ALU Design Project Report

Submitted By: PRITHVI SAHA

Employee ID: 6094

# 1. Brief Introduction

The Arithmetic Logic Unit (ALU) is a crucial combinational block in any digital computing system. It forms the core of the datapath in CPUs, microcontrollers, and digital signal processors, where it performs mathematical calculations and logic-based decision-making. Whether it's executing an addition instruction, evaluating a conditional branch, or shifting data for alignment, the ALU is central to the functioning of modern hardware.  
  
This project focuses on the design and verification of a parameterized ALU in Verilog, a widely used hardware description language (HDL) for modeling and simulating digital circuits. The ALU supports a comprehensive set of operations, including:  
- Arithmetic operations: Unsigned and signed addition, subtraction, increment, decrement, comparison, and compound arithmetic shifts with multiplication.  
- Logical operations: AND, OR, XOR, NOT, NAND, NOR, XNOR.  
- Bitwise operations: Shift left/right and rotate left/right, with flexible operand control.  
  
The ALU accepts two configurable-width operands (OPA, OPB), a 4-bit operation code (CMD), and several control signals like MODE, CIN, and INP\_VALID to guide the functional behavior. The MODE input allows the ALU to switch between arithmetic and logical processing dynamically, while flags such as Carry Out (COUT), Overflow (OFLOW), and comparator flags (G, L, E) provide additional outputs essential for control decisions in sequential systems.  
  
Beyond design, this project emphasizes comprehensive functional verification, which is critical in modern digital design workflows. A structured Verilog testbench has been developed to:  
- Apply randomized and directed test stimuli to the ALU.  
- Verify expected vs. actual outputs for every supported instruction.  
- Observe flag behavior under boundary and corner cases (e.g., overflow conditions).  
- Analyze signal transitions via waveform visualization tools.  
  
Verification results are used to confirm that the ALU operates correctly under all command combinations and edge scenarios. This approach ensures that the design is not only functionally correct, but also robust, synthesizable, and ready for hardware implementation or future integration into larger digital systems.

# 2. Objectives

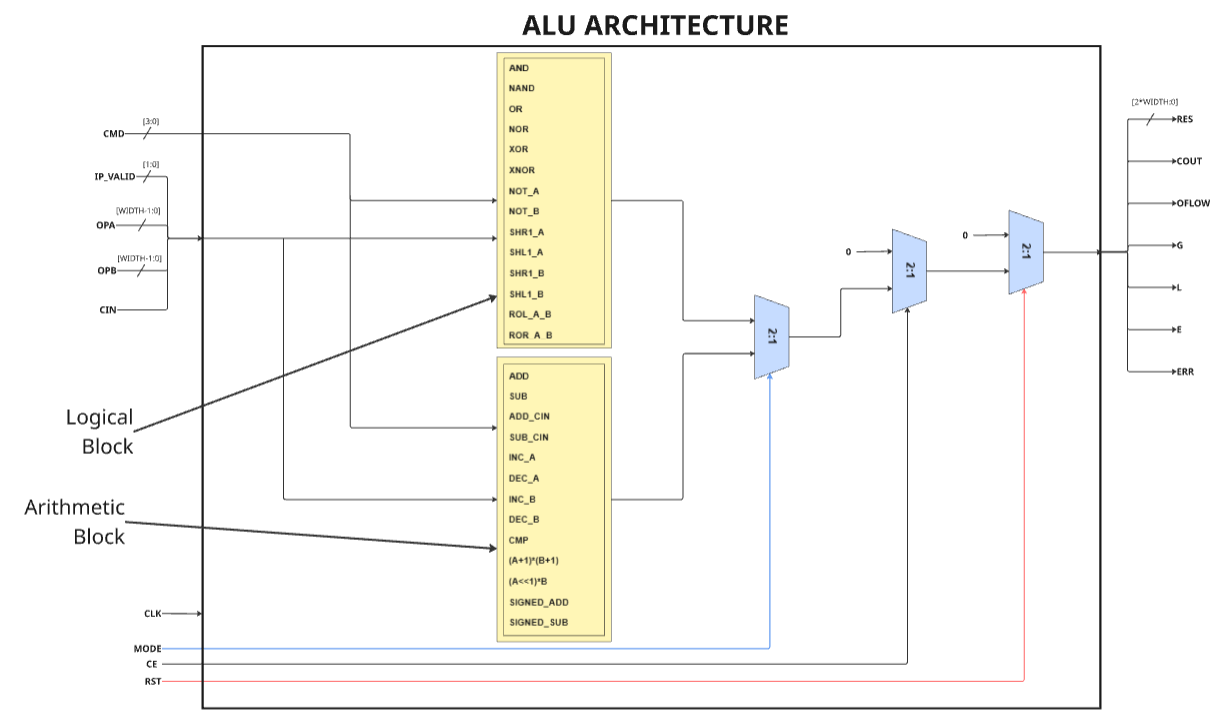
The primary objective of this project is to design and verify a robust, modular Arithmetic Logic Unit (ALU) using Verilog HDL. The ALU should be capable of performing a wide range of arithmetic and logical operations while supporting configurability and status flag generation.  
  
