16-BIT RISC ARCHITECTURE BASED PROCESSOR

GROUP-16

1) INSTRUCTION SET ARCHITECTURE (ISA)

1.1) Introduction

We have used 16-bit instructions. RISC consists of defining the following instruction formats: R-type, I-type, and B-Type. R-type instructions operate on three registers. I-type instructions operate on two registers and a 5-bit immediate.

1.2) R-type instructions

The name R-type is short for register-type. R-type instructions use three registers as operands: two as sources, and one as a destination. The figure below shows the R-type machine instruction format. The 16-bit instruction has six fields: op, rs1, rs2, rd, and func. Each field is 3 or 2 bits, as indicated. **op** (also called opcode or operation code) and **func** (also called the function). All R-type instructions have an opcode of **00**. The specific R-type operation is determined by the function fields. The operands are encoded in the three

fields: rs1, rs2, and rd. The first two registers, rs1, and rs2 are the source registers; rd is the destination register.

R-TYPE

3-bit	3-bit	3-bit	3-bit	2-bit**(00)**
Rs2	Rs1	Rdest	func	op

Instructions	func
ADD	000
SUB	001
AND	010
OR	011
NOT	100
XOR	101

1.3) I-type instructions

The name I-type is short for immediate-type. I-type instructions use two register operands and one immediate operand. The figure below shows the I-type machine instruction format. The 16-bit instruction has five fields: op, Addreg, rd, funct, and offset. The first three fields, op, Addreg, and rd, are like those of R-type instructions. The offset field holds the 5-bit immediate. The funct holds the operation to be performed. The operation is determined by the opcode and funct. The operands are specified in the two fields Addreg and offset . Addreg and offset are always used as source operands. rd is used as a destination.

I-TYPE

5-bit	3-bit	3-bit	3-bit	2-bit
offset	Addreg	Rd	func	op

Instructions	OP
LDR	01
STR	10

1.4) B-type instructions

The name B-type is short for branch-type. This format is used only with branch instructions. B-type instructions use two register operands and one immediate operand. Figure below shows the B-type machine instruction format. The 16-bit instruction has 5 fields: op, funcB, rs1, rs2 and offset. The offset fields hold the 5-bit immediate. The operation is determined by the opcode and funcB.

B-TYPE

5-bit	3-bit	3-bit	3-bit	2-bit**(10)**
offset	rs2	rs1	funcB	op

Instructions	funcB
BE	000
BNE	001
BL	010
BLE	011
BG	100
BGE	101

1.5) CALL AND HALT

The "CALL" instruction initiates a function call, with the 5-bit field specifying the function's starting address, a 2-bit opcode (11) denoting a function operation. This facilitates jumping to a subroutine, preserving the return address for resumption post-execution. The "HALT" instruction, encoded with 111111 in the 6-bit field, 11111 in the 5-bit field, and 111 and 11 for the 3-bit and 2-bit fields respectively, stops program execution entirely, signaling the end of processing or entering an idle state. These instructions are crucial for control flow and system operation management.

Function call

6-bit	5-bit	3-bit**(110)**	2-bit**(11)**
	Func start addr	func	op

Function call return

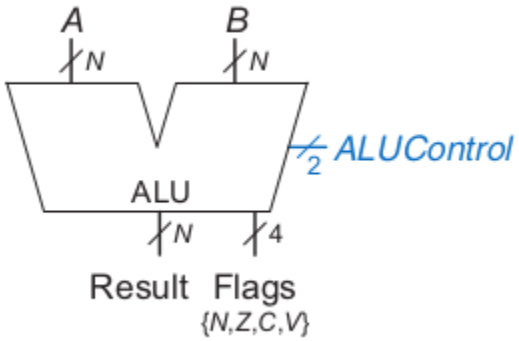
6-bit**(111111)**	5-bit**(XXXXX)**	3-bit**(111)**	2-bit**(11)**
		func	op

