**Project – Parameterized ALU**

This project involves the design and verification of a parameterized Arithmetic and Logic Unit (ALU), with a default operand width of 8 bits. The ALU is capable of performing a variety of simple unsigned arithmetic and logical operations, as well as two special signed arithmetic operations. Operation selection is determined by a mode signal (0 for logical operations, 1 for arithmetic operations) and a 4-bit command input that specifies the desired operation.

The ALU is synchronous, with outputs available one clock cycle after the input and control signals are applied. For multiplication operations, a latency of two clock cycles is introduced. The design supports parameterization for flexibility in bit-width scalability.

The development process began with a study of ALU architecture, followed by the formulation of a suitable micro-architecture. This was followed by RTL design using Verilog, code linting to ensure style and syntax correctness, functional simulation, comprehensive testing, and coverage analysis to verify the completeness and correctness of the implementation.

**Objectives:**

* Study and analyze different ALU designs to identify the required operations, control logic, and timing behaviour.
* Design a scalable and flexible micro-architecture supporting various bit-widths.
* Create a synthesizable and efficient RTL implementation of the ALU in Verilog.
* Apply linting tools to identify and correct potential coding style violations and syntactic issues.
* Write directed and corner-case testbenches to ensure correct behaviour across all command inputs and mode combinations.
* Use Code coverage tools to ensure that all parts of the design have been adequately exercised by tests.

**Design Architecture:**

The ALU is designed as a parameterized, synchronous digital module with an 8-bit default operand width. It supports a set of arithmetic and logical operations based on a 4-bit command (CMD) and a 1-bit mode signal that selects between logical (0) and arithmetic (1) operations.

**Inputs:**

* clk: System clock for synchronous operation.
* ce: Clock enable signal. When low, the ALU holds its current state.
* reset: Active-high asynchronous reset that clears internal registers and sets all outputs to default (typically 0).
* opa and opb: Operands for the operation, each parameterizable (default 8 bits).
* Inp\_valid: 2-bit input validity signal:
  + MSB corresponds to operand B validity.
  + LSB corresponds to operand A validity.

Operations are only initiated when respective bits are high, else error is triggered.

* mode: 1-bit signal to select operation type:
  + 0 – Logical operations
  + 1 – Arithmetic operations
* cmd: 4-bit command to specify the operation to be executed.

**Outputs:**

* res: Result of the selected operation (operand width + 1 by default, double the width if multiplication).
* oflow: Indicates Overflow in the Output in ADD, ADD\_CIN, SUB, SUB\_CIN, SP\_1, SP\_2.
* cout: Indicates Carry out or Borrow out in ADD, ADD\_CIN, SUB, SUB\_CIN, SP\_1, SP\_2
* g,l,e: Indicates the magnitude of operand A against operand B, in CMP,SP\_1,SP\_2.
* err: Indicates if there is any error in the input validity to the command input, overflow of operand B in shift amount (1’b1 in either positions 4,5,6,7 of opb).
* neg: Indicates if the output is negative, for SP\_1 and SP\_2.
* zero: Indicates if the output is zero, for SP\_1 and SP\_2.

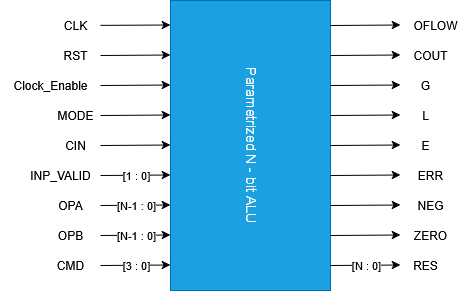


Figure : ALU Block

**Timing Behaviour:**

* The ALU responds to input and control signals on the rising edge of the clock, only when clk\_en is high and required inputs are valid.
* The result of most operations is available after 1 clock cycle.
* For multiplication operations, output is available after 2 clock cycles, managed internally via delay logic.

