CS2323: Computer Architecture

Early implementation status and challenges report

Design and FPGA Implementation of a 64-bit RISC-V Processor with FPU and UART Interface

Submitted By:

Krishna H. Patil	(EE24BTECH11036)
Arnav Yadnopavit	(EE24BTECH11007)

Indian Institute of Technology Hyderabad

December 18, 2025

Contents

1. Design Plan

As suggested in the proposal stage, the final deliverable of our project is to design and demonstrate a **multicycle 64-bit RISC-V processor** that supports the **RV64I base instruction set** and is compatible with the **M (Multiply/Divide)** and **F (Floating Point)** extensions. The processor will also incorporate a basic **Floating Point Unit (FPU)** and a two-level cache hierarchy (**L1 and L2 caches**) to improve performance and to try to match a realistic system's behavior.

To reach this final goal, we have structured our work into slow and steady but incremental stages. The first stage focuses on developing a **single-cycle 64-bit RISC-V CPU** implementing the core integer instruction set without the M and F extensions. This phase aims to ensure the working and clear understanding of the fundamental processor components and their interconnections. Specifically, the following modules form the core of the first stage:

- **ALU (Arithmetic Logic Unit):** Performs basic arithmetic and logical operations.
- **Datapath:** Defines the overall data flow between key hardware components.
- **Control Unit:** Generates control signals based on instruction decoding.
- **Immediate Generator:** Extracts and sign-extends immediates from instructions.
- **Instruction Memory (IMEM):** Stores program instructions in hexadecimal format.
- **Data Memory (DMEM):** Used for load and store operations.
- **Register File:** Contains 32 general-purpose 64-bit registers supporting dual reads and single writes per cycle.
- **Program Counter and Branch Logic:** Manages instruction sequencing and control flow.

The design is being implemented in **Verilog HDL** and simulated using **Xilinx Vivado**.

1.1 block diagram

For the single-cycle processor stage, we decided to follow the datapath and control model discussed during class and as detailed in *Patterson and Hennessy's "Computer Organization and Design RISC-V Edition"*.

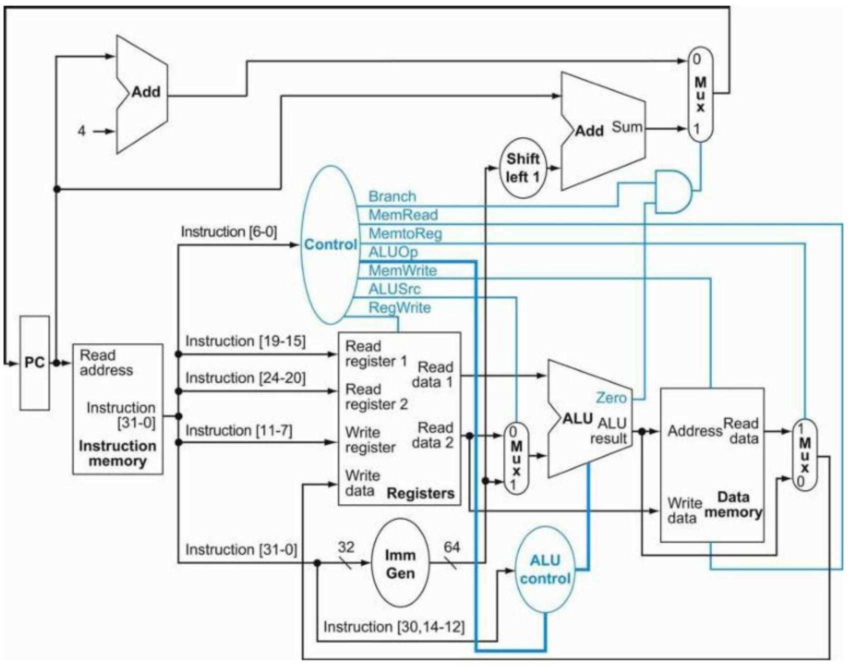


Figure 1: The single cycle block diagram (Adapted from Patterson and Hennessy).