Halt

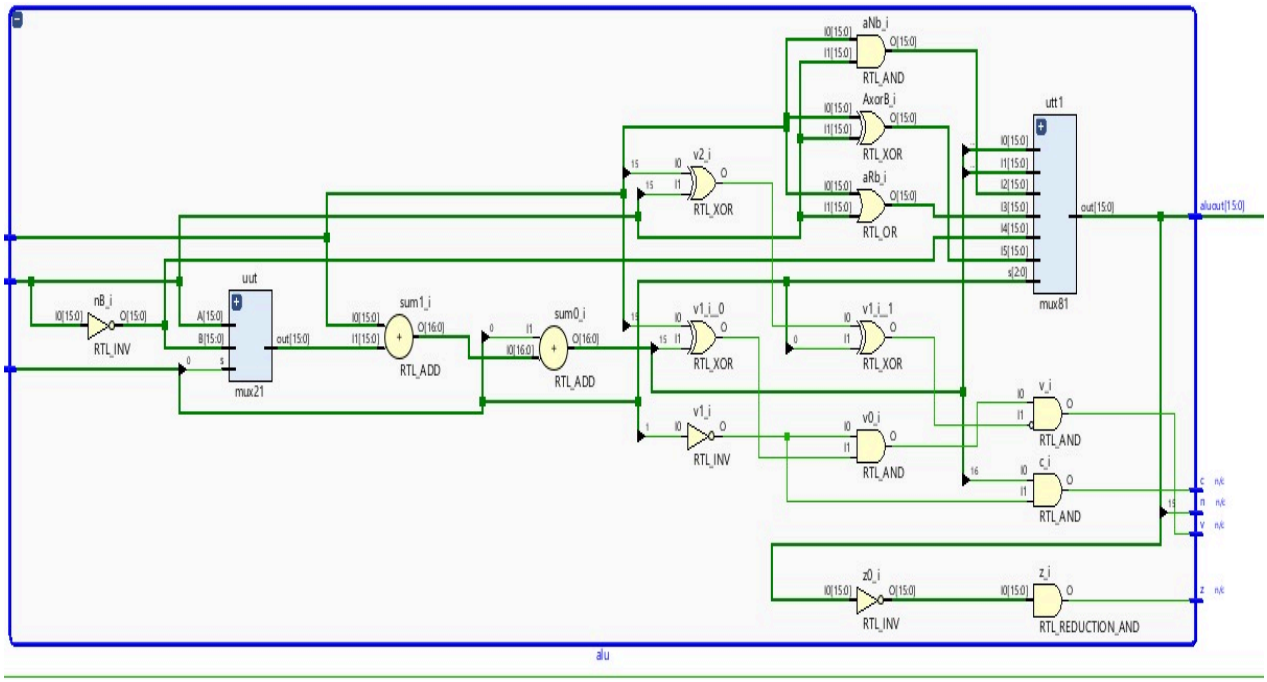
6-bit**(111111)**	5-bit**(11111)**	3-bit**(111)**	2-bit**(11)**

2) ALU

Arithmetic Logic Unit (ALU), which is a key part of any digital system, responsible for performing various arithmetic and logical operations. The ALU takes two 16-bit inputs, `A` and `B`, and a 3-bit control signal, `aluControl`, to decide which operation to perform. The result of the operation is output as a 16-bit value `aluout`, along with flags that provide important information about the result: `z` (zero flag), `n` (negative flag), `v` (overflow flag), and `c` (carry flag). The ALU supports operations like addition, bitwise AND, OR, XOR, and negation. It uses internal modules like `mux21` (a 2-to-1 multiplexer) and `mux81` (an 8-to-1 multiplexer) to control data flow and select the appropriate operation. Addition and subtraction are implemented using carry logic, with subtraction achieved by complementing one of the inputs. The flags provide useful status information: the `z` flag indicates if the result is zero, `n` shows if the result is negative, `v` detects overflow in arithmetic operations, and `c` reflects carry or borrow during addition or subtraction. This ALU design is modular and demonstrates an efficient way to perform essential computations in digital circuits.



