ALU PROJECT

Introduction

This project presents the design and implementation of an Arithmetic Logic Unit (ALU) using Verilog HDL. The ALU is a fundamental combinational circuit that performs a wide range of arithmetic and logical operations on binary inputs. Designed as a parameterized and modular block, the ALU supports operations such as addition, subtraction, bitwise logic (AND, OR, XOR, etc.), shifts, comparisons, and specialized functions like signed arithmetic and multiplication with latency control.

While most operations are purely combinational, the design includes clocked logic for pipelining and handling multi-cycle operations (like multiplication), aligning the output with the timing needs of synchronous digital systems. The ALU accepts command inputs to select operations, along with mode and control signals, and generates outputs including result, carry-out, overflow, and condition flags (greater, equal, less).

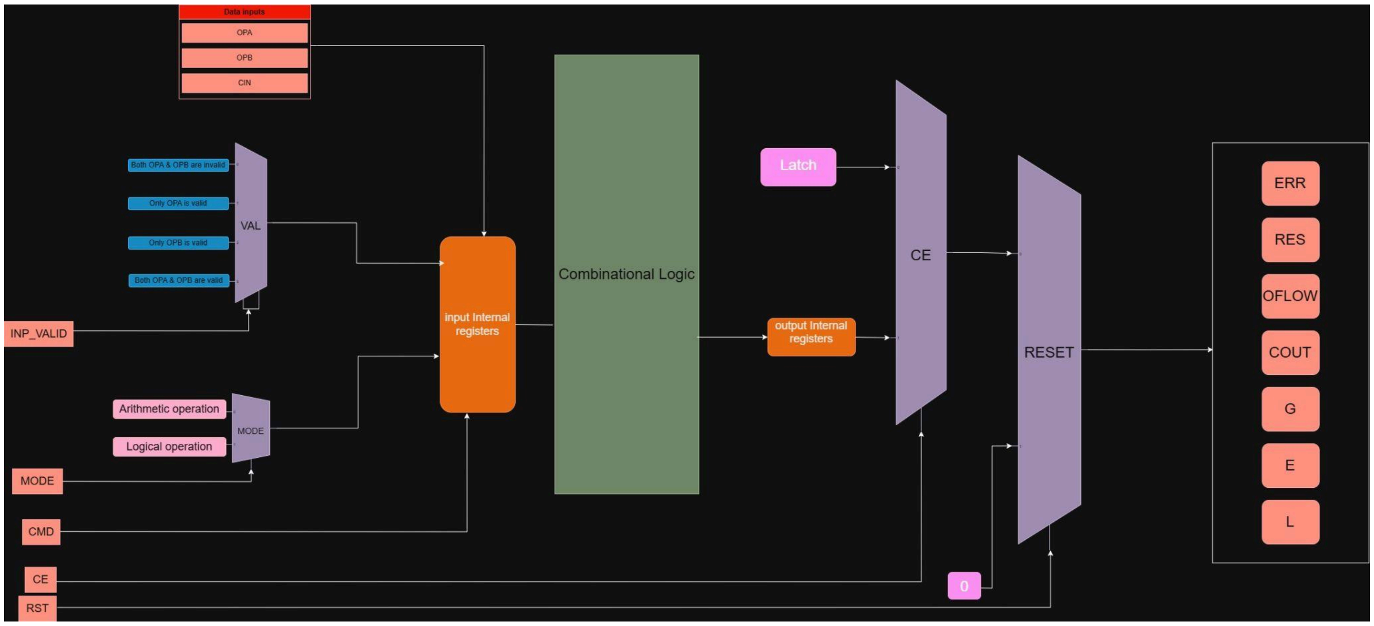
This Verilog-based ALU forms a critical component in processor and embedded system design, offering a flexible and efficient foundation for performing core computational tasks.

Objectives

- ➢ To design and implement an Arithmetic Logic Unit (ALU) in Verilog HDL capable of performing a variety of arithmetic and logical operations.
- ➢ To support combinational as well as multi-cycle operations (e.g., multiplication with delays) using appropriate control logic.
- ➢ To incorporate parameterization for data width and command structure, enabling reusability and scalability.

