ALU VERIFICATION PLAN

Sumanth

EMP ID : 6115

VERIFICATION DOCUMENT ALU

CHAPTER 1 – PROJECT OVERVIEW

1.1 ALU Introduction:

This project aims to design a flexible and high-performance Arithmetic Logic Unit (ALU) suitable for a variety of digital systems, ranging from simple embedded devices to complex processors. One of the core strengths of this ALU is its parameterized structure, allowing it to support different bit-widths such as 16, 32, 64, or even 128 bits. This scalability makes it adaptable to both low resource and high-performance environments. The ALU takes in two operands, OPA and OPB, and operates in two clearly defined modes arithmetic mode when MODE is set to 1, and logical mode when MODE is 0. Arithnetic mode supports 12 operations and Logical mode supports 15 operations covering everything from basic arithmetic like addition, subtraction, and multiplication to logical operations like AND, OR, XOR, and advanced functions such as bit rotations. The dual mode system efficiently leverages a 4-bit command input, which ensures logical separation and efficient decoding of operations.

To meet the demands of real-world system integration, the ALU includes intelligent control features. The INP_VALID signal helps manage scenarios where operands may arrive at different times, which is common in pipelined or asynchronous systems. In addition, a built-in timeout mechanism ensures the ALU doesn't stall indefinitely, enhancing reliability. The design also includes comprehensive status outputs such as carry detection, overflow flags, and comparison results like greater than, less than, and equal. These flags allow external systems like control units or processors to make decisions based on the outcomes of ALU operations, such as branching or exception handling.

Another highlight of this design is its robust error-handling capability. For operations like bit rotations, where operand B must follow strict rules, the ALU is equipped to detect and flag invalid inputs instead of failing silently. This proactive error detection simplifies debugging and integration, especially in complex systems. Overall, the project focuses on creating a reusable, scalable, and reliable ALU design that balances computational power with smart system-level features, making it highly suitable for modern digital applications..

Design Features:

- **Fixed Timeout**
  The 16-cycle timeout is hardcoded. If a system needs a shorter or longer wait, the design must be changed.
- **Mode-Dependent Command Codes**
  Same command code does different things in arithmetic and logical modes — this can lead to software mistakes.
- **Single Error Signal (ERR)**
  Only one ERR signal is used for all errors, so you can't tell if it was a timeout, invalid command, or input problem.
- **Limited Rotate Range**
  Rotate operations only support up to 8 positions — may not be enough for larger bit-widths.
- **Fixed Operand Priority**
  In case of timeout, the ALU always uses the latest operand — which may not match what some systems expect.
- **Wasted Resources**
  Even unused operations still take up hardware space due to the parameterized design — not ideal for small or resource-limited chips.

Advantages of ALU:

- **Flexible bit width support :**
  The design is scalable it can start as a Parameterized-bit ALU and expand to 32, 64, or even 128 bits without redesigning the core logic. This saves development time and reduces the chance of introducing new errors.
- **Extensive Functionality:**
  It supports a total of 25 operations (11 arithmetic and 14 logical), enabling everything from basic math to advanced bit manipulations like rotation reducing the need for additional external processing units.

- **Intelligent Input Management:**
  With an INP_VALID signal and built-in timeout mechanism, the ALU gracefully handles input synchronization. If inputs don't arrive in time, the timeout ensures the system doesn't freeze or hang
- **Status Flags:**
  Each operation generates status outputs such as carry, overflow, comparison results (greater, less, equal), and error indicators providing the system with rich feedback for better decision-making.
- **Power Efficiency:**
  Through a clock enable (CE) feature, the ALU can be disabled when idle. This minimizes power consumption, which is especially beneficial in battery-powered or energy-sensitive applications.
- **Built-in Error Detection:**
  The ALU is capable of catching invalid operation scenarios, particularly during rotate operations where improper input formats can lead to malfunction. This built-in check helps prevent such issues

Disadvantages of ALU:

- **Fixed Timeout Duration:**
  The input waiting mechanism is hardcoded to wait for exactly 16 clock cycles. This may not be ideal for all systems too slow for high-speed applications or too fast for slower ones and it can't be reconfigured.
- **Ambiguous Command Structure:**
  Some command codes serve multiple purposes depending on the selected mode (e.g., arithmetic vs logic). For instance, CMD = 0 could mean ADD or AND, which can easily lead to programming mistakes if not handled carefully
- **Generic Error Flag:**
  The ALU provides a single error signal for all error types, without specifying what went wrong. Whether it was a timeout, an invalid command, or a bad input, the system gets the same flag making debugging more difficult
- **Hardware Overhead:**

While feature-rich, the ALU's complexity means it consumes more logic resources. For applications with simple requirements, this design might be unnecessarily large and inefficient

- **Strict Rotate Input Rules:**
  Rotate operations have tight constraints for example, certain bits in operand B (like bits [7:4]) must be zero. These requirements can be tricky to meet in software and may lead to unintended errors if not handled properly

Use Cases of ALU:

- **Arithmetic Operations:**
  Handles essential arithmetic functions such as addition, subtraction, multiplication, and division. These are fundamental for tasks like incrementing counters
- **Logical Operations**
  Executes standard logic operations like AND, OR, XOR, and NOT. These are widely used for value comparisons, implementing decision-making structures, and applying bitwise masks in digital systems.
- **Bit Manipulation (Shifting & Rotation):**
  Supports left and right shifts, as well as bit rotations, which are vital in tasks such as data encoding, cryptographic algorithms, and efficient bit-level operations.
- **Data Comparison:**
  Compares two operands to determine equality, greater than, or less than relationships. This functionality supports conditional logic, loop control, and decision branching in both software and hardware.
- **Checksum and CRC Computation**
  Can be used to compute checksums or cyclic redundancy checks (CRC), which are widely applied in error detection for communication protocols and data transmission systems.
- **Control Signal Generation**
  Uses comparison results to generate control signals in FSMs (Finite State Machines) and controllers
- **Carry and Overflow Detection:**

Monitors for carry-out or arithmetic overflow, ensuring proper handling of signed and unsigned data and maintaining operation accuracy during calculations.

- **Executing CPU Instructions**

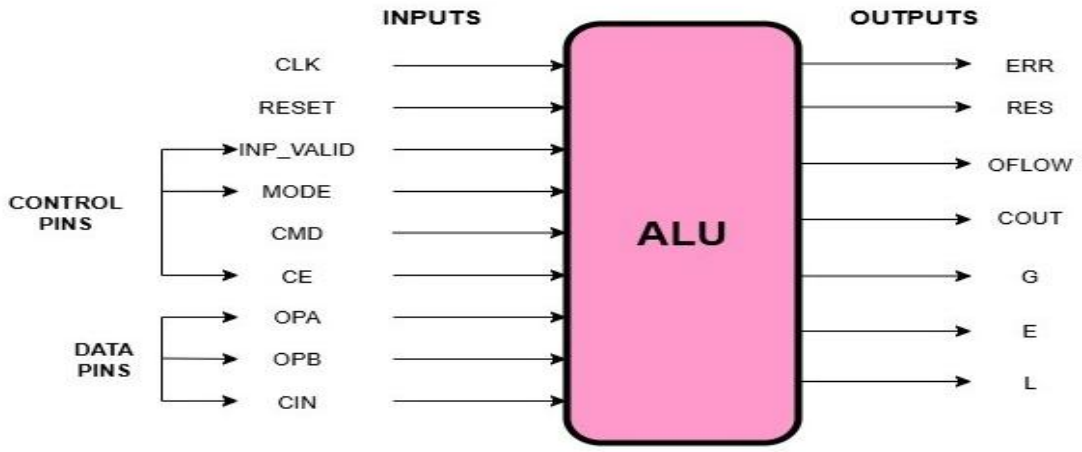
  The ALU is the core part of the execution stage of a CPU it carries out actual operations for instructions

1.2 Verification Objectives

- Review design documentation: analyze functional requirements, supported operations, inputs/outputs, operating modes and edge-case behaviors.
- Construct the verification plan.
- Develop functional coverage and assertion plan.
- Frame the testbench architecture for the alu design.
- Creation of template codes for testbench components
- Implement and enhance testbench for coverage, integrating the functional coverage and SystemVerilog assertions.
- Validating the functional correctness of the ALU, considering the corner cases as well.
- Validating the timing of the operations.
- Checking its robustness against errors.