2. Current Implementation Status

The first implementation stage, corresponding to the complete single-cycle 64-bit RISC-V CPU, has been successfully completed. All core components have been implemented in Verilog HDL, tested through simulation testbenches, and verified using Vivado synthesis tools. The design has been developed in a modular structure, with each hardware block individually implemented, simulated, and validated before integration.

The following Verilog modules constitute the single-cycle processor:

- `ALUControl.v` – Generates control signals for the ALU based on the instruction function codes.
- `ImmGen.v` – Extracts and sign-extends immediate values from the instruction according to RISC-V formats (I, S, B, U, J).
- `alu.v` – Performs arithmetic and logical operations on 64-bit operands as dictated by control signals.
- `controlunit.v` – Decodes the opcode and generates the necessary control lines for other modules.
- `datapath.v` – Integrates all major processor components, forming the complete dataflow for instruction execution.
- `dmem.v` – Implements the data memory for load and store instructions.

- `imem.v` – Stores program instructions
- `pc.v` – Handles the program counter logic, including sequential and branch address updates.
- `regfile.v` – Implements the 32 general-purpose 64-bit registers with dual read and single write capabilities.
- The block-level interconnections form a conventional single-cycle datapath where each instruction is fetched, decoded, executed, and committed within a single clock cycle. The `controlunit.v` and `ALUControl.v` modules generate all necessary control signals, while the `datapath.v` module integrates these functional units to define the overall dataflow.
- At the moment, the Verilog implementation of all modules has been completed and validated through simulation testbenches. Functional verification has confirmed the correctness of arithmetic, logical, and branch instruction execution. However, during synthesis of the integrated top-level module, we observed higher-than-expected LUT utilization. Individual modules (including IMEM and DMEM) synthesized efficiently and were correctly inferred as BRAMs when compiled separately, but in the integrated synthesis, these memories appear to be mapped to LUT, significantly increasing the total LUT count.
- We are currently investigating this synthesis behavior. Our hypothesis is that BRAM inference is being lost due to integration-level synthesis context or certain HDL constructs. Once this is fixed, we will proceed to FPGA implementation and then extend the verified single-cycle core into a multicycle 64-bit processor.
- Preliminary simulations demonstrate correct execution of R-type and I-type instructions such as `ADD`, `SUB`, `AND`, `OR`, and `ADDI`, Work is now focused on transitioning this verified single-cycle design into a multicycle architecture.

3. Early Results

At this stage of the project, the core functional blocks of the 64-bit single-cycle RISC-V processor have been implemented and verified through simulation. Each module was individually tested with dedicated testbenches to confirm functional correctness before top-level integration. The primary objective of these early experiments was to ensure proper instruction decoding, control signal generation, arithmetic and logic operations, and program counter updates.

Note: Some initial testbench templates were generated with ChatGPT and later modified as per project requirements.

Program Counter (PC)

The `pc.v` module was implemented as a 64-bit register with synchronous reset and enable signals. It updates its value every clock cycle based on the branch control logic and immediate offsets. Simulation tests confirmed that the PC correctly increments sequentially during normal execution and properly applies branch and jump targets when control signals are asserted. The timing diagram also verifies clean transitions without metastability or skipped cycles.