➢ To ensure the ALU handles signed and unsigned operations, carry, overflow, and comparison flags effectively.
➢ To verify the functionality and correctness of the ALU through comprehensive simulation testbenches in a Verilog simulation environment.
➢ To demonstrate the ALU's behavior under different operational modes and input conditions, including edge cases and invalid inputs.

Architecture



ALU architecture

The architecture of the Verilog ALU consists of clearly defined input, processing, and output stages structured in an RTL-compliant design. At the input stage, two operand registers temporarily store incoming values OPA and OPB, along with control signals like CMD, MODE, and INP_VALID. A dedicated control decoder interprets the opcode (CMD), operation mode (MODE), and input validity (INP_VALID) to generate control signals for enabling specific functional units. The datapath includes separate arithmetic and logical units for executing operations such as addition, subtraction, shifting, bitwise AND, OR, XOR, and more. A multiplexer (OpSel) dynamically selects the output from these units or from a dedicated multiplication block, which introduces intentional delays using internal registers to simulate 1-cycle and 2-cycle latency for different multiplication modes. The selected result then passes into an output register block that captures the final result (RES) along with condition flags such as carry (COUT), overflow (OFLOW), and comparison indicators (G, E, L). This modular and pipelined structure ensures a clean separation of logic, better testability, and compatibility with synthesis flows, while supporting both combinational and clocked operations.

Working

The Arithmetic Logic Unit (ALU) processes operands and operation codes to perform a wide range of arithmetic and logical operations, determined by control signals. Internally, the design uses pipelining techniques to handle certain operations like multiplication that require more than one cycle to complete.

Operation Mode Selection – MODE Signal:
MODE = 1 (Arithmetic Mode): Enables arithmetic operations such as ADD, SUB, INC, DEC, MUL, and signed operations like ADD_SIGN and SUB_SIGN.
MODE = 0 (Logical Mode): Enables logical operations including AND, OR, XOR, NOT, NAND, NOR, XNOR, SHL, SHR, and rotational shifts.

Operand Validity – INP_VALID Signal:
Operand validity is controlled via a 2-bit signal INP_VALID:
2'b00: Both OPA and OPB are invalid. Operation is skipped and the error flag ERR is asserted.
2'b01: Only OPA is valid; single-operand operations are performed on OPA.
2'b10: Only OPB is valid; single-operand operations are performed on OPB.
2'b11: Both operands are valid, enabling full binary operations.

Combinational Operation and Result Handling:
The ALU operations are combinationally executed, and the result is stored in an intermediate result register.
The result is then forwarded to the output register RES on the next rising edge of the clock, introducing a 1-clock-cycle latency for all standard operations.

Multiplication with Internal Pipeline Registers:
For multiplication operations (e.g., INC_MUL, SHL1_A_MUL_B), the result is not immediately available.
Instead, the multiplication result is first captured in a temporary register mul_res.