**Operations**

* Unsigned Arithmetic:
  + ADD: Unsigned Addition between opa and opb.
  + SUB: Unsigned Subtraction between opa and opb.
  + ADD\_CIN: Unsigned Addition with Carry IN and between opa and opb.
  + SUB\_CIN: Unsigned Subtraction with Borrow IN and between opa and opb.
  + INC\_A: Increment opa by 1.
  + DEC\_A: Decrement opa by 1.
  + INC\_B: Increment opb by 1.
  + DEC\_B: Decrement opa by 1.
  + CMP: Compares opa and opb.
  + ADD\_MUL: Increments opa and opb by 1 and then multiplies them.
  + SH\_MUL: Shifts opa by 1 and then multiplies opa and opb.
* Signed Arithmetic:
  + SP\_1: Signed addition and comparison of opa and opb.
  + SP\_2: Signed subtraction and comparison of opa and opb.
* Logical:
  + AND: ANDs the two operands.
  + NAND: NANDs the two operands.
  + OR: ORs the two operands.
  + NOR: NORs the two operands.
  + XOR: XORs the two operands.
  + XNOR: XNORs the two operands.
  + NOT\_A: Inverts opa.
  + NOT\_B: Inverts opb.
  + SHR1\_A: Shifts opa by 1 to the right.
  + SHL1\_A: Shifts opa by 1 to the left.
  + SHR1\_B: Shifts opb by 1 to the right.
  + SHL1\_B: Shifts opb by 1 to the left.
  + ROL\_A\_B: Rotates opa by log2(width) bits of opb to the left. If the opb bits from the MSB to the log2(width) are active, the error is asserted.
  + ROR\_A\_B: Rotates opa by log2(width) bits of opb to the right. If the opb bits from the MSB to the log2(width) are active, the error is asserted.

**Working:**

**1. Input Phase:**

At every positive edge of the clock (clk), the ALU samples the input signals only if clk\_en is high, inp\_valid indicates valid operands, and the asynchronous active-high reset is not asserted.

* inp\_valid[0] controls acceptance of operand A.
* inp\_valid[1] controls acceptance of operand B.
* If inp\_valid = 2’b11 where only one input is required, err is active and the output becomes 0.
* mode selects between logical (0) and arithmetic (1) operations.
* CMD defines the specific operation to execute.
* When reset is high, internal registers and outputs are reset to zero immediately.

**2. Operation Phase:**

Once valid inputs are captured, the ALU decodes the mode and CMD to direct the operands through appropriate internal functional blocks:

* Logical Operations Block (mode = 0):
  + Performs operations like AND, OR, XOR, NOT, SHR, SHL, ROL, ROR using bitwise logic and shifts.
* Arithmetic Operations Block (mode = 1):
  + Performs operations like ADD, SUB, INC, DEC, CMP, ADD\_CIN, SUB\_CIN, and special arithmetic operations like:
  + ADD\_MUL: Increment both operands, then multiply.
  + SH\_MUL: Shift A left by 1, then multiply with B.
* Signed Arithmetic Sub-block:
  + SP\_1: Performs signed addition and comparison of opa and opb.
  + SP\_2: Performs signed subtraction and comparison of opa and opb.

**3. Output Phase:**

For all single-cycle operations, the output is available at the next positive edge of the clock (1-cycle latency).

For multi-cycle operations (ADD\_MUL, SH\_MUL), an intermediate computation stage is inserted, and the result is output after 2 positive clock edges from the input sampling.

The output result is stored in register res.

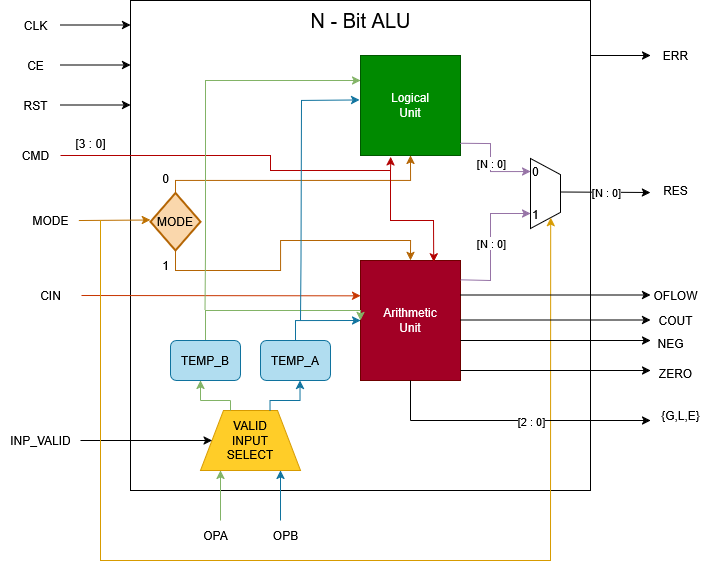


Figure : ALU Expanded Block

**Test-Bench Architecture:**

The testbench is designed as a modular verification environment to functionally validate the ALU design. It follows a structured approach using stimulus files, self-checking mechanisms, and result logging to ensure correctness and traceability of test cases.