Specific objectives include:  
  
Design Implementation:  
- Develop a Verilog-based ALU module supporting parameterized data width.  
- Implement a variety of operations including unsigned/signed addition and subtraction, increment, decrement, comparison, and logic-based operations such as AND, OR, XOR, and shift/rotate functions.  
- Integrate functional control via opcode (CMD) and mode selector (MODE) for dynamic operation switching between arithmetic and logical domains.  
- Generate correct status outputs including carry out, overflow, comparator flags (G, L, E), and error flags under invalid conditions.  
  
Design Verification:  
- Build a comprehensive Verilog testbench to stimulate the ALU with valid and edge-case inputs.  
- Apply both directed and randomized test vectors to ensure extensive coverage of operation space.  
- Capture waveform outputs to verify correct timing, output values, and status flag behavior for each instruction.  
- Ensure testability, determinism, and debug-friendly simulation outputs to accelerate the validation process.  
- Confirm that the ALU behavior aligns with functional specifications under all conditions, including invalid or corner-case scenarios.

# 3. Architecture

The ALU architecture consists of several major components that handle data input, operation control, and result generation. The main inputs include two operands (OPA and OPB), a carry-in bit (CIN), and several control signals such as CLK, RST, CE, INP\_VALID, CMD, and MODE. The CMD signal determines the specific operation to perform, and the MODE bit selects whether it is arithmetic or logical in nature.

The ALU computes results combinationally based on the opcode and operands, and generates output signals such as RES (result), COUT (carry out), OFLOW (overflow), G/L/E (comparator flags), and ERR (error detection for invalid command usage). All these outputs are crucial for validating correctness and making decisions in processor pipelines.

**3.1 ALU Architecture**



**3.3 Design Architecture**

The ALU design architecture is composed of two main internal functional blocks: the Arithmetic Block and the Logical Block. These blocks are selected based on the operation mode (MODE signal), and are controlled by the opcode (CMD). Each block supports a subset of operations relevant to its function:

* The **Logical Block** handles operations such as AND, NAND, OR, NOR, XOR, XNOR, NOT (on A and B), single-bit shifts (SHR, SHL) and rotate operations (ROL\_A\_B, ROR\_A\_B).
* The **Arithmetic Block** supports ADD, SUB, operations with carry-in (ADD\_CIN, SUB\_CIN), increment/decrement on A or B, compare, signed/unsigned ADD and SUB, and compound expressions involving shifts and multiplication.

The overall architecture is structured such that inputs — including the operands (OPA, OPB), opcode (CMD), carry-in (CIN), and control signals — are distributed to both blocks. The decision of which block processes the inputs is governed by the MODE signal. Logical operations are executed when MODE is 0, and arithmetic operations are executed when MODE is 1.

