Computer Architecture Project

Phase 1

Done by:
Yazan AbuBakir 20231232
Ahmad Alkayid 20220890

1. Introduction

This report describes the process for Phase 1 of the RISC-V processor project.

Phase 1 Overview:

- **Implementing a custom RISC-V assembler for a subset of RV32I instructions**
- **Designing the core CPU modules in Verilog**
- **Integrating the modules into a 5-stage pipeline (without hazards or forwarding)**
- **Testing each part with Verilog testbenches**
- **Running simple arithmetic and memory programs on the CPU**

This report is organized in the exact order in which we implemented the system, starting with the assembler, since the assembler’s output is required to run programs on the CPU.

2. Assembler

We built the assembler using Python because it is easy to write, run, and debug. Python also provides useful features like the re library for parsing text, as well as convenient structures such as dictionaries and tuples, which helped us organize instruction formats and opcode values. The assembler's job is to convert RISC-V assembly code into 32-bit machine code that can be loaded directly into our CPU's instruction memory.

Assembler Requirements

Phase 1 requires supporting all base RISC-V instruction formats:

- R-type: add, sub, and, or, xor, shifts
- I-type: addi, lw, jalr
- S-type: sb, sw
- SB-type: conditional branches
- U-type: lui, auipc
- UJ-type: jal

Step 1 – Define all instructions

We started by creating a Python dictionary that lists every instruction we support.
Each entry includes its type (R, I, S, etc.), opcode, funct3, and funct7.
This dictionary guides how each instruction is encoded.

Step 2 – Create helper functions

We wrote small helper functions to:

- convert numbers and immediates
- turn registers like x5 into binary
- build two's-complement values for immediates

These functions make the encoding process simpler and cleaner.

Step 3 – Encode instructions

The encoder checks the instruction type and places all fields
(rd, rs1, rs2, immediate, opcode...) into the correct 32-bit format.
Each instruction type (R, I, S, SB, U, UJ) has its own encoding rules.

Step 4 – Parse the assembly line

We used Python's re library to split each line and extract the operands.
This also handles special layouts like lw x1, 0(x2).

Step 5 – Produce the final hex output

After encoding, the 32-bit binary is converted into an 8-digit hex value.
Each hex line is written into the output .hex file that our CPU loads into instruction memory.

Step 6 – Testing the assembler

We wrote small assembly programs to make sure our assembler worked correctly.
These tests helped us confirm that labels, immediates, and instruction encodings were generated properly before moving on to CPU integration.

2. CPU module Development

After finishing the assembler, we started building the main CPU modules in Verilog. Each module performs one important job inside the processor. We tested every module separately. Below is a detailed explanation of each module and what it does.

1.Program Counter

The Program Counter is a fundamental component of the CPU responsible for holding the address of the instruction that will be executed in the current cycle. Because each RISC-V instruction is 32 bits (4 bytes), the PC typically increments by 4 every clock cycle to move to the next instruction in memory.
This updating happens on the positive edge of the clock, in accordance with the project requirements.
Dummy functional test:
Cycle 0 → PC = 0
Cycle 1 → PC = 4
Cycle 2 → PC = 8
This shows the PC correctly steps sequentially through the instruction memory.

2.Instruction Memory

The Instruction Memory is a read-only memory (ROM) used to store the machine-code program generated by our assembler. We load this memory using $readmemh so that the CPU can fetch instructions during simulation.
When the PC provides an address, the instruction memory outputs the corresponding 32-bit instruction, which will then enter the decode stage.
Dummy functional test:
For PC = 8 → Instruction fetched = 0x002081B3, confirming correct addressing.

3.Register File

The register file provides fast storage for frequently used data and contains 32 registers, each 32 bits wide.
It supports two simultaneous reads and one write, which is essential for RISC-V instructions that generally require two operands and produce one result.
By convention, register x0 is hardwired to zero, meaning that any attempt to write to it is ignored.
Dummy functional test:
Writing value 25 into x5 and then reading x5 correctly returns 25.

4.Immediate Generator

Many RISC-V instructions include immediate values, but these values are encoded in different bit positions depending on the instruction format. The Immediate Generator is responsible for extracting and sign-extending these immediate fields so they can be used by the ALU or for calculating memory and branch addresses.

It supports all required immediate types: I-type, S-type, SB-type, U-type, and UJ-type.

Dummy functional test:

For the instruction addi x3, x1, -7, the generated immediate is 0xFFFFFFF9, which is the correct sign-extended 32-bit value.