ALU DESIGN



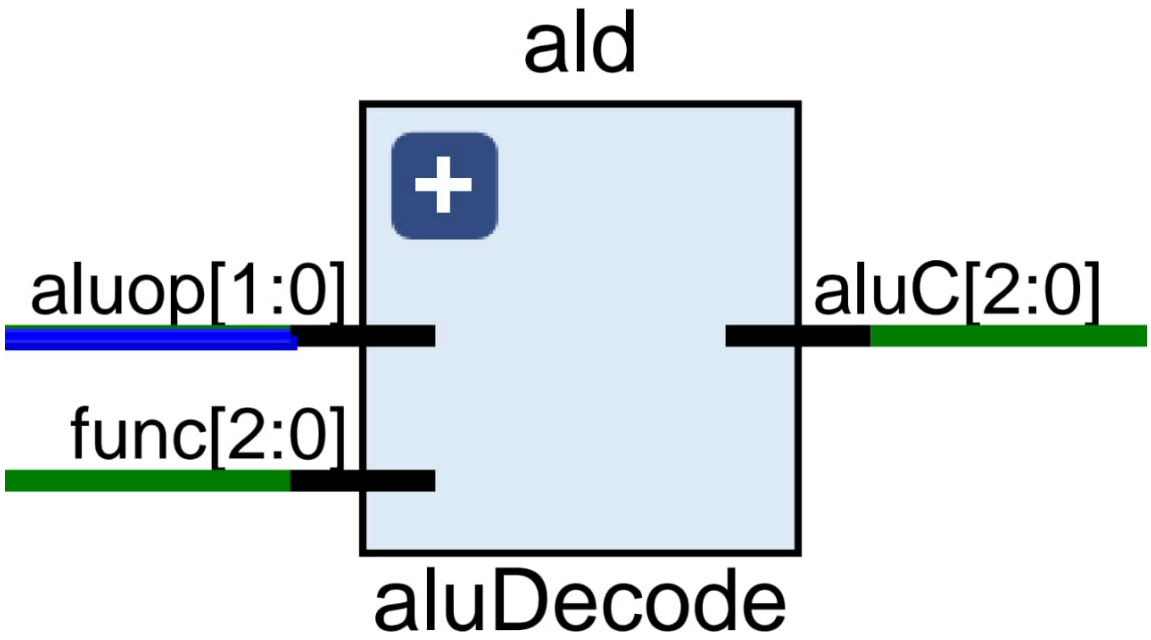
ALU control	Instructions
000(ADD)	ldr, str, add
001(SUB)	br, sub
010(ADD)	add
011(OR)	or
100(NOT)	not
101(XOR)	xor

B) Datapath Architecture

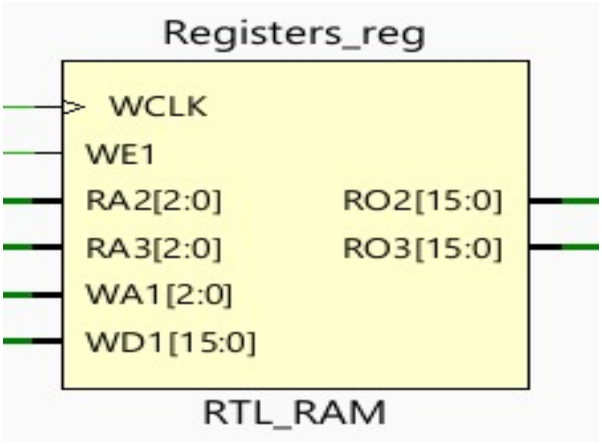
1. Block Diagram

The datapath of the microprocessor consists of the following key components:

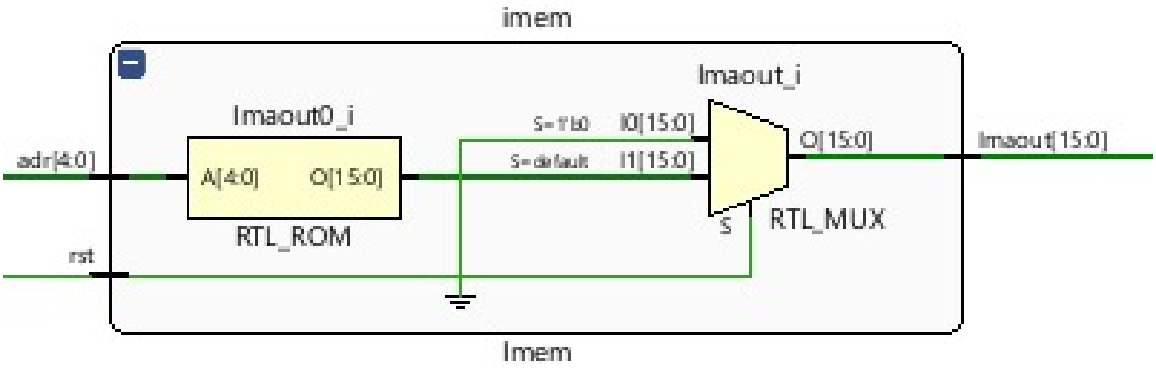
- **ALU (Arithmetic Logic Unit)**: Executes arithmetic and logical operations.



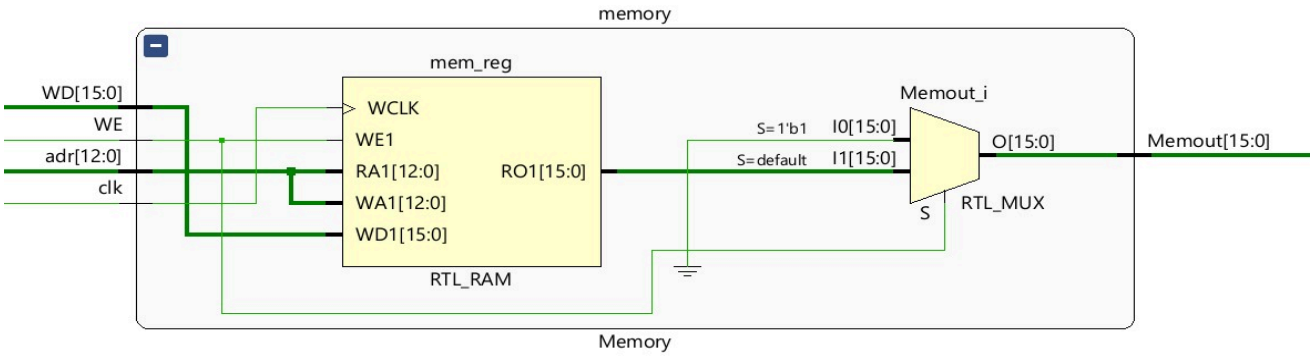
- **Register File**: Provides fast storage for operand data and stores computation results.



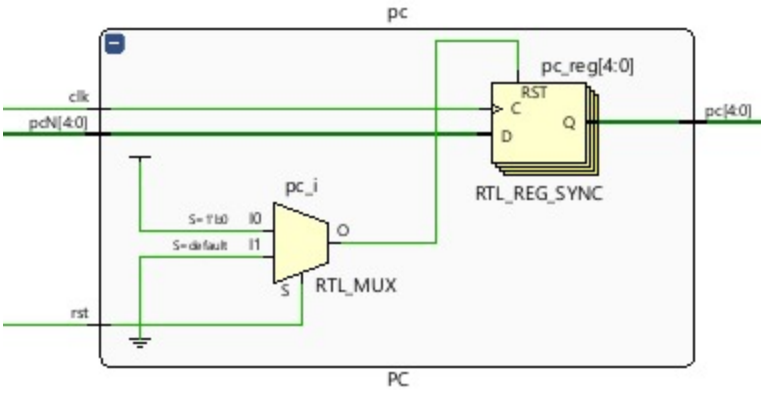
- **Instruction Memory**: Stores the program's instructions to be fetched and executed.



