DESIGN DOCUMENT ALU PROJECT

Brief Introduction :

This project focuses on the design and implementation of a parameterized Arithmetic Logic Unit (ALU) using Verilog HDL, a key combinational logic block used in central processing units (CPUs) and digital signal processors (DSPs). The ALU is designed to perform a wide range of arithmetic operations (addition, subtraction, increment, decrement, multiplication), logical operations (AND, OR, XOR, NOT, etc.), and bit manipulation tasks such as shifts and rotations.

The ALU supports signed and unsigned computations, with internal flag generation for carry-out, overflow, and comparison results (greater-than, less-than, equal). Additionally, it includes error detection mechanisms for invalid control sequences, such as excessive rotation values.

To ensure correctness and robustness, the ALU is tested with a comprehensive Verilog testbench, driven by a set of pre-defined test vectors. The verification process checks the output results, status flags, and error conditions against expected values. This design is fully synthesizable, modular, and suitable for integration into larger digital systems like microprocessors, ALU cores, or custom hardware accelerators.

Moreover, the ALU is parameterized, allowing designers to easily modify the operand and command widths to suit various performance or area requirements, making it scalable and reusable in multiple applications.

Objectives :

- To design and implement a parameterized ALU in Verilog HDL that performs a wide range of operations efficiently and accurately.
- To support arithmetic operations including:
    - Unsigned and signed addition, subtraction, and operations with carry-in.
    - Increment and decrement for single operands.
    - Multiplication with pre-processing of operands for flexibility.

  - Comparison with result flags for greater-than, less-than, and equal.

- To implement logical operations such as AND, OR, XOR, XNOR, NAND, NOR, and bitwise NOT on individual operands.
- To incorporate bit manipulation functions:
  - Shifts: Logical left and right shifts.
  - Rotations: Left and right rotate operations with dynamic rotation values and error detection.
- To allow configurability through parameterization:
  - Operand width (WIDTH)
  - Command width (CMD_WIDTH)
  - Multiplication support through conditional compilation (MUL macro)
- To design a robust control unit capable of opcode decoding based on operation mode (arithmetic/logical) and command code (CMD).
- To develop a verification environment using a Verilog testbench that:
  - Drives stimulus from an external test vector file.
  - Monitors and logs results using a scoreboard.
  - Compares actual results with expected values and generates a pass/fail report.
- To ensure the ALU design is:
  - Synthesizable: Compliant with synthesis tools for FPGA/ASIC implementation.
  - Scalable: Easily extendable to wider data widths or additional instructions in the future.

Architecture :