5.ALU - Arithmetic Logic Unit

The ALU performs the actual computation inside the CPU.

It takes two 32-bit inputs and executes an operation determined by the control unit.

Supported operations include:

- arithmetic (add, subtract)
- logic (AND, OR, XOR)
- shift operations (logical and arithmetic)

The ALU’s output is used for register writes, memory addressing, and making branch decisions.

Dummy functional tests:

5 + 7 → 12

10 − 3 → 7

These results show correct ALU functionality.

6.Data Memory

The Data Memory stores program data and is accessed by load and store instructions such as lw, lh, sw, sb.

Based on the control signals, the memory performs either a read or write.

This module supports the required 8-bit addressable memory layout and is essential for implementing the load/store architecture of RISC-V.

Dummy functional test:

After storing 0xABCD1234 at address 200, loading from the same address returned 0xABCD1234, confirming correct memory behavior.

7.Control Unit

The Control Unit interprets the opcode, funct3, and funct7 fields of each instruction to generate the necessary control signals for the datapath.

These signals determine:

- whether the register file should write
- whether the data memory should read or write
- whether the ALU uses a register or immediate
- which ALU operation to perform
- whether the instruction is a branch, load, store, or jump

This module ensures that each instruction activates the correct hardware components inside the pipeline.

The ALU Control Unit decides exactly what operation the ALU should do for each instruction. It takes the 2-bit ALUOp from the main Control Unit and combines it with the instruction's funct3 and funct7 values. Using this information, it creates a 4-bit alu_ctrl signal that tells the ALU whether it should add, subtract, shift, or perform a logical operation. This unit is important because some instructions look similar at first but actually require different actions, and the ALU Control Unit makes sure the ALU does the correct job every time.

3.Pipeline Integration

After verifying every module, we connected them into the standard 5-stage RISC-V pipeline:
IF – Fetch instruction
ID – Decode instruction, read registers
EXE – Execute using ALU
MEM – Load/store memory
WB – Write results back to register file
Pipeline registers were added between stages to hold intermediate values.
This allowed multiple instructions to be processed at the same time.

Instruction Fetch (IF) Stage

The Instruction Fetch stage is where everything begins. Here, the CPU looks at the current value of the Program Counter (PC) and uses it to pick the next instruction from memory. Once it grabs the instruction, the PC is moved forward to point to the following one. This stage keeps the flow of instructions steady so the processor always has work to do.

Instruction Decode (ID) Stage

In the Decode stage, the CPU takes a closer look at the instruction it just fetched and figures out what needs to happen next. It identifies which registers are involved, what type of operation is required, and whether an immediate value is needed. During this stage, the register file provides the values stored in the requested registers, and the control unit prepares the signals that guide the rest of the pipeline.

Execute (EXE) Stage

The Execute stage is where the actual calculation or decision takes place. The ALU uses the values from the previous stage and performs the needed operation—this might be adding numbers, comparing values, shifting bits, or working out a memory address. For branch instructions, this stage also decides if the program should continue normally or jump somewhere else.

Memory Access (MEM) Stage

In the Memory stage, the CPU works with data stored in memory. If the instruction is a load, the processor reads data from memory. If it's a store, it writes data to a specific memory address. Instructions that don't involve memory simply pass through this stage untouched. It acts as a bridge between calculations and actual stored data.

Write Back (WB) Stage

The Write Back stage is the last stop in the pipeline. Here, the CPU takes the final result from either the ALU or the Data Memory and places it into the destination register. This ensures that the output of the instruction is saved and ready to be used by any upcoming instructions.

4.Conclusion

In Phase 1 of the project, we built the core parts of a simplified RISC-V processor. We started by creating a Python-based assembler that translates RISC-V assembly instructions into 32-bit machine code. This assembler allowed us to test our CPU with real programs and made the rest of the project possible.

We then designed and implemented all required CPU modules in Verilog, including the Program Counter, Instruction Memory, Register File, Immediate Generator, ALU, Data Memory, and Control Unit. Each module was tested on its own to make sure it behaved correctly before connecting everything together.

After that, we combined the modules into a 5-stage pipeline. We tested the full pipeline using simple arithmetic and load/store programs, and the CPU produced the correct results. Observing the simulation waveforms helped us confirm that values flowed correctly between stages and that control signals were working as expected.

Overall, Phase 1 gave us a solid understanding of how a pipelined CPU operates and how different modules interact to complete an instruction. The work completed here prepares us for Phase 2, where we will add important performance features such as forwarding, hazard detection, branch prediction, and caching.