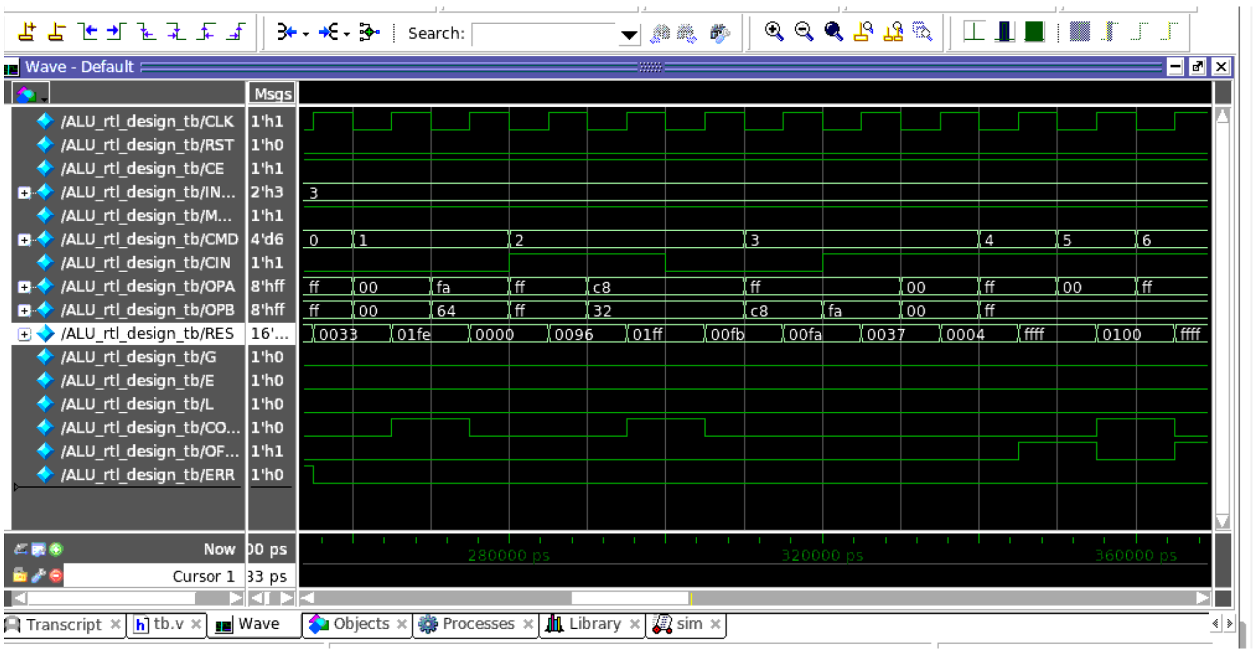
To simulate realistic latency for complex operations:
A 1-cycle delay is introduced using one internal register for partial computation.
A 2-cycle delay is achieved by staging the value across two internal registers, emulating a multi-stage pipeline.
The final multiplication result is moved into the output register RES only after the delay stages have completed, maintaining synchronous behavior without combinational glitches.

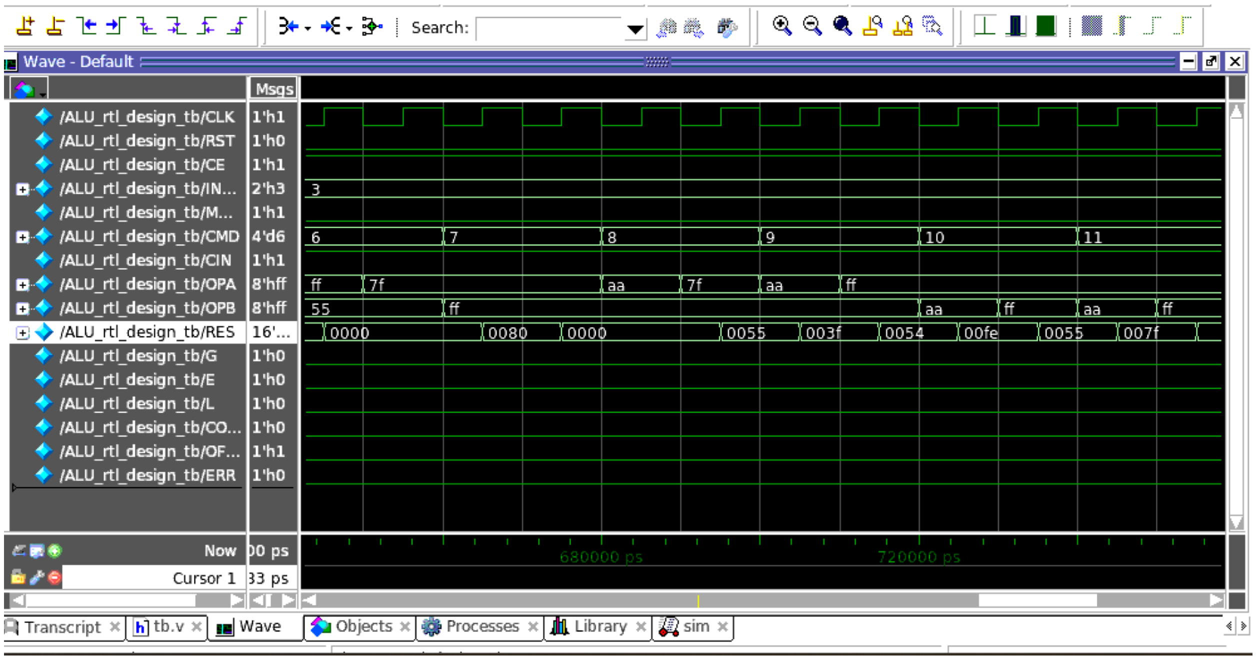
Status Flag Updates:
COUT: Indicates carry-out in unsigned arithmetic operations.
OFLOW: Signals overflow in signed operations.
G, E, L: Comparison result flags (Greater, Equal, Less).
ERR: Asserted when operand validity is insufficient or an unsupported opcode is detected.