1.2 DUT Interfaces:

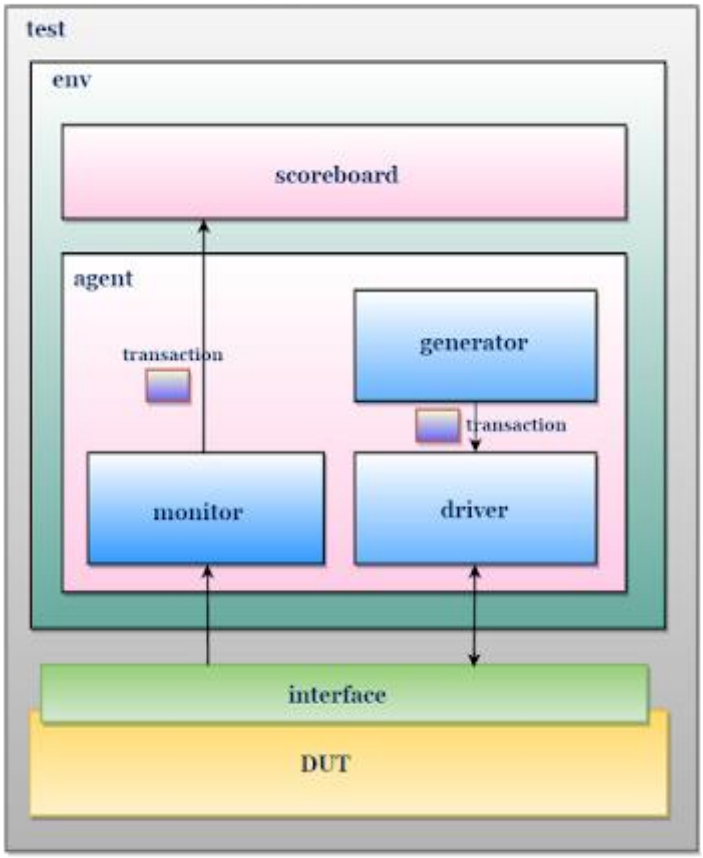
These are the signals present in the interface which is to be shared between the Test and Design.