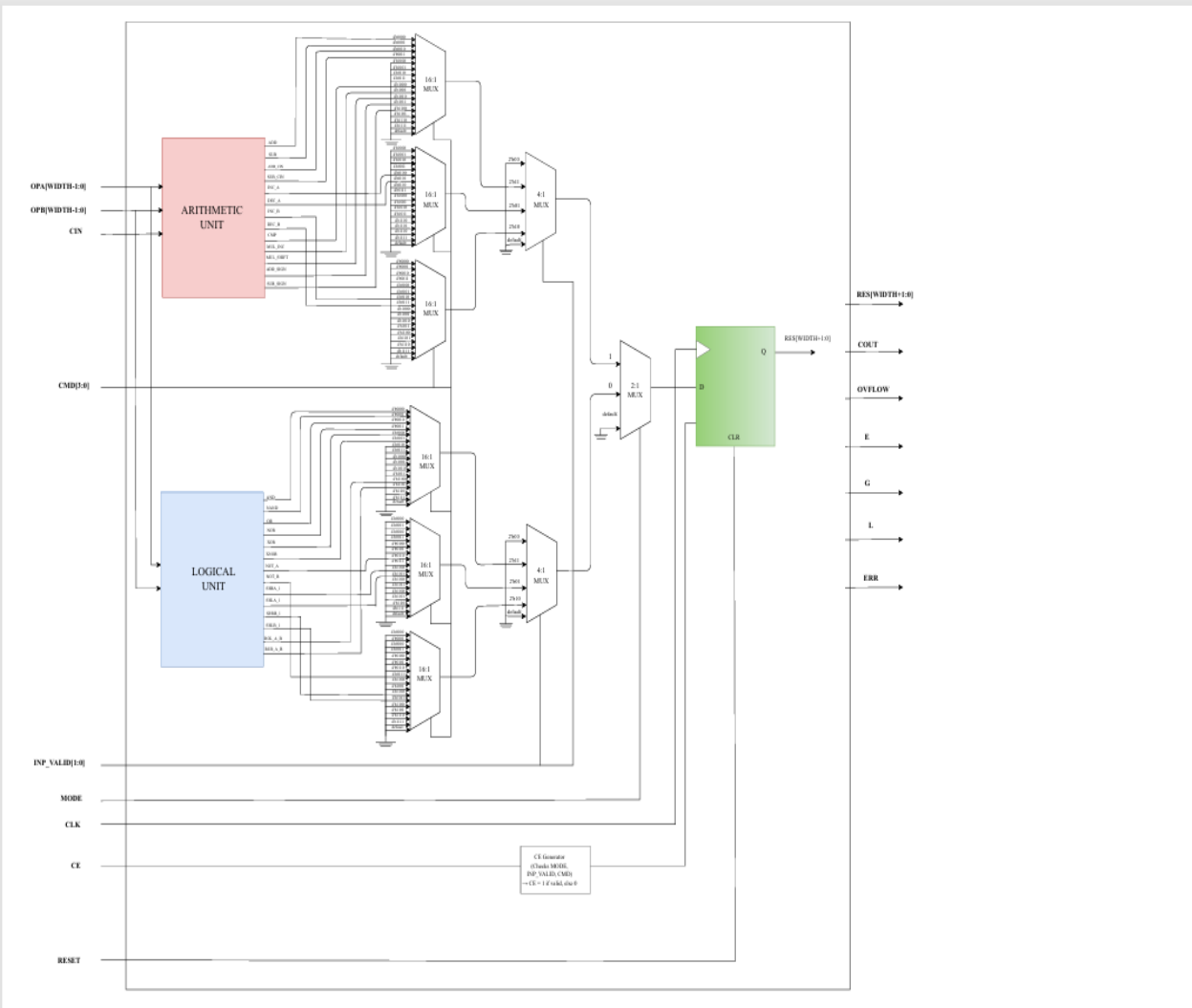

Figure 1: ALU Design Architecture

The Arithmetic Logic Unit (ALU) is designed using a modular and parameterized architecture, ensuring flexibility, scalability, and ease of integration into complex digital systems. This modular design promotes reusability and simplifies verification and debugging processes.

Inputs:

- **OPA, OPB:** Operand inputs for all operations.

- **CIN:** Carry-in used for arithmetic operations like ADD with carry.
- **CMD:** Operation selector input to choose the required function.
- **MODE:** Selects the operation type:
    - MODE = 1: Arithmetic operations.
    - MODE = 0: Logical operations.
- **INP_VALID (2-bit):** Indicates which operand(s) are valid:
    - Supports unary (e.g., NOT_A) and binary operations.
- **CE:** Clock Enable for synchronous control.
- **CLK, RST:** Standard clock and reset signals for synchronous logic.

Internal Registers: These registers are used to produce one clock cycle delay for the operation except multiplication operation, For the multiplication operation Three clock cycle delay are produced by using the temporary register

Outputs:

- **RES:** Final result of the ALU operation.
    - Width = WIDTH + 1 for arithmetic.
    - Width = 2 * WIDTH for multiplication.
- **COUT:** Carry-out from arithmetic operations.
- **OFLOW:** Indicates arithmetic overflow.
- **ERR:** Flags errors (e.g., invalid operations or shift values).
- **G, L, E:** Flags showing comparison results:
    - G = Greater, L = Less, E = Equal