Result

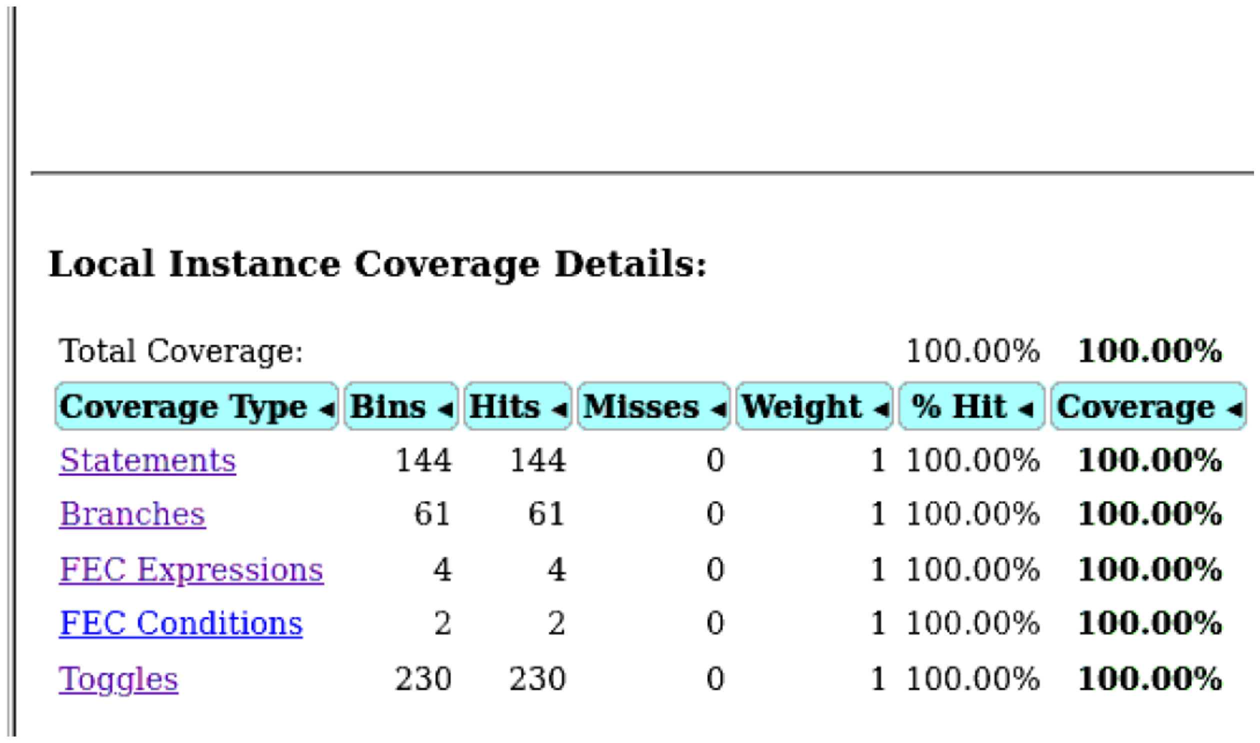
The simulation of the Verilog-based ALU confirms that all arithmetic and logical operations are executed correctly according to the provided opcode. The design accurately processes inputs based on the MODE and INP_VALID signals, and produces valid outputs including the result (RES) and corresponding status flags (COUT, OFLOW, G, E, L, and ERR).

Notably, multiplication operations correctly reflect one and two clock cycle delays, as implemented using internal pipeline registers. The waveform analysis validates that the ALU responds predictably across all supported instructions, including edge cases like overflows, comparisons, and invalid input conditions.