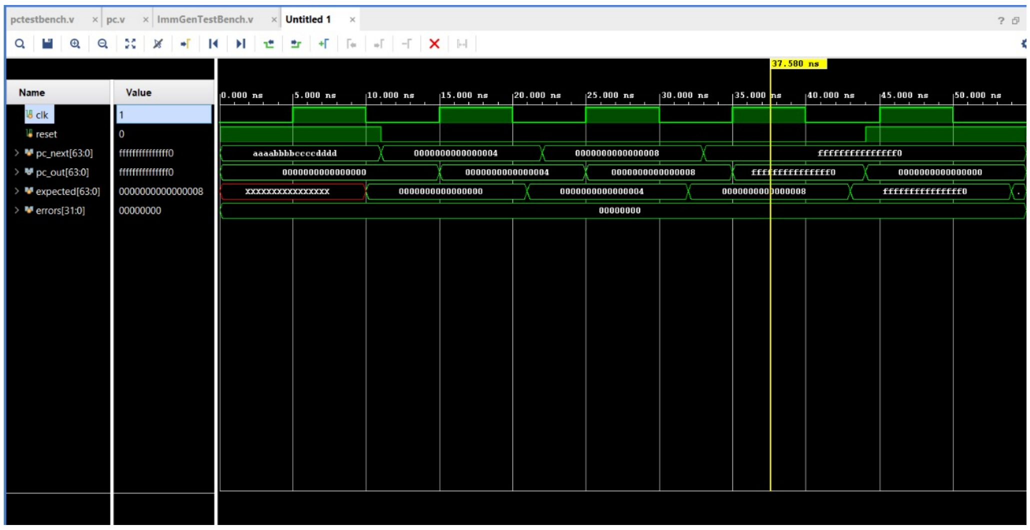


Figure 2: Waveform of the Program Counter

3.1 ALU and ALU Control Verification

The Arithmetic Logic Unit (ALU) and ALU Control Unit were among the first modules to be tested. The ALU supports all major arithmetic, logical, and comparison operations defined in the RISC-V base integer ISA. The control unit generates operation select signals based on `func7`, `func3`, and the instruction type. Table **??** summarizes the verified ALU control mapping.

The synthesis report for the ALU module is summarized below. The module uses 985 LUTs and 4 DSP blocks, which is well within the available FPGA resources.

control [4:0]	**out [63:0]**	**branchAlu**	**valid**
00000	a + b	0	1
01000	a - b	0	1
0?100	a ^b	0	1
0?110	a | b	0	1
0?111	a & b	0	1
0?001	a <<b	0	1
00101	a >>b (logical)	0	1
01101	a >>b (arithmetic)	0	1
0?010	{63'b0, $(signed(a) < signed(b))$}	0	1
0?011	{63'b0, (a <b)}	0	1
1?000	0	(a == b)	0
1?001	0	(a != b)	0
1?100	0	$(signed(a) < signed(b))$	0
1?101	0	$(signed(a) >= signed(b))$	0
1?110	0	(a <b)	0
1?111	0	(a >= b)	0
default	0	0	0

Table 1: ALU Control Truth Table for `control` = {`branchinst`, `func7[5]`, `func3`}

Site Type	**Used**	**Fixed**	**Prohibited**	**Available**	**Util%**
Slice LUTs*	985	0	0	63400	1.55
LUT as Logic	985	0	0	63400	1.55
LUT as Memory	0	0	0	19000	0.00
Slice Registers	0	0	0	126800	0.00
F7 Muxes	1	0	0	31700	<0.01
F8 Muxes	0	0	0	15850	0.00
DSPs (DSP48E1)	4	0	0	240	1.67