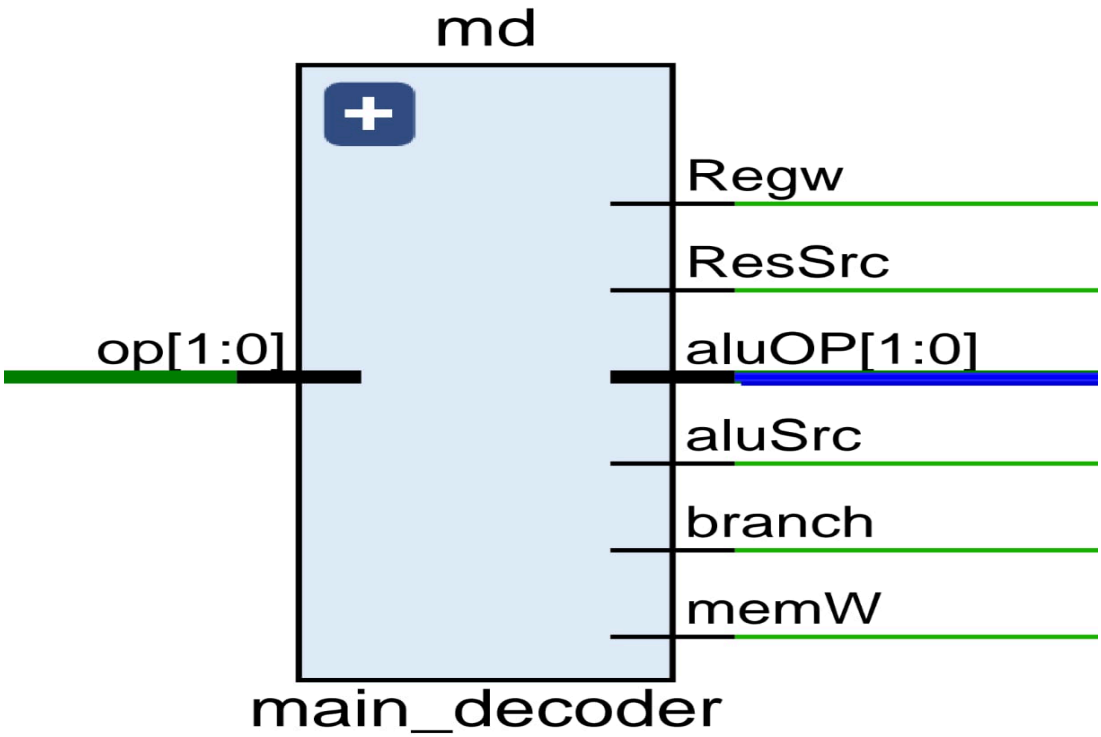
- **Data Memory**: Stores data for load and store instructions.



- **Multiplexers (MUX)**: Select inputs for the ALU and PC.
- **Program Counter (PC)**: Tracks the address of the next instruction.



- **Control Unit**: Decodes instructions and generates appropriate control signals for each stage of execution.



2. Function of Each Component

- **Program Counter (PC)**: Holds the memory address of the current instruction. It increments sequentially or jumps based on control signals.
- **Instruction Memory**: Provides the current instruction to be executed.
- **ALU**: Processes arithmetic and logic operations using control signals from the Control Unit.
- **Register File**: Stores and retrieves operand data for operations.
- **Data Memory**: Provides data storage and retrieval for load/store instructions.
- **MUX**: Directs the appropriate data or address input to components.
- **Control Unit**: Generates control signals to manage the datapath components, including ALU operations, memory access, and register updates.