Search:
Wave - Default
Msgs
/ALU_rtl_design_tb/CLK 1'h1
/ALU_rtl_design_tb/RST 1'h0
/ALU_rtl_design_tb/CE 1'h1
/ALU_rtl_design_tb/IN... 2'h3
/ALU_rtl_design_tb/M... 1'h1
/ALU_rtl_design_tb/CMD 4'd6
/ALU_rtl_design_tb/CIN 1'h1
/ALU_rtl_design_tb/OPA 8'hff
/ALU_rtl_design_tb/OPB 8'hff
/ALU_rtl_design_tb/RES 16'...
/ALU_rtl_design_tb/G 1'h0
/ALU_rtl_design_tb/E 1'h0
/ALU_rtl_design_tb/L 1'h0
/ALU_rtl_design_tb/CO... 1'h0
/ALU_rtl_design_tb/OF... 1'h1
/ALU_rtl_design_tb/ERR 1'h0
3
0 1 2 3 4 5 6
ff 00 fa ff c8 ff 00 ff 00 ff
ff 00 64 ff 32 c8 fa 00 ff
0033 01fe 0000 0096 01ff 00fb 00fa 0037 0004 ffff 0100 ffff
Now 00 ps
Cursor 1 33 ps
280000 ps 320000 ps 360000 ps
Transcript tb.v Wave Objects Processes Library sim


Search:
Wave - Default
Msgs
/ALU_rtl_design_tb/CLK 1'h1
/ALU_rtl_design_tb/RST 1'h0
/ALU_rtl_design_tb/CE 1'h1
/ALU_rtl_design_tb/IN... 2'h3
/ALU_rtl_design_tb/M... 1'h1
/ALU_rtl_design_tb/CMD 4'd6
/ALU_rtl_design_tb/CIN 1'h1
/ALU_rtl_design_tb/OPA 8'hff
/ALU_rtl_design_tb/OPB 8'hff
/ALU_rtl_design_tb/RES 16'...
/ALU_rtl_design_tb/G 1'h0
/ALU_rtl_design_tb/E 1'h0
/ALU_rtl_design_tb/L 1'h0
/ALU_rtl_design_tb/CO... 1'h0
/ALU_rtl_design_tb/OF... 1'h1
/ALU_rtl_design_tb/ERR 1'h0
3
6 7 8 9 10 11
ff 7f aa 7f aa ff
55 ff aa ff aa ff
0000 0080 0000 0055 003f 0054 00fe 0055 007f
Now 00 ps
Cursor 1 33 ps
680000 ps 720000 ps
Transcript tb.v Wave Objects Processes Library sim

ALU waveform



Local Instance Coverage Details:

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Total Coverage:					100.00%	100.00%
Statements	144	144	0	1	100.00%	100.00%
Branches	61	61	0	1	100.00%	100.00%
FEC Expressions	4	4	0	1	100.00%	100.00%
FEC Conditions	2	2	0	1	100.00%	100.00%
Toggles	230	230	0	1	100.00%	100.00%

Code coverage

This confirms the ALU's functional correctness and the successful integration of pipeline techniques for delay-sensitive operations.

Conclusion

The Verilog-based ALU successfully meets all specified functional requirements, demonstrating reliable performance across a wide range of arithmetic and logical operations. The modular design supports clear separation of functionality, enabling easier debugging, extension, and reuse. The use of internal registers for pipelining ensures proper timing for multi-cycle operations such as multiplication, while maintaining single-cycle execution for other combinational instructions.

Simulation results confirm accurate output behavior and correct status flag updates under various operating conditions, validating the design's correctness. Additionally, the ALU code is fully synthesizable and structured for efficient hardware implementation, making it suitable for integration into larger digital systems.

Improvement

To enhance the current ALU design, the following improvements are planned:

- Extended Operation Set: Add support for more complex arithmetic operations such as division, modulus, and barrel shifting.

- Wider Data Width: Generalize the ALU for scalable data widths (e.g., 16-bit, 32-bit) to support higher-precision computations.

- Formal Verification: Incorporate formal methods to mathematically prove correctness across all possible inputs, ensuring robustness beyond simulation.

- Optimized Multiplication: Replace the current register-based delay approach with a dedicated pipelined multiplier unit for efficient multi-cycle operations.

- Pipelined Stages: Introduce multi-stage pipelining to improve throughput and enable concurrent instruction execution.