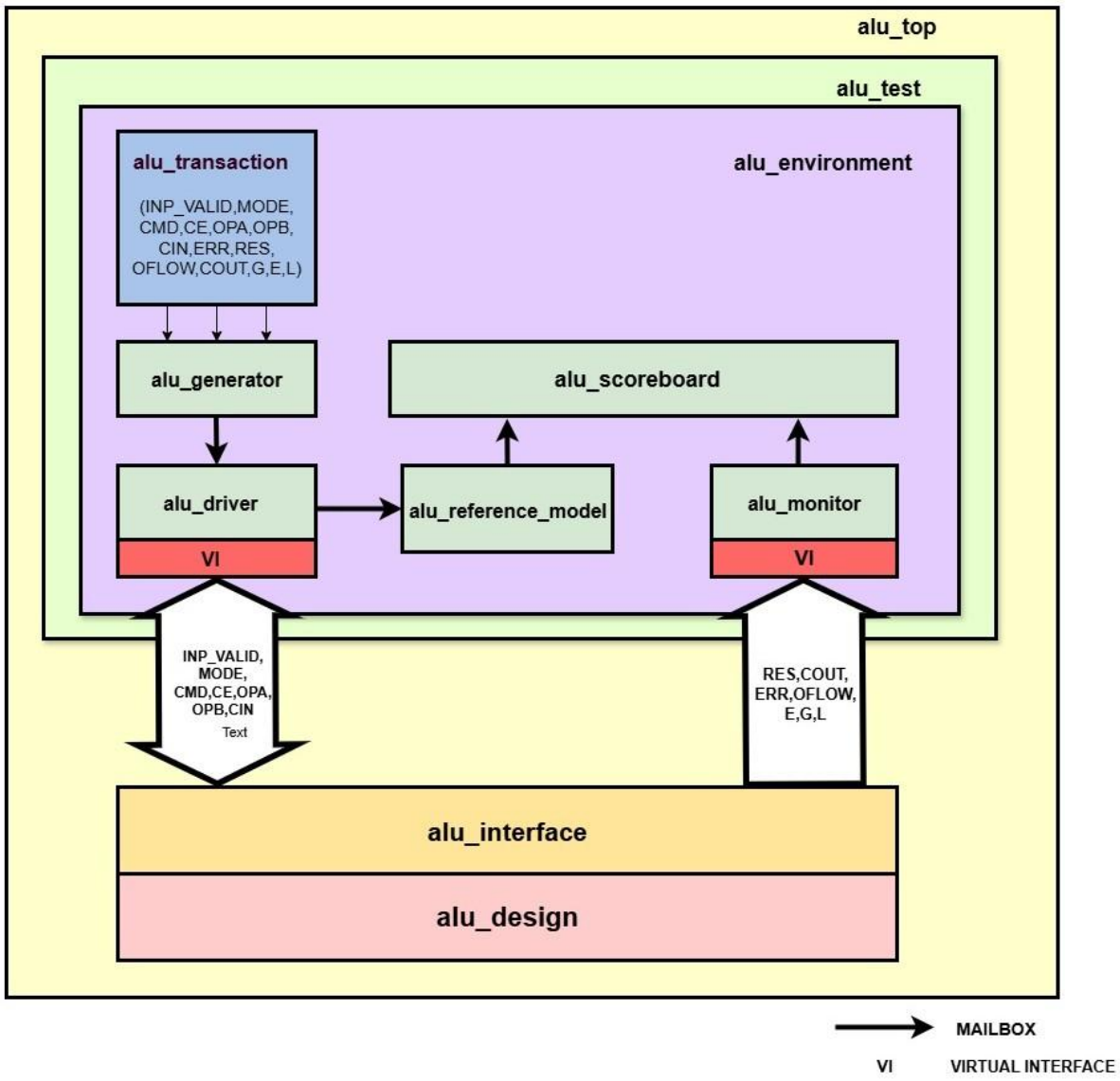


Single Name	Type	Size(bits)	Description
CLK	INPUT	1	Clock signal which is used for sample the values
RESET	INPUT	1	Asynchronous Active High RESET
CE	INPUT	1	Clock enable used to enables the design
INP_VALID	INPUT	2	Shows the Validity of the Operands(active high) MSB shows the validity of the OPB and LSB shows the validity for OPA.
MODE	INPUT	1	If the value is 1 the ALU is in Arithmetic Mode else it is in Logical Mode
CMD	INPUT	Parameterized	Commands for the Operation
OPA	INPUT	Parameterized	Operand 1
OPB	INPUT	Parameterized	Operand 2
CIN	INPUT	1	Carry in signal used for add_cin and sub_cin
ERR	OUTPUT	1	Active High error flag
RES	OUTPUT	Parameterized	Result of the instruction Perfromed by the ALU
OFLOW	OUTPUT	1	Overflow flag
COUT	OUTPUT	1	Carry out flag
G	OUTPUT	1	Greater than flag
E	OUTPUT	1	Equality flag
L	OUTPUT	1	Lesser than flag

CHAPTER 2 – TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 Testbench Architecture:

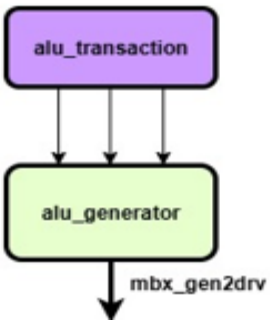




2.2 Components Details and Flowchart:

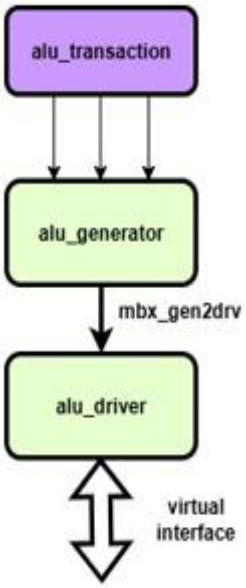
Transaction:

Transaction component is an object that encapsulates the stimulus exchanged between testbench components, containing all randomized inputs and non-randomized outputs of the DUT, excluding the clock signal, which is generated separately in the top module. The transaction and can have constraints to target specific test scenarios.