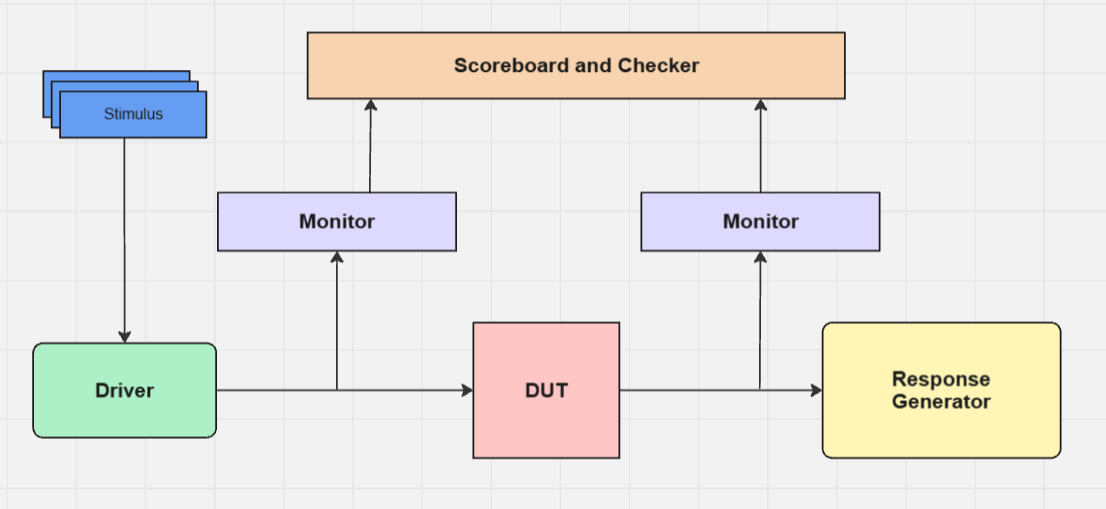
Both the arithmetic and logical blocks process the inputs in parallel, and the relevant output is selected using a set of 2:1 multiplexers based on the MODE and CMD values. The final selected output is passed to the RES line and status flag generators.

Status flags such as **COUT**, **OFLOW**, **G**, **L**, **E**, and **ERR** are also computed within this architecture:

* **COUT** and **OFLOW** are primarily related to arithmetic operations.
* **G**, **L**, and **E** are result comparisons between operands.
* **ERR** is flagged for invalid logical shift/rotate operations or malformed input configurations.

This modular approach not only simplifies operation decoding but also ensures clean segregation of arithmetic and logical pathways, enabling optimized synthesis and easier debugging.

**3.2 Testbench Architecture**

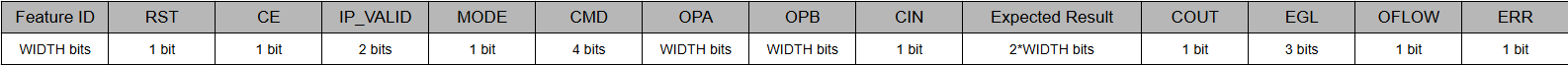


**3.2 Testbench Architecture**

The provided diagram represents a structured testbench architecture designed to verify the functionality of the Verilog-based ALU. This environment uses a self-checking mechanism where expected outputs are included in the stimulus itself, eliminating the need for a separate response generator. Here's how each block functions in this context:

1. **Stimulus (with Expected Response)**

The Stimulus block is responsible for generating test vectors — combinations of input operands (OPA, OPB), control signals (CMD, MODE, CIN, etc.), and expected outputs (EXP\_RES, EXP\_COUT, EXP\_OFLOW, etc.).



**3.3 Packet Architecture**

Each stimulus "packet" includes:

* Input signals required to drive the DUT.
* The expected result and flags precomputed externally.
* Validity control (RST, INP\_VALID, CE, MODE) to indicate the relevance of data.

This embedded approach simplifies the testbench by removing the need for a separate response generator block.

1. **Driver**

The Driver reads stimulus packets and applies the contained input signals to the DUT. It:

* Drives operands and control signals into the DUT in a valid and timed manner.
* Synchronizes with the DUT using clock (CLK) and reset (RST) signals.
* Ensures that data is injected properly according to the timing and interface protocol.

1. **DUT (Design Under Test)**

This is the Verilog ALU implementation, which processes input data based on the opcode and mode, then produces:

* Output result (RES)
* Carry out (COUT)
* Overflow (OFLOW)
* Comparator outputs (G, L, E)
* Error signal (ERR)

1. **Monitors** There are two monitors in this environment:

* **Input Monitor**: Captures all the stimulus packets as they are applied to the DUT.
* **Output Monitor**: Observes the DUT’s actual outputs.

Both monitors extract the relevant data and send it to the Scoreboard for comparison.

1. **Scoreboard and Checker**

The Scoreboard receives:

* Actual outputs from the Output Monitor.
* Expected outputs embedded within the original stimulus packet via the Input Monitor.

The Checker within the Scoreboard performs:

* Comparison of actual vs. expected outputs (result, flags, error).
* Pass/fail logging, including mismatches.
* Optional debug output or waveform triggers for failed cases.

Because the expected response is already part of the input, there is no need for a separate response generator, reducing complexity and improving test speed.