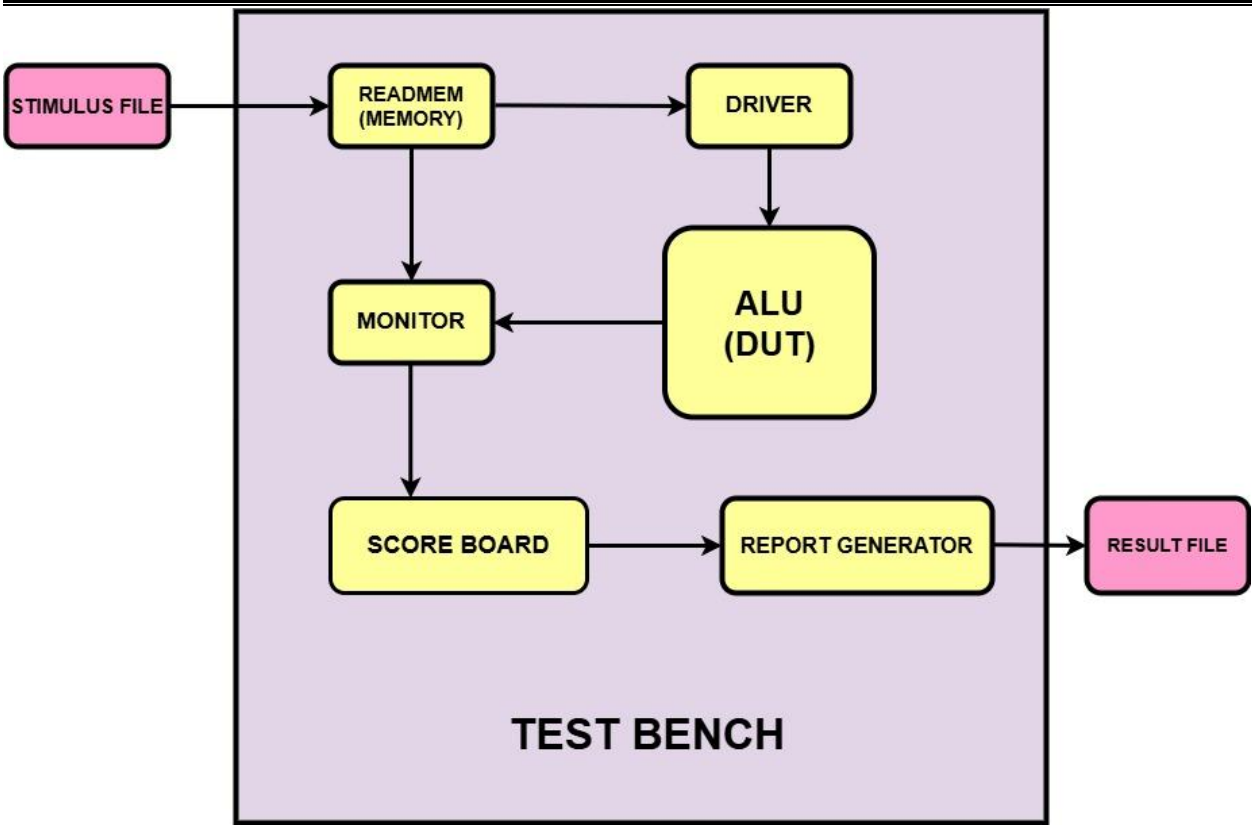

Figure 2: ALU Testbench architecture

The testbench architecture shown in Figure 2 is designed to verify the ALU's functionality in a modular and structured manner. It starts with a file named Stimulus.txt, which contains a list of predefined test vectors. These vectors include operands (OPA, OPB), control signals (CIN, CMD, MODE, INP_VALID, CE), and represent various test cases covering both normal and edge scenarios. The contents of this file are loaded into a memory array or queue using the $readmemb construct, allowing for easy modification of test cases without altering the testbench code.

Once the vectors are loaded, they are fed into the Driver module. The Driver is responsible for applying these inputs to the ALU in synchronization with the system clock (CLK) and clock enable (CE). It ensures that each test case is driven into the ALU at the correct time, maintaining proper sequencing and timing for accurate testing.