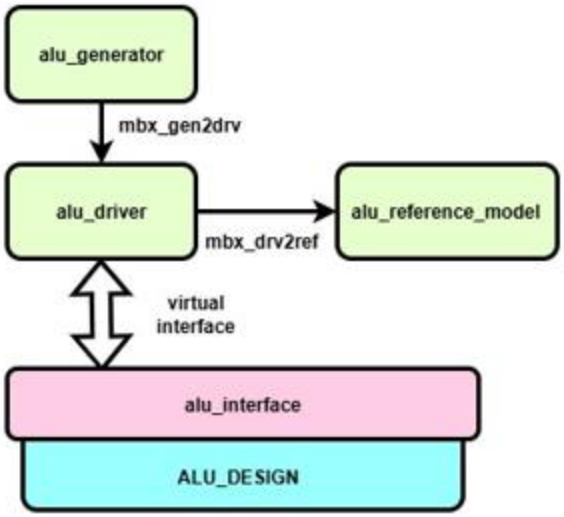
Generator:

Generator component of the testbench which generates constrained random stimuli (transactions) for the DUT. The generator then sends the generated stimuli to the driver through a mailbox(mbx_gen2drv).



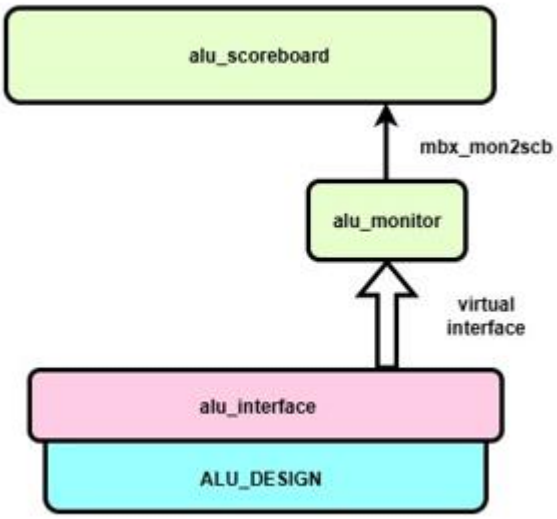
Driver:

Driver component converts high-level transactions into pin-level activity at the DUT inputs. It receives transactions from the generator via a mailbox(mbx_gen2drv) and drives them to the DUT using a virtual interface. Also, it forwards the received transactions to the reference model through another mailbox (mbx_drv2ref) for result prediction and comparison.



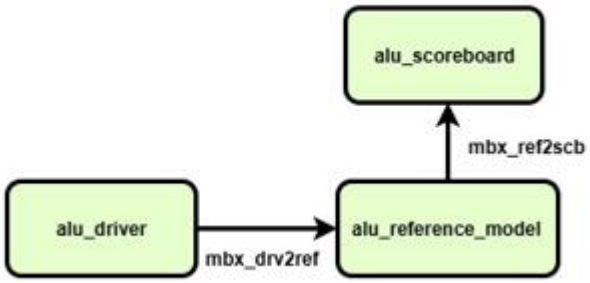
Monitor:

Monitor component converts pin-level activity from the DUT outputs into high-level transactions. It captures the DUT's output signals via a virtual interface and packages them into transactions, which are then sent to the scoreboard through a mailbox(mbx_mon2scb) for comparison and analysis.