**Inputs:**

* **stimulus.txt:** A text file containing a sequence of 57-bit wide test vectors. Each line corresponds to a single test case with the following content:

**TEST VECTOR CONTENT:**

|  |  |  |
| --- | --- | --- |
| Content Name | Sliced part | Description |
| FEATURE\_ID | curr\_test[56 : 49] | Unique identification to each test case. |
| EXPECTED\_RES | curr\_test[23 : 8] | Expected Result (9 bits) |
| EXP\_COUT | curr\_test[7] | Expected Carry Out |
| EXP\_OFLOW | curr\_test[3] | Expected Overflow |
| EXP\_ERR | curr\_test[2] | Expected Error Trigger |
| EXP\_NEG | curr\_test[1] | Expected Negative, based on the output, only for SP\_1 and SP\_2. |
| EXP\_ZERO | curr\_test[0] | Expected zero, based on the output, only for SP\_1 and SP\_2. |
| EXP\_EGL | curr\_test[6 : 4] | Gives the Comparison output of the operands, exclusive to CMP, SP\_1 and SP\_2. |
| INP\_VALID | curr\_test[48 : 47] | Indicates the validity of the two inputs, MSB gives the validity of OPA and LSB for OPB. |
| OPA | curr\_test[46 : 39] | Gives the first input operand (8 bits) |
| OPB | curr\_test[38 : 31] | Gives the second input operand (8 bits) |
| CMD | curr\_test[30 : 27] | Gives the Operation to perform on the respective Operand |
| MODE | curr\_test[24] | Indicates Operation performed is Logical(0) or Arithmetic(1). |
| CE | curr\_test[25] | Gives the Clock Enable, responsible for controlling the synchronous behaviour of the ALU. |
| CIN | curr\_test[26] | Gives the Carry in for operations, ADD\_CIN and SUB\_CIN. |

**Testbench Components:**

* **Clock and Reset Generation:**
  + Clock (clk) and asynchronous reset (rst) signals are generated internally within the testbench and are not controlled via stimulus input.
* **Driver Task:**
  + Extracts individual fields from the current 57-bit curr\_test vector.
  + Drives the extracted input values (operands, mode, CMD, INP\_VALID, etc.) to the ALU (DUT).
  + Also sends Feature ID and Expected Output to the Scoreboard.
* **DUT (Design Under Test):**
  + Instance of the ALU module under test.
  + Receives stimulus from the driver and generates output accordingly.
* **Monitor Task:**
  + Observes both inputs and outputs of the DUT.
  + Packs DUT outputs into a 17-bit response signal.
  + Sends this packed response to the Scoreboard for checking.
* **Scoreboard Task:**
  + Compares the actual DUT response with the expected output using case equality.
  + Generates a 29-bit SCB\_VAL output that includes:
    1. Feature ID
    2. Expected Output
    3. Actual Output
    4. Pass/Fail flag
  + Logs SCB\_VAL into an output file called result.txt for later analysis.

**Outputs:**

* **result.txt:** A text file where the Scoreboard stores test results, providing a record of:
  + Which test case ran
  + What output was expected and received
  + Whether the test case passed or failed

**Timing Behaviour:**

The testbench runs in a cycle-accurate manner, adhering to the ALU's latency:

* One clock cycle for most operations, whereas two cycles for multiplication
* Synchronization is handled within the testbench tasks to ensure valid comparison timing.