---

3. Instruction Execution Process

Example for the instruction **lw rd, offset(rs):**

1. **Fetch**: The instruction is fetched from the instruction memory using the PC.

2. **Decode**: The opcode, source register (**rs**), destination register (**rd**), and immediate value **(offset**) are decoded.
3. **Address Calculation**: The ALU computes the effective memory address as **rs + offset**.
4. **Memory Access**: Data is fetched from the computed address in data memory.
5. **Writeback**: The fetched data is written to the destination register `rd`.

---

4. Trade-offs and Design Decisions

Cost vs. Speed

- To maintain a balance between cost and performance, shared components like multiplexers are used.

Single-Cycle vs. Multi-Cycle Design

- A multi-cycle design is adopted to reuse components like the ALU and memory in different cycles, reducing hardware requirements.

Shared vs. Dedicated Components

- Shared components like the ALU and memory reduce cost but require more control signals, adding complexity.

Edge-Triggered Registers vs. Latching

- Edge-triggered registers are used for synchronization, ensuring accurate timing in clock cycles.

Other Considerations

- Instruction format and control signals were designed for simplicity and scalability.

C) Controller Design

Control signals

op(2-bit)	Regw	memW	aluSrc	aluOP(2-bit)	ResSrc	branch
00	1	0	0	10	0	0
01	1	0	1	00	1	0
10	0	0	0	01	0	1
11	0	1	1	00	1	0

Explanation of Signals:

- **Regw**: Determines if the register file is written (1) or not (0).
- **memW**: Controls whether memory is written (1) or not (0).
- **aluSrc**: Selects between register or immediate value for the ALU's second operand (1 for immediate, 0 for register).
- **aluOP**: Determines the operation type for the ALU:
    - 10: Arithmetic operations
    - 01: Branch comparison

- 00: Logical operations

- **ResSrc**: Chooses the source for the result (1 for memory, 0 for ALU).
- **branch**: Indicates if the instruction is a branch (1 for branch, 0 otherwise).

aluop	func	aluC	Explanation
00	XXX	000	Default operation when aluop = 00 .
01	XXX	001	Subtract operation when aluop = 01.
10	000	000	Pass-through or AND operation for func = 000 -
10	001	001	Function-specific operation (e.g., OR) for func = 001 .
10	010	010	Function-specific operation (e.g., ADD) for func - 010 .
10	011	011	Function-specific operation (e.g., XOR) for func = 011
10	100	100	Function-specific operation (e.g., SLT) for func = 100
10	101	101	Function-specific operation (e.g., NOR) for func = 101.
10	110	110	Function-specific operation (e.g., shift) for func = 110.
10	111	111	Function-specific operation (e.g., custom) for func = 111.
Others	XXX	000	Default operation for any other aluop value.

F) Remarks:

a) Learnings: This project provided valuable insights into the design and implementation of a 16-bit RISC processor. Key learnings include:

1. **Understanding of Processor Design**:
    - Learned the step-by-step process of creating a functional processor, from defining the **Instruction Set Architecture (ISA)** to implementing the datapath and control logic.
    - Gained a deeper understanding of how different components like the **ALU**, **Register File**, and **Instruction Memory** interact within a CPU.