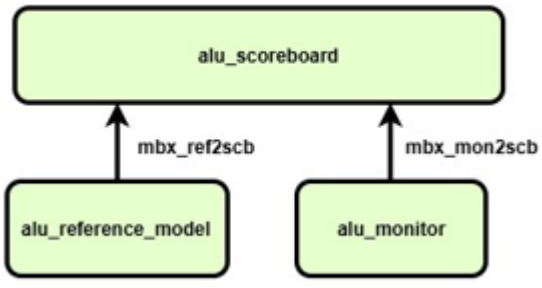
Reference Model:

Reference Model serves as a golden implementation/expected output for output prediction, validation and evaluation of the actual output. It is typically non-synthesizable. It is used to validate functionality and evaluate system performance. The model receives input transactions from the driver via a mailbox(mbx_drv2ref), processes them according to the intended functionality, and sends the predicted outputs to the scoreboard through another mailbox(mbx_ref2scr) for comparison.



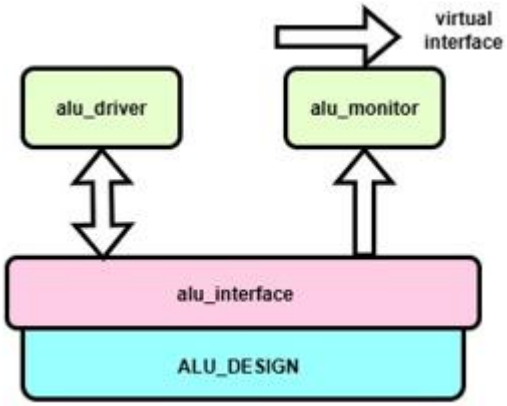
Scoreboard:

Scoreboard component receives the expected transactions from the reference model via one mailbox(mbx_ref2scb) and the actual transactions from the monitor via another(mbx_mon2scb). It compares these transactions to validate functional correctness and generates a report highlighting any mismatches or confirming successful operation.



Interface:

The Interface component encapsulates a bundled set of signals that connect the testbench to the DUT at the pin level. It directly maps to the DUT's input and output ports. This interface is accessed by testbench components such as the driver and monitor using virtual interface handles, enabling structured and reusable connectivity.



CHAPTER 3 – VERIFICATION RESULT AND ANALYSIS

3.1 Errors in DUT:

Incorrect Output for Two Operand Operation:

SUB_CIN : When both operands are equal and cin is high, the overflow signal is not being asserted as expected.

INC_A: This operations are not functioning correctly

INC_B: This operations are not functioning correctly;

DEC_B: This operations are not functioning correctly

MUL_SHIFT: This operations are not functioning correctly

OR: This operations are not functioning correctly

SHR1_A: This operations are not functioning correctly

SHR1_B: These operations are not functioning correctly

ROR_A_B: Although the rotate-right operation seems to be functioning, the error signal is not being asserted in failure scenarios.

ROL_A_B:Not performing as expected

INP_VALID_00: When a two-operand operation is issued, and INP_VALID is '00', the error is not being asserted.

Cycle_waiting: During the 16-cycle wait state, if INP_VALID transitions from '01' to '10' in the next cycle, the design incorrectly takes both inputs as valid and performs the operation, which leads to erroneous output**.**

3.2 Coverage Report:

Input Coverage:

Covergroup type:

cg_drv

Summary	Total Bins	Hits	Hit %
Coverpoints	554	529	**95.48%**
Crosses	64	36	**56.25%**

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CARRY_IN	2	1	1	50.00%	**50.00%**	**50.00%**
CLOCK_ENABLE	2	2	0	100.00%	**100.00%**	**100.00%**
COMMAND	32	18	14	56.25%	**56.25%**	**56.25%**
INPUT_VALID	4	3	1	75.00%	**75.00%**	**75.00%**
MODE	2	2	0	100.00%	**100.00%**	**100.00%**
OPERAND_A	256	253	3	98.82%	**98.82%**	**98.82%**
OPERAND_B	256	250	6	97.65%	**97.65%**	**97.65%**

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
MODE_CMD	64	36	28	56.25%	**56.25%**	**56.25%**

Output Coverage:

Covergroup type:

cg_monitor

Summary	Total Bins	Hits	Hit %
Coverpoints	264	245	**92.80%**
Crosses	0	0	**0.00%**