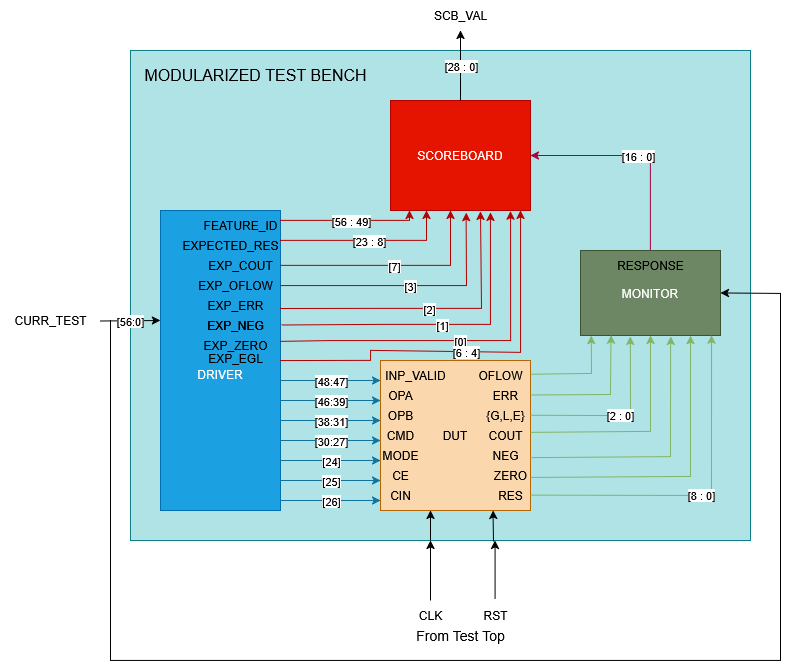


Figure : Test Bench Architecture

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | |
| **CURR\_TEST\_CASE (57 bit)** | | | | | | | | | | | | | | |
| **Feature ID** | **INPUT VALID** | **OPA** | **OPB** | **CMD** | **CIN** | **CE** | **MODE** | **Expected RES** | **COUT** | **EGL** | **OV** | **ERR** | **NEG** | **ZERO** |
| **[56:49]** | **[48:47]** | **[46:39]** | **[38:31]** | **[30:27]** | **26** | **25** | **24** | **[23:8]** | **7** | **[6:4]** | **3** | **2** | **1** | **0** |

Figure : Input Packet

**Quality of Code Assessment:**

* **Lint:**

Linting is the process of analyzing RTL code (typically written in Verilog/SystemVerilog) for common coding errors, bad practices, and potential issues that could lead to simulation mismatches or synthesis problems.

In the context of this project, linting was used to:

* Detect syntax and semantic issues (e.g., unused signals, uninitialized variables, mismatched widths).
* Enforce coding standards, improving consistency and readability.
* Identify problematic constructs that could affect synthesis, like inferred latches or unintended blocking behaviour.
* Catch race conditions and unintentional combinational loops early in the development cycle.

By resolving lint errors and warnings, the ALU design was made cleaner, more maintainable, and less error-prone.

* **Code Coverage:**

Code coverage evaluates how much of the RTL code is actually exercised during simulation. It is a critical measure to ensure that the testbenches are effective and comprehensive.

Types of coverage used:

* Line/Statement Coverage: Checks whether each line of RTL code has been executed at least once.
* Branch Coverage: Verifies that all possible decision paths (e.g., if-else, case statements) are taken during simulation.
* Toggle Coverage: Ensures that all bits in flip-flops and wires toggle (change state) during simulation.

For the ALU project, code coverage helped ensure:

* All command cases in the CMD decoder were tested.

Achieving high coverage metrics indicates that the test environment thoroughly verifies the design, reducing the risk of undetected bugs in production.

**Results:**

The ALU design was thoroughly tested for both regular arithmetic and logical operations, as well as selected corner cases to validate its robustness. These corner cases included scenarios such as:

* Clock enable (ce) set to 0 (disabling operation),
* Executing a MUL operation immediately after an ADD,
* Changing only operand values while keeping control signals constant,
* Varying only the CMD signal between cycles,
* Repeating the same operation with identical inputs.

The RTL code was lint-checked and synthesized using Xilinx Vivado, ensuring it was clean, synthesizable, and hardware-compatible. Functional simulation was performed using a structured Verilog testbench in Questa SIM, which enabled clear observation of input-output behaviour over time.

A total of 76 test cases were executed, covering standard use cases and edge conditions.