Table 2: Synthesis Resource Utilization for `alu.v`

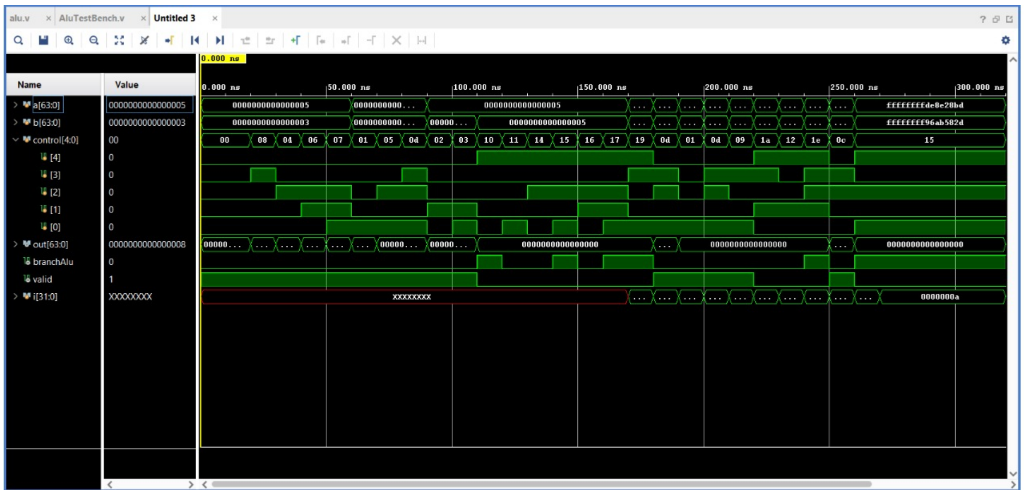
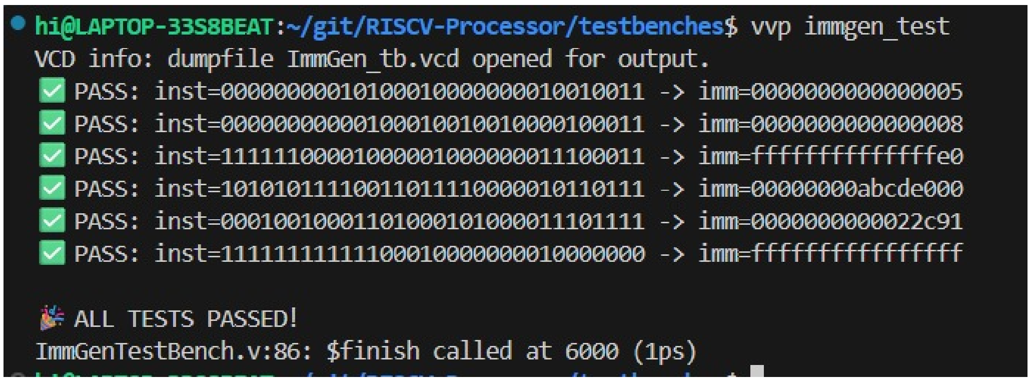


Figure 3: Simulation waveform of the ALU

3.2 Immediate Generator

The Immediate Generator (`ImmGen.v`) was tested with instruction samples of types I, S, B, U, and J. Timing diagrams confirm proper extraction and sign-extension of immediate values according to the RISC-V specification



```
● hi@LAPTOP-33S8BEAT:~/git/RISCV-Processor/testbenches$ vvp immgen_test
  VCD info: dumpfile ImmGen_tb.vcd opened for output.
   ✅ PASS: inst=00000000010100010000000010010011 -> imm=0000000000000005
   ✅ PASS: inst=00000000000100010010010000100011 -> imm=0000000000000008
   ✅ PASS: inst=11111100001000001000000011100011 -> imm=ffffffffffffffe0
   ✅ PASS: inst=10101011110011011110000010110111 -> imm=00000000abcde000
   ✅ PASS: inst=00010010001101000101000011101111 -> imm=0000000000022c91
   ✅ PASS: inst=11111111111100010000000010000000 -> imm=ffffffffffffffff

   🎉 ALL TESTS PASSED!
  ImmGenTestBench.v:86: $finish called at 6000 (1ps)
```