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CARR_OUT	2	2	0	100.00%	**100.00%**	**100.00%**
EQUAL	1	0	1	0.00%	**0.00%**	**0.00%**
ERROR	1	0	1	0.00%	**0.00%**	**0.00%**
GREATER	1	1	0	100.00%	**100.00%**	**100.00%**
LESSER	1	1	0	100.00%	**100.00%**	**100.00%**
OVERFLOW	2	2	0	100.00%	**100.00%**	**100.00%**
RESULT_CHECK	256	239	17	93.35%	**93.35%**	**93.35%**

Assertion Coverage:

Assertions Coverage Summary:

Search:

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
/top/intf/assert__0	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__1	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__2	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__3	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__ppt_clock_enable	0	1209	4433	3224	0	0	2	Covered
/top/intf/assert__ppt_reset	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__ppt_timeout_arithmetic	0	0	4433	4433	0	0	0	ZERO
/top/intf/assert__ppt_timeout_logical	810	0	4433	3623	0	0	17	Failed
/work.alu_interface/assert__0	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__1	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__2	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__3	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__ppt_clock_enable	0	1209	4433	3224	0	0	2	Covered
/work.alu_interface/assert__ppt_reset	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__ppt_timeout_arithmetic	0	0	4433	4433	0	0	0	ZERO
/work.alu_interface/assert__ppt_timeout_logical	810	0	4433	3623	0	0	17	Failed

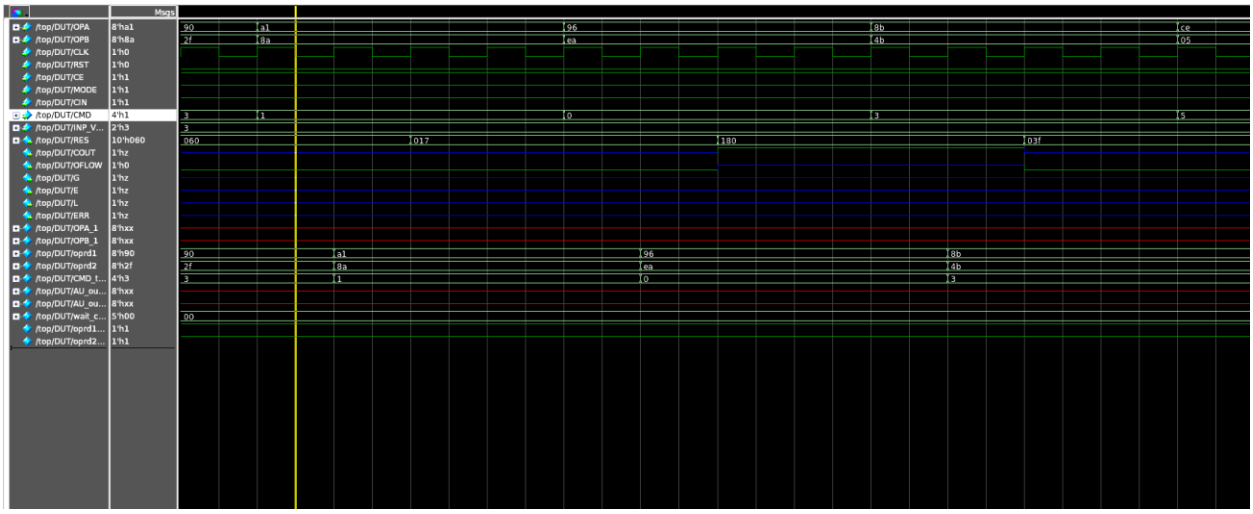
Overall Coverage:

Local Instance Coverage Details:

Total Coverage: 89.85% **81.25%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	117	106	11	1	90.59%	**90.59%**
Branches	73	67	6	1	91.78%	**91.78%**
FEC Conditions	20	10	10	1	50.00%	**50.00%**
Toggles	204	189	15	1	92.64%	**92.64%**

Output Waveform for Two cycle operation:



Output waveform for Three cycle operation