2. **Modularity and Integration**:
    - Realized the importance of modular design in complex systems. By breaking the processor into submodules (e.g., ALU, Branch Unit), debugging and testing became manageable.
    - Learned how to integrate and synchronize these modules to achieve proper execution of instructions.
3. **Control Signal Management**:
    - Learned how control signals are generated (via decoders) and how they guide data flow through the datapath.
    - Understood how the **main_decoder** and **aluDecoder** modules work in tandem to implement the fetch-decode-execute cycle.
4. **Branching and Function Calls**:
    - Explored conditional branching and how the **Branch Unit** and stack enable control flow changes for loops, conditionals, and function calls.
5. **Hardware Description Languages (HDLs)**:
    - Practiced using **Verilog** for implementing and simulating processor components.
6. **Teamwork and Collaboration**:
    - Learned to divide tasks effectively, ensuring equal contribution while maintaining cohesive integration of all components.

b)Possible Improvements:

While the current design is functional, there are several ways to improve its performance:

1. **Pipeline Implementation**:
   - Introduce pipelining to overlap instruction execution stages (fetch, decode, execute, etc.).
   - This would increase instruction throughput by allowing multiple instructions to execute simultaneously.
2. **Instruction-Level Optimization**:
   - Add more optimized instructions to handle specific operations like multiply (`MUL`) or shift (`SHL`, `SHR`) instead of relying solely on basic operations (`ADD`, `SUB`).
3. **Memory Access Optimization**:
   - Introduce a **cache memory** system to reduce latency for frequently accessed data or instructions.
4. **Dual-Port Register File**:
   - Use a dual-port register file to allow simultaneous read/write operations, reducing bottlenecks in operand fetching.
5. **Reduce Critical Path Delays**:
   - Optimize combinational circuits, especially in the **ALU**, **Decoder**, and **Branch Unit**, to minimize propagation delay.
6. **Power Efficiency**:
   - Use clock gating or other low-power design techniques to reduce energy consumption during idle states.

c)Different ways:

Reflecting on the project, here's what could be improved in future implementations:

1. **Plan Testing Earlier**:
    - Allocate more time to test individual modules in isolation before integration. This could prevent time-consuming debugging at later stages.
2. **Expand the ISA Thoughtfully**:
    - Start with a smaller and more focused ISA that covers essential operations. Expand it incrementally based on simulation results and needs.
3. **Implement a Multi-Cycle Datapath**:
    - Explore a multi-cycle design for a more balanced trade-off between speed and hardware complexity, reducing resource usage for specific instructions.
4. **Use Advanced Debugging Techniques**:
    - Employ **assertion-based verification** to catch errors earlier during simulation.
    - Leverage waveform analysis tools like **ModelSim** to visualize signal transitions better.
5. **Optimize Resource Usage**:

- Minimize redundant logic, particularly in control signal generation and data path routing, for a more efficient FPGA implementation.

6. **Introduce Higher-Level Abstractions**:
    - Use parametrized modules in Verilog to simplify scaling (e.g., supporting larger word sizes like 32-bit).
7. **Detailed Documentation**:
    - Create better inline code documentation and a step-by-step project guide to simplify understanding and replication of the design.

d) Advice: If someone is embarking on a similar project, here's our advice:

1. **Start Small**:
    - Begin with a simple CPU design (e.g., a single-cycle processor with a few instructions) before adding complex features like branching, function calls, or pipelining.
2. **Understand the Basics**:
    - Fully grasp the fundamentals of CPU architecture, RISC principles, and Verilog/VHDL before diving into the project.
3. **Plan Thoroughly**:

- Define the ISA, datapath, and control signals early on. A well-thought-out design will save significant debugging time later.

4. **Modular Design**:
   - Break the project into manageable submodules (e.g., ALU, Register File, Memory) and verify each module independently before integration.
5. **Test Incrementally**:
   - Test the CPU incrementally: first individual modules, then the integrated datapath, and finally complete instruction execution.
6. **Simulation Tools**:
   - Familiarize yourself with simulation tools like **ModelSim** or **Vivado**. These will be essential for debugging and verification.
7. **Leverage Reusability**:
   - Reuse common modules like multiplexers or decoders. Avoid redundant code by making modules generic and parameterized.
8. **Documentation and Teamwork**:
   - Maintain thorough documentation and communicate effectively with team members to ensure everyone understands their roles and contributions.