careful design, comprehensive testing, and clear separation of responsibilities, the simulator achieves its educational and technical objectives while providing a solid foundation for future development and experimentation.

# VIII. Bibliography:

**[1] riscv-spec.pdf**