Figure 4: Waveform showing immediate generation

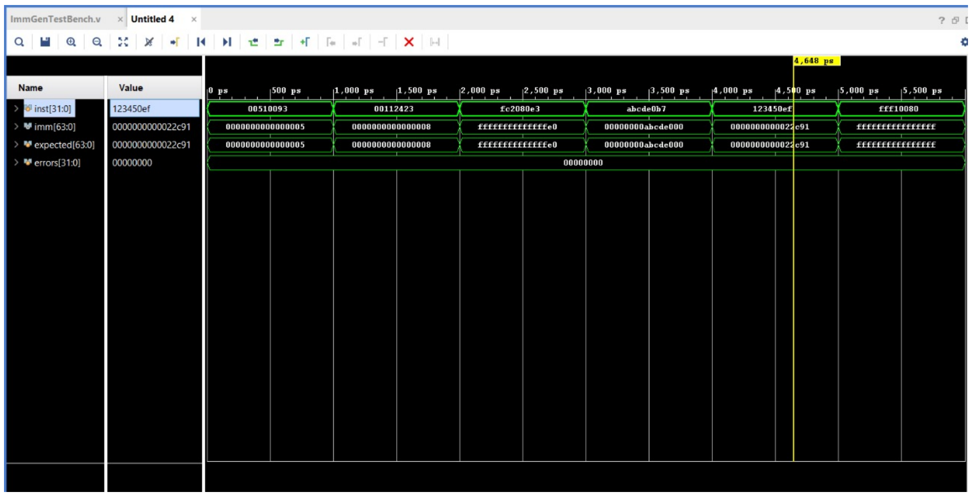


Figure 5: Waveform showing immediate generation

3.3 Memory Modules (IMEM and DMEM)

Both instruction and data memories were validated through simulation using preloaded contents initialized with the `$readmemh` directive. The `imem.v` module successfully fetches instructions word-by-word based on the current PC value, while `dmem.v` correctly performs load and store operations. Testbench outputs confirm the expected data flow and address calculations.

3.4 Control Unit

The control unit (`controlunit.v`) was verified for correct decoding of major RISC-V instruction formats. A bug causing constant load instruction decoding was identified and