As the ALU processes each input, the Monitor module observes the outputs. These outputs include the result (RES), carry-out (COUT), overflow (OFLOW), error flags (ERR), and comparison flags (G, L, E). The Monitor logs both the applied inputs and the corresponding outputs for each test

case. This information is then passed to the Scoreboard, which acts as a checker by comparing the actual outputs of the ALU with the expected (golden) values, either derived from the test vectors or generated internally.

Finally, the Report Generator formats the verification results. It determines whether each test case has passed or failed and compiles a summary report. This report is written to an output file, typically named results.txt, which provides a clear and organized view of the ALU's performance across all test cases, helping developers quickly identify and debug any issues.

Working :

The ALU operates by interpreting the control inputs CMD and MODE to identify the specific arithmetic or logical task to perform on the given operands. Depending on the nature of the operation, the ALU may complete the process in one clock cycle or, for more complex operations like multiplication, use internal registers to handle it across multiple cycles. The input validity signals (INP_VALID) ensure that only legitimate and meaningful data is used to trigger computations. Meanwhile, status flags are continuously updated to reflect key result conditions, such as carry generation, arithmetic overflow, or the outcome of comparisons, providing immediate feedback for decision-making or further processing.

The ALU consists of the following functional components:

Pin Name	Pirority	Width	Direction	Description
CLK	1	-	INPUT	Asynchronous clock
RST	2	1	INPUT	Asynchronous reset
CE	3	1	CONTROL INPUT	Clock enable ,It enables the operation of ALU
INP_VALID	4	2	CONTROL INPUT	It decides which operand is valid 00 = OPA and OPB are invalid 01 = Only OPA is valid 10 = Only OPB is valid 11 = OPA and OPB are valid

MODE	-	1	CONTROL INPUT	Determine the behaviour of ALU MODE = 1 Arithmetic operation MODE = 0 Logical operation
CMD	-	Parameterized	CONTROL INPUT	Specifies which operation to Perform
OPA	-	Parameterized	DATA PIN	Parameterized Operand
OPB	-	Parameterized	DATA PIN	Parameterized Operand
CIN	-	1	DATA PIN	Carry in used for Addition with carry and Subtraction with carry
RES	-	Parameterized	OUTPUT	Parameterized output port where results are stored
COUT	-	1	OUTPUT	Single bit carry out used in Arithmetic operation
EGL	-	3	OUTPUT	Used for comparison of Two Operands
OFLOW	-	1	OUTPUT	Single bit over flow used for subtraction
ERR	-	1	OUTPUT	Error bit raised high when unpredictable behaviour of ALU

The internal control logic uses always blocks triggered by clock and reset, and performs operation decoding and execution in a **combinational** style for fast response.

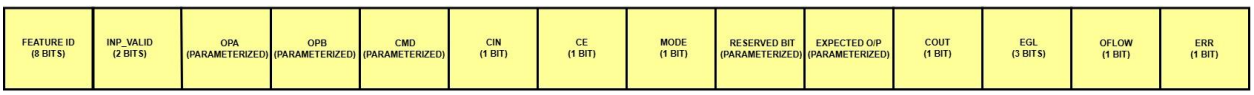

Figure 3: Testcase Packet

Command	MODE	Operation	Description	Flag Affected
0	1	ADD	Unsigned Addition	COUT
1	1	SUB	Unsigned Subtraction	OFLOW
2	1	ADD_CIN	Addition with carry	COUT
3	1	SUB_CIN	Subtraction with carry	OFLOW
4	1	INC_A	Increment OPA by 1	COUT
5	1	DEC_A	Decrement OPA by 1	OFLOW
6	1	INC_B	Increment OPB by 1	COUT
7	1	DEC_B	Decrement OPB by 1	OFLOW
8	1	CMP	Comoparison of OPA and OPB	E= Equal L = Less G = Greater
9	1	MUL1	Increment both operands by 1 and Multiply	OFLOW
10	1	MUL2	Shift OPA lefty by 1 and Multiply	OFLOW
11	1	ADD_SIGN	Signed Addition	OFLOW
12	1	SUB_SIGN	Signed_Subtraction	OFLOW