The waveform captured from simulation highlights the response of the ALU to each test vector, verifying correctness against expected outputs.

The waveform confirms that all tested scenarios, including edge cases, were handled correctly. Results from the scoreboard were logged to result.txt, showing feature ID, expected and actual outputs, and pass/fail status for each test case.

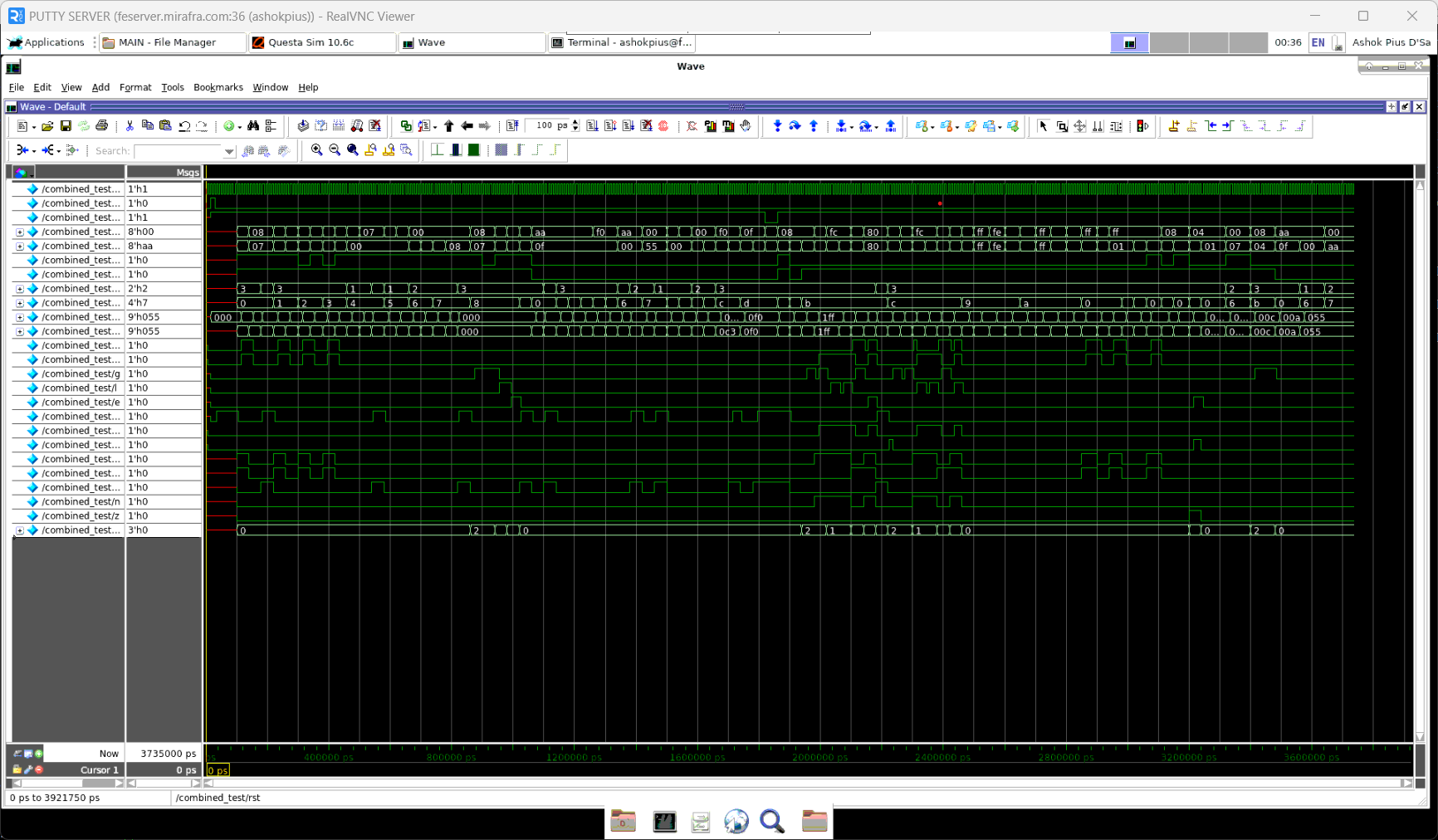


Figure : Questa SIM Simulation of ALU against 76 test cases

**Coverage Report:**

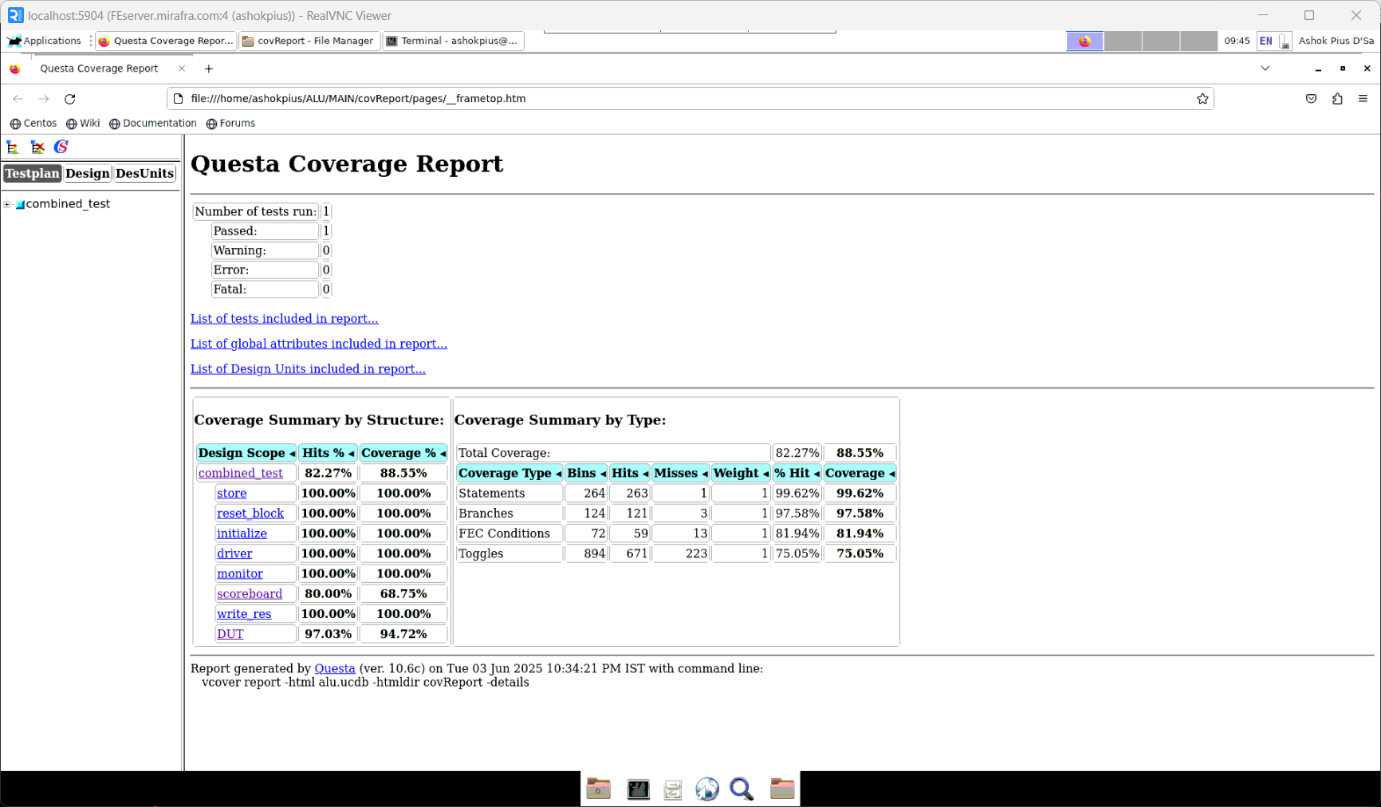
****

Figure : Coverage Report for the 76 test cases

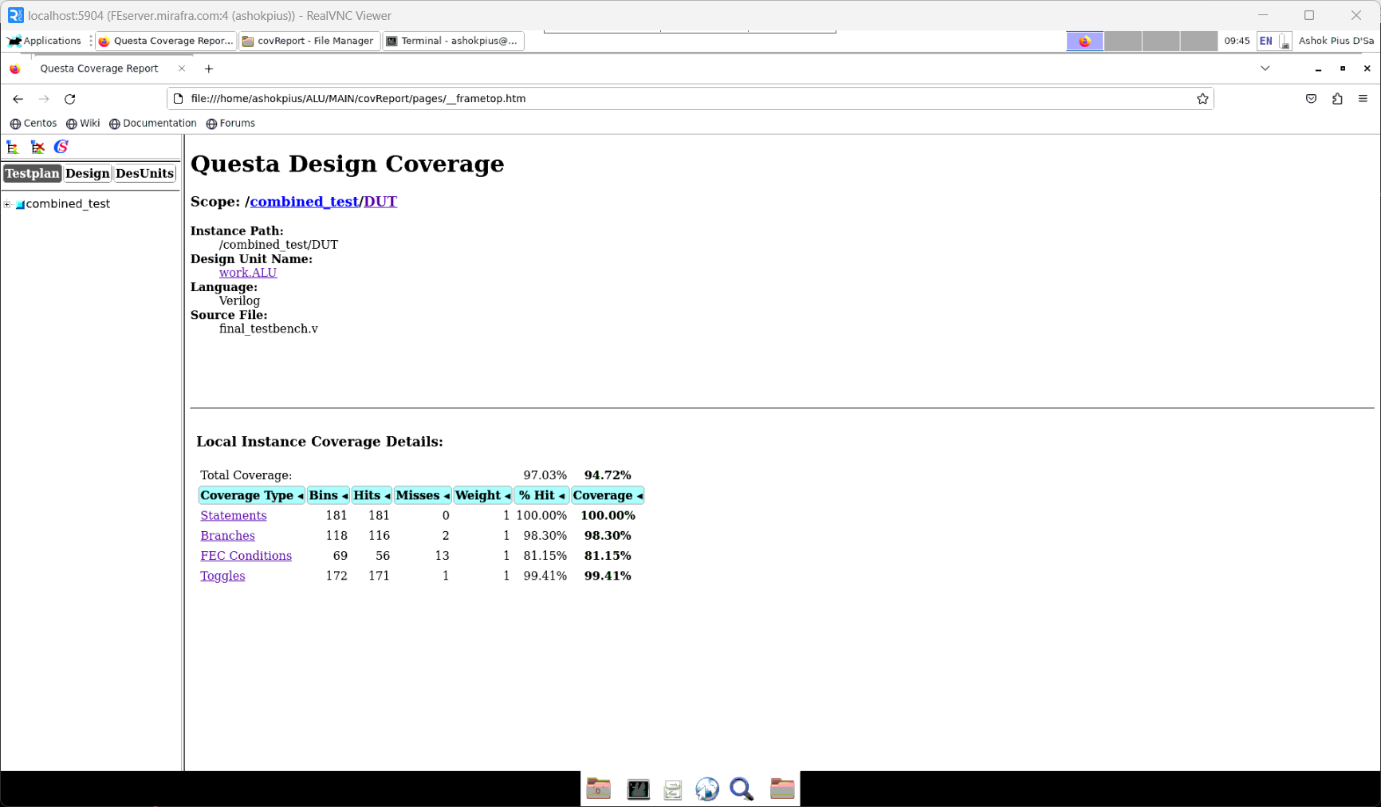
****

Figure : Expanded Format of DUT Coverage

**Conclusion:**

* Successfully designed a parameterized ALU with a default 8-bit width, supporting a wide range of arithmetic and logical operations.
* Implemented both unsigned and signed arithmetic operations, including special multi-cycle functions like ADD\_MUL and SH\_MUL.
* Developed a clean and synthesizable RTL design using Verilog.
* Employed linting to catch and resolve potential design flaws, improving code quality and maintainability.
* Created a comprehensive testbench to validate all supported operations and command combinations.
* Verified the timing behaviour and latency, ensuring single-cycle and multi-cycle operations met the design requirements.
* Achieved high code coverage across all logic branches and scenarios, ensuring thorough functional verification.
* Gained hands-on experience in a structured digital design flow, including architecture definition, RTL design, simulation, and quality assessment.
* Built a design that is reusable and extendable, making it suitable for integration in larger processor or SoC projects.

**Future Work:**

* Pipelining for Performance.
* Formal Verification.
* Power and Area Optimization.
* Support for Additional Operations.