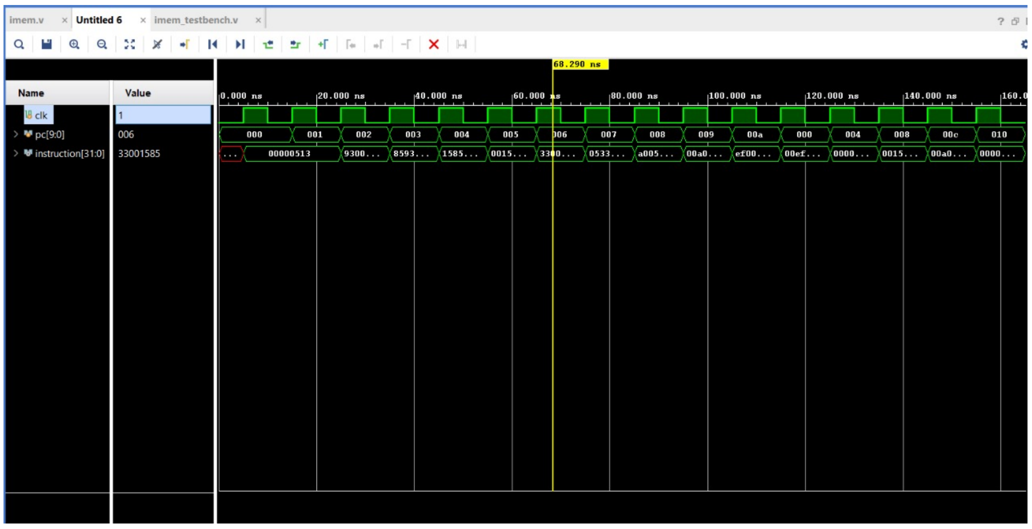


Figure 6: Simulation output for IMEM

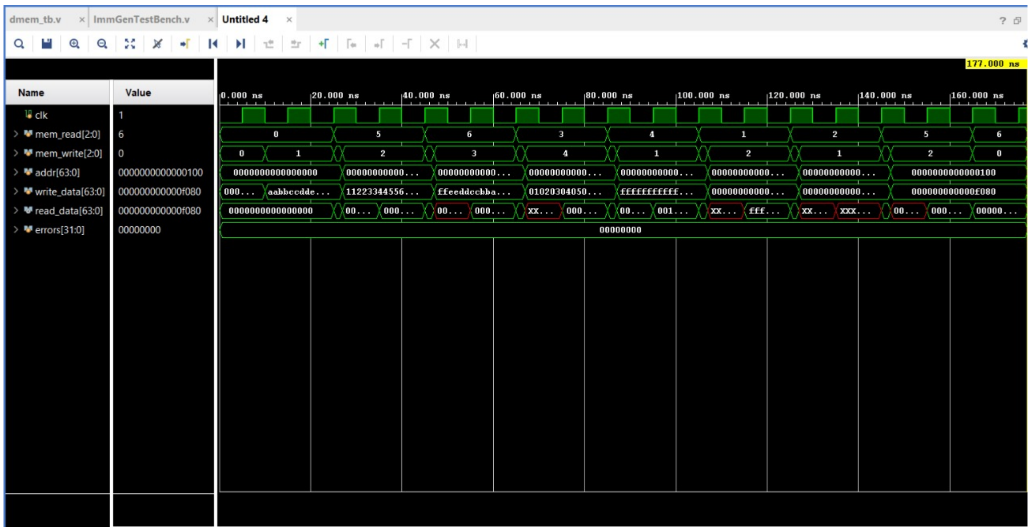


Figure 7: Simulation output for DMEM

fixed by restructuring the control signal assignment logic.

Simulation Summary

Timing diagrams and testbench waveforms demonstrate correct signal transitions and data propagation across all major modules. These include:

- ALU operations (`ADD`, `SUB`, `AND`, `OR`, etc.)
- Instruction memory fetch and PC increment
- Immediate generation and control signal decoding
- Data memory load/store operations

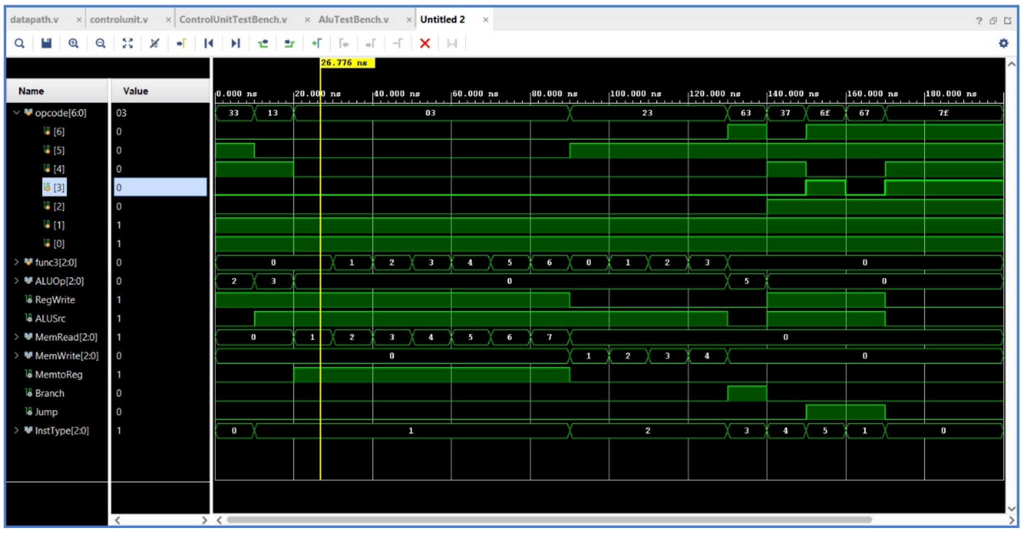


Figure 8: Control Unit waveform

Overall, the early simulation and synthesis results confirm that the individual modules and their integrated datapath operate as expected. The next focus will be on addressing synthesis resource optimization, implementing a multicycle control FSM, and extending the instruction set to include the M and F extensions.

4. Challenges and Observations

4.1 Revisiting Verilog and Design Fundamentals

At the beginning of the project, we faced some difficulty recalling Verilog syntax and structural design conventions. Although both of us had prior exposure, we had not implemented a large-scale processor design before, which required us to revise Verilog such as procedural blocks, signal assignment types, and module interfacing. We spent time reviewing these fundamentals and standardizing our design conventions to ensure consistent module interfaces, and signal widths across all components.