# 4. Working

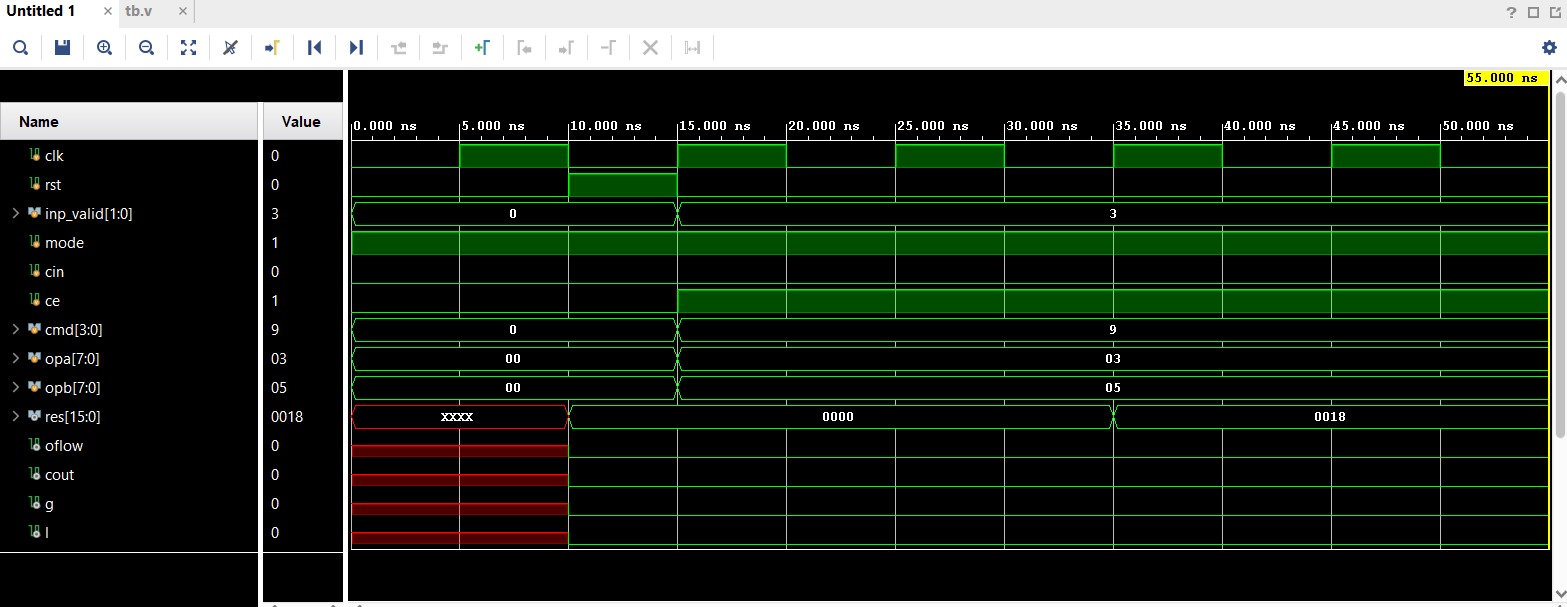
The ALU operates as a fully combinational unit, meaning the output is updated immediately in response to changes in the input operands or control signals, without waiting for a clock edge. Here's how the ALU processes data step-by-step:

1. **Input Reception**:
   * Two operands (OPA and OPB), carry-in (CIN), and the command opcode (CMD) are provided as inputs.
   * The mode signal (MODE) determines whether the operation is arithmetic (MODE = 1) or logical (MODE = 0).
   * The input validity flag (INP\_VALID) ensures that both operands are valid before any operation is processed.
2. **Operation Selection**:
   * Based on the CMD and MODE, the control logic decodes which internal operation should be executed.
   * If the operation is logical (e.g., AND, OR, XOR), the logical block is activated.
   * If the operation is arithmetic (e.g., ADD, SUB, INC), the arithmetic block is used.
3. **Parallel Processing**:
   * Both arithmetic and logical blocks receive the input data simultaneously.
   * The actual operation is performed in parallel by both blocks, but only the output from the relevant block (as chosen by MODE) is selected using multiplexers.
4. **Output Selection**:
   * The output of the active block (logical or arithmetic) is routed through a multiplexer to the final output bus (RES).
   * Along with the result, various status flags such as COUT, OFLOW, G, L, E, and ERR are computed depending on the result and command context.
5. **Flag Update**:
   * **COUT** (carry out) and **OFLOW** (overflow) flags are generated primarily during arithmetic operations.
   * **G**, **L**, and **E** are comparators based on the values of OPA and OPB.
   * **ERR** is raised when invalid input patterns are detected, especially for logical rotate operations.
6. **Combinational Behavior**:
   * Since the ALU is designed combinationally, it responds immediately to input changes without relying on clocking.
   * However, control signals like CLK, CE, and RST are used for synchronization and enabling in an integrated environment (e.g., datapath or pipelined processor).