0	0	AND	Bitwise AND	-
1	0	NAND	Bitwise NAND	-
2	0	OR	Bitwise OR	-
3	0	NOR	Bitwise NOR	-
4	0	XOR	Bitwise XOR	-
5	0	XNOR	Bitwise XNOR	-
6	0	NOT_A	Compliment of OPA	-
7	0	NOT_B	Compliment of OPB	-
8	0	SHR1_A	Shift Right OPA by 1	-
9	0	SHL1_A	Shift Left OPA by 1	-
10	0	SHR1_B	Shift right OPB by 1	-
11	0	SHL1_B	Shift Left OPB by 1	-
12	0	ROL_A_B	Rotate Left OPA by OPB	-
13	0	ROR_A_B	Rotate Right OPA by OPB	-

The parameterized ALU module is designed to perform a wide range of arithmetic and logical operations on WIDTH-bit operands. It accepts two inputs (OPA, OPB), a carry-in (CIN), a CMD_WIDTH-bit command (CMD), mode control (MODE), input validity signals (INP_VALID), and standard clock/reset inputs (CLK, RST, CE). On the rising edge of the clock with CE asserted, inputs are latched into internal registers. If RST is active, all internal states are cleared.

In arithmetic mode (MODE = 1), the ALU executes operations like addition, subtraction (with/without carry/borrow), and signed arithmetic with overflow and status flag detection. It supports pipelined multiplication (OPA+1)×(OPB+1) and (OPA<<1)×OPB over three cycles, and includes comparison logic to set greater (G), less (L), or equal (E) flags. When only one operand is valid, it performs increment or decrement operations.

In logical mode (MODE = 0), the ALU handles standard logic operations (AND, OR, XOR, etc.), bitwise rotations (ROL, ROR) based on OPB, and unary functions like NOT or single-bit shifts. If the shift/rotate count exceeds operand width, the ERR flag is raised.

All operations update relevant flags such as COUT, OFLOW, and ERR. The result (RES) is extended up to 2WIDTH+1 bits using conditional compilation (ifdef) to accommodate wide operations. This structured, flag-driven design ensures robustness, synthesizability, and easy integration into larger digital systems.

Results:

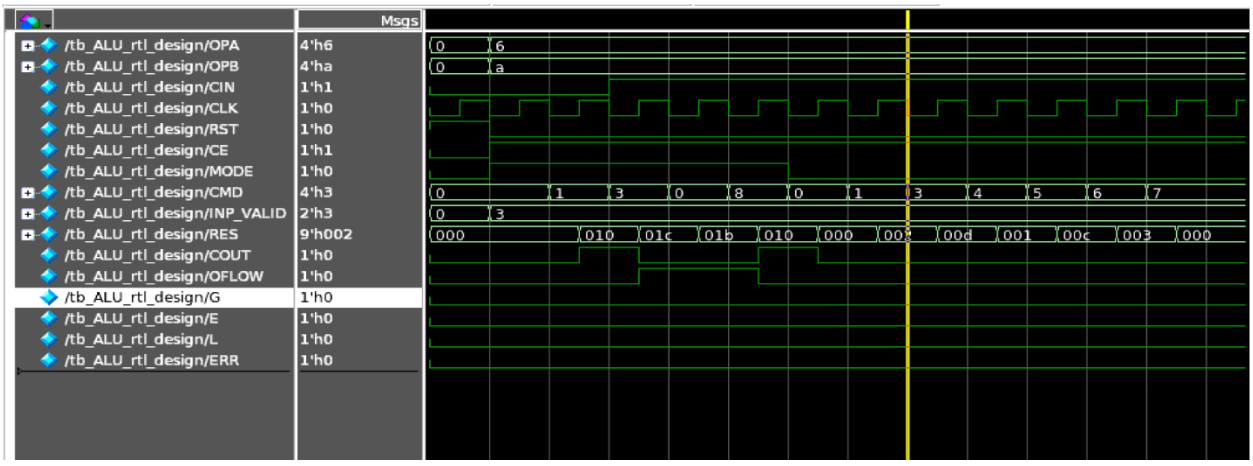

Figure 4: Waveform from the questa SIM

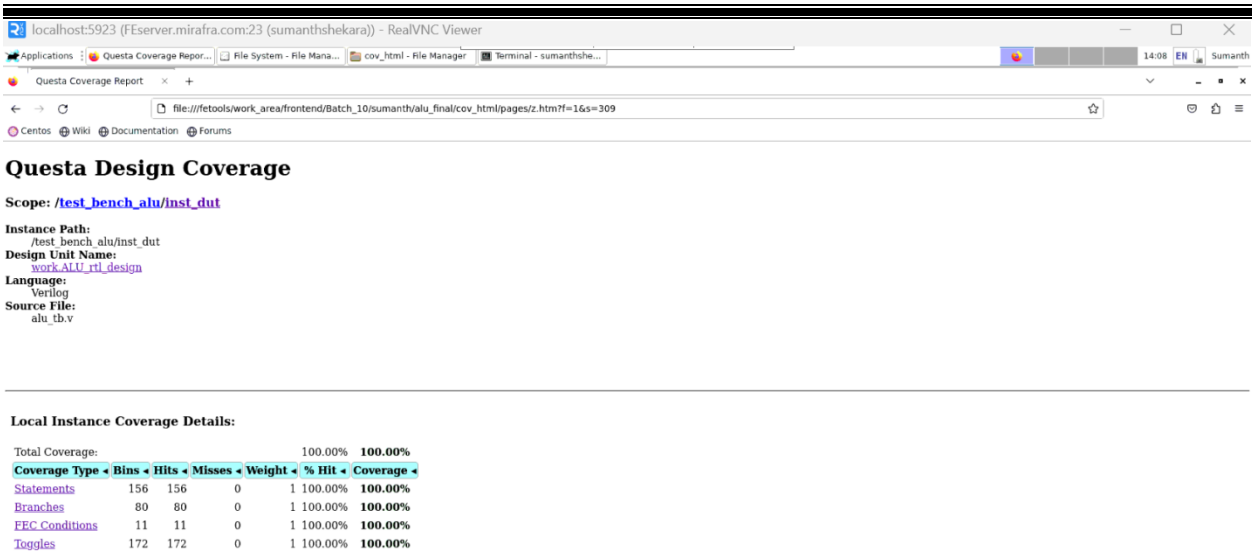

Figure 5: Coverage report

Conclusion :

The ALU was successfully implemented in Verilog and verified through comprehensive simulation. It supports a wide range of arithmetic, logical, shift, rotate, and comparison operations, with detailed status flag generation and error detection for enhanced reliability. The modular, parameterized design ensures reusability and scalability for integration into complex systems. A stimulus-driven testbench, golden reference models, and scoreboard-based comparisons were employed to validate functionality, ensuring accuracy and robustness. This structured verification confirms that all functional requirements were met, delivering a reliable and efficient ALU suitable for processors and digital systems requiring high computational integrity and flexibility.

Future Improvement:

To improve performance and reduce latency-related errors in multiplication and other arithmetic operations, the ALU will be optimized to execute all operations within a fixed three-clock-cycle pipeline. This pipelined approach ensures consistent timing across operations, enhances

throughput, and provides more predictable system behavior, which is critical in timing-sensitive applications.

In terms of functionality expansion, future iterations of the ALU will support floating-point arithmetic for high-precision real number computations and complex number arithmetic to address the needs of applications such as signal processing and scientific simulations. These enhancements will broaden the ALU's capability to handle advanced computational tasks.

To introduce greater flexibility in instruction execution and input handling, the ALU will utilize a refined inp_valid signal mechanism. When inp_valid = 01, operand A is captured and stored; when inp_valid = 10, operand B is stored similarly. Once both operands are valid (inp_valid = 11), the ALU performs the designated operation in the subsequent clock cycle. This staged input method enables more controlled operand loading and execution, making it ideal for systems that require synchronized or conditional processing.

Finally, the ALU's architecture will be designed with scalability in mind, allowing seamless integration of new operation codes and functional extensions. This forward-compatible structure ensures that the ALU remains adaptable to future computational demands without requiring major redesigns.