4.2 Memory Initialization and Data Loading

Another significant learning point was related to instruction and data memory initialization. Initially, we were unsure how to include program data in the memory arrays during simulation or synthesis. After consulting documentation and testing different methods, we learned that the `$readmemh` system task can be used to load program or data contents directly into the instruction and data memories, even during synthesis. This allowed us to test various instruction sequences and memory operations efficiently without manual intervention.

4.3 Module Integration and Debugging

Although each module worked correctly in isolation, we encountered multiple integration challenges when combining them into the top-level `datapath.v`. These included signal width mismatches, inconsistent control signal names, and interface definition conflicts between modules developed individually. We resolved these issues through waveform-based debugging, iterative simulation, and careful reorganization of interconnections. This process provided valuable experience in modular hardware design and top-level system integration.

4.4 Control Unit Logic Issue

A specific problem was identified within the `controlunit.v` module, where the processor always executed load-type instructions regardless of the opcode. Upon inspection, it was found that this behavior resulted from hardcoded control signals defaulting to the load instruction pattern. The issue was corrected by implementing a structured control flow that dynamically assigns control signals based on opcode decoding. After this fix, the control unit correctly handled arithmetic, logic, and branch instructions in simulation.

4.5 Synthesis Issue: Unexpected LUT Growth when Integrating Modules

During integration and synthesis of the single-cycle top-level datapath, we observed that the overall LUT usages exceed acceptable limits, despite individual modules (ALUControl.v, ImmGen.v, alu.v, controlunit.v, datapath.v, dmem.v, imem.v, pc.v, regfile.v) synthesizing within constraints when compiled separately. Preliminary investigation suggests that instruction and data memories which were inferred as BRAMs during isolated synthesis are getting implemented as LUT in the full top-level synthesis, leading to a large increase in LUT count.

Evidence and observations:

- Each module synthesizes under resource limits in isolation; the integrated top-level synthesis report shows a large rise in LUT utilization.
- Vivado synthesis reports indicate memory inference differences between module-level and top-level runs (some memory arrays are reported as LUT in the top-level run).
- No functional differences were detected in simulation, which means this is a synthesis/resource-mapping issue rather than a functional error.

Probable causes:

- Differences in synthesis optimizations at top-level by vivado cause certain modules to be inferred as LUT instead of BRAM .

- Multiple instantiations or unintended replication caused by how the top-level connects submodules.
- Use of constructs or parameters that does not allow BRAM inference like non-constant sizes, unsupported initialisation.

5. Next Steps

Having completed and verified the single-cycle 64-bit RISC-V processor, our next goal is to extend the design into an optimized and feature-rich architecture. We plan to combine the next three logical development phases—multicycle execution, multiplication/division extension, and floating-point integration—into a single unified implementation phase.

5.1 Phase 2–4: Multicycle Design, M Extension, and FPU Integration

The combined phase will include the following major objectives:

- **Multicycle RISC-V Processor:** Modify the datapath to reuse functional units across multiple clock cycles, thereby reducing hardware duplication and improving timing closure. Implement a finite-state machine (FSM)–based control unit to manage instruction sequencing across fetch, decode, execute, memory, and write-back stages. Validate the design through simulation using the same test programs from the single-cycle phase to ensure functional equivalence.
- **M Extension (Multiplication and Division):** Integrate hardware to implement RISC-V's M extension, supporting integer multiplication and division instructions. The design will either use a simple iterative algorithm for area efficiency or a pipelined multiplier for better throughput, depending on synthesis results and FPGA resource availability.
- **FPU Integration (F Extension):** Implement or integrate a basic IEEE-754 compliant single-precision Floating-Point Unit (FPU) supporting addition, subtraction, and multiplication operations. Extend the control unit to decode floating-point opcodes and manage register file access for floating-point operations. Functional testing will be carried out with mixed integer and floating-point programs.

5.2 Additional Enhancement: Cache Subsystem (Tentative)

If time permit, we plan to prototype a simple memory hierarchy by introducing L1 and L2 caches.The primary objective will be to demonstrate reduced average memory access time and validate correct cache coherence behavior for load/store operations.