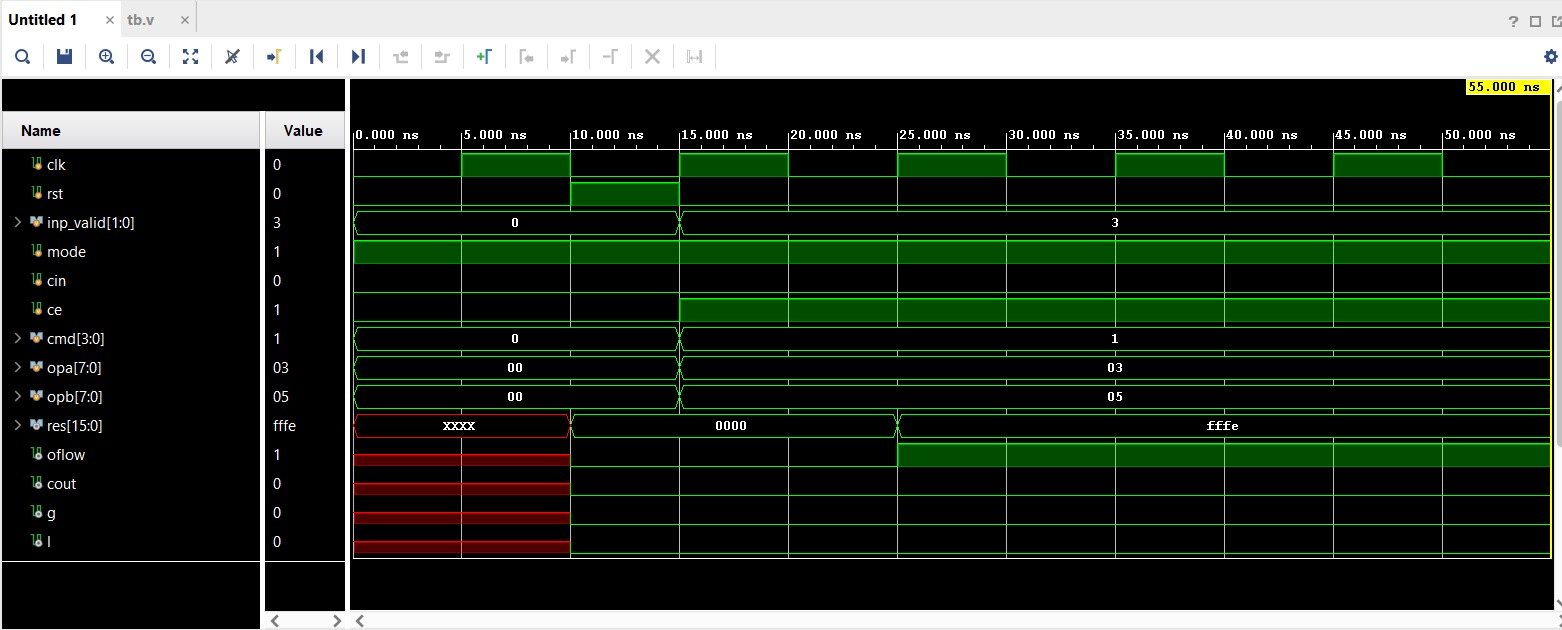
This approach ensures low-latency computation suitable for integration into larger processor cores or datapaths, and simplifies verification due to its deterministic behavior.

# 5. Result

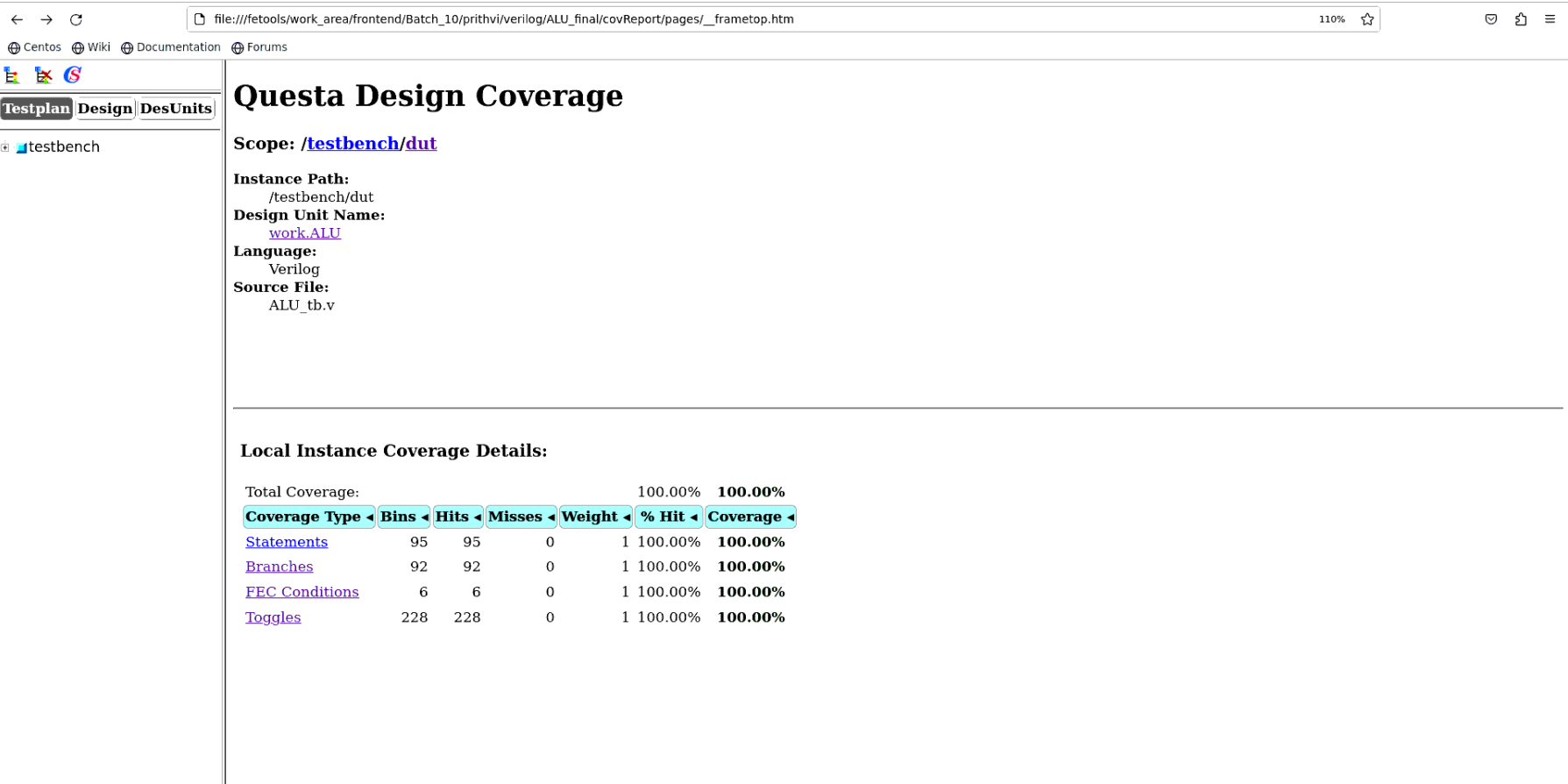
Multiplication Operation After 2 cycle delay:



Other Operations after 1 cycle delay:



Coverage Report:



# 6. Conclusion

The Verilog-based ALU was successfully designed, simulated, and verified against all defined functional requirements. The ALU supports a rich instruction set spanning both arithmetic and logical domains, with correct output generation and real-time status flag updates.  
  
The integration of a rigorous verification environment ensured high confidence in the functional correctness of the design. The testbench stimulated the ALU with a diverse set of inputs, including edge cases, and waveform inspection confirmed the correct response and flag assertions across all tested conditions.  
  
The design is fully synthesizable, modular, and ready for hardware deployment or integration into larger processor systems. With proper parameterization and clean coding practices, the ALU can be easily extended or reused in future projects.

# 7. Future Improvements

- Support for more complex operations such as division and modulus.  
- Integrate formal verification techniques for deeper correctness proof.