**ALU Project**

**-------------------------------------------------------------**

**Jason Linus Rodrigues**

**Emp id: 6078**

**-------------------------------------------------------------**

**Brief Introduction:**

The Arithmetic and Logic Unit (ALU) is a fundamental combinational circuit used in digital systems to perform arithmetic and logical operations. In Verilog, the ALU is designed using sequential and combinational logic constructs, ensuring that its output depends only on current inputs while requiring a clock signal. It can perform a variety of operations such as addition, subtraction, AND, OR, XOR, and comparisons, all based on a clock enable, mode, command. The operation selected is determined by the mode and command, which guides the ALU to execute the desired function. Being modular, the Verilog-based ALU can be easily adapted to different bit-widths and extended functionalities.

**Objectives:**

* To design and synthesize a N-bit Arithmetic Logic Unit which is a digital circuit that performs arithmetic and logical operations using Verilog.
* To Perform the specified command on the given inputs with a one-cycle delay, except for the multiplication operation, which generates the result in the third clock cycle.
* To test the functionality of the ALU design by applying a variety of input combinations and verifying the output results that matches with the expected values.

**Architecture:**

The ALU is parameterized to operate on 8-bit wide input data, allowing it to process two 8-bit operands and generate a 16-bit result. The ALU interface includes the following inputs:

* clk (Clock signal)
* rst (Reset signal)
* operand\_A and operand\_B (8-bit input operands)
* cmd (Command or opcode for selecting the operation)
* inp\_valid (Indicates when input data is valid)
* CE (Clock Enable signal)
* mode (Defines the operational mode is either arithmetic or logical)
* Cin (Carry-in for arithmetic operations)

The outputs of the ALU include:

* res (16-bit result)
* err (Error flag)
* oflow (Overflow flag)
* cout (Carry-out flag)
* g (Greater-than flag)
* l (Less-than flag)
* e (Equal-to flag)

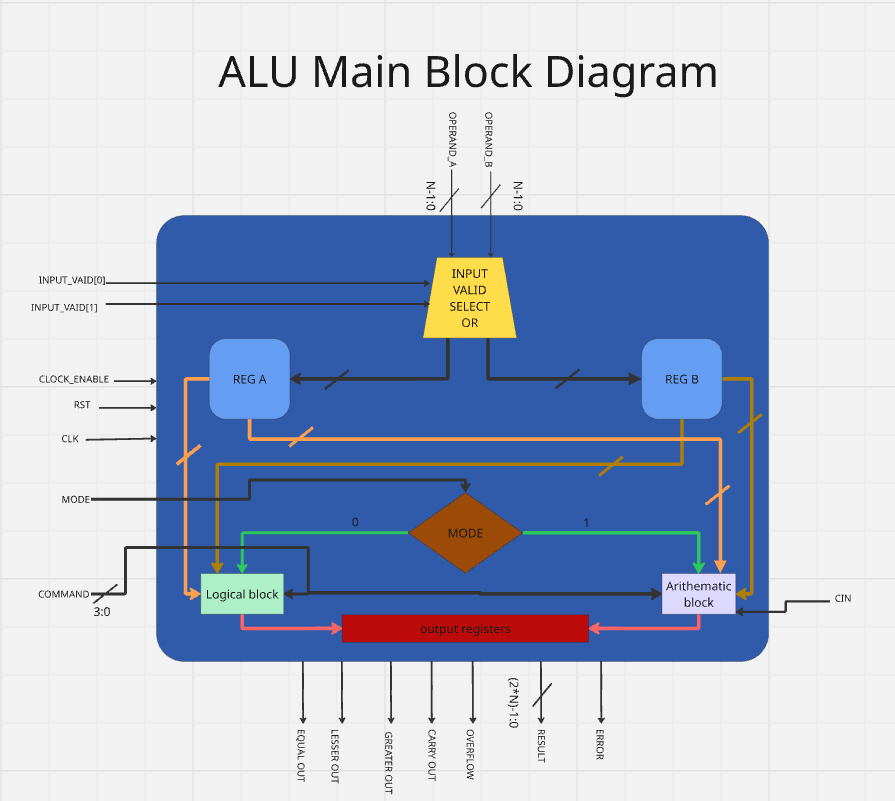


Figure 1: ALU architecture

An **input valid selector block** ensures that only valid data is captured into temporary registers, which provide stable inputs for the Arithmetic Logical Unit

The **arithmetic block** of the ALU is designed to perform a wide range of operations on the input operands. These operations include:

* **Basic Arithmetic operations:** Addition and subtraction
* **Carry-based Operations:** Addition with carry (Cin) and subtraction with carry
* **Increment/Decrement operations:**
  + Increment operand\_A by 1
  + Decrement operand\_A by 1
  + Increment operand\_B by 1
  + Decrement operand\_B by 1
* **Comparison operations:** Determine whether operand\_A is greater than, less than, or equal to operand\_B
* **Multiplication operations:**
  + Multiplication of (operand\_A + 1) and (operand\_B + 1)
  + Multiplication of (operand\_A + 1) with operand\_B
* **Signed Arithmetic operations:** Signed addition and signed subtraction

Similarly, the ALU supports several basic **logical operations**, each performing a specific function on binary inputs:

* **Basic Gate operations:**
* AND
* NAND
* OR
* NOR
* XOR
* XNOR
* NOT on operand\_A
* NOT on operand\_B
* **Shift operations:**
* Shift Left by 1 on operand\_A
* Shift Left by 1 on operand\_B
* Shift Right by 1 on operand\_A
* Shift Right by 1 on operand\_B
* **Rotate Operations:**
* Rotate Left
* Rotate Right

**Testbench Architecture**

**A diagram of a structure

AI-generated content may be incorrect.**

Figure 2: Testbench Architetcure

A screen shot of a computer

AI-generated content may be incorrect.

Figure 3: stimulus packet and response packet format

The ALU testbench architecture consist of as follows:

1) **stimulus.txt (57-bit Test Vector)**  
This external file holds all predefined test cases for the ALU. Each line in the file represents a 57-bit vector that encapsulates all required input signals for a single test scenario.

2) **Stimulus Generator (readmem\_inp)**  
This module reads the test vectors from stimulus.txt using the $readmemh system task. The vectors are loaded into an internal memory array, where each one is sequentially provided as curr\_test\_case[56:0] to the driver. This mechanism effectively translates file-based inputs into usable simulation signals.

3)**Driver**  
The driver extracts individual input fields from the current test vector and applies them to the ALU Design Under Test (DUT). It synchronizes the signal driving process with the simulation clock to ensure accurate stimulus delivery.

4)**ALU DUT (Design Under Test)**  
This is the core ALU implementation being tested. It processes the incoming input signals from the driver and generates output signals such as RES, COUT, EGL, OFLOW, and ERR, based on the ALU's logic.

5)**Response Module**  
Captures the ALU DUT’s outputs for each test case and stores them in a response packet. This packet is associated with the corresponding test vector for further analysis.

6) **Monitor**  
The monitor observes the ALU's outputs in real-time and stores them into the response packet. It formats the output data into a structure that facilitates comparison with expected results.

7)**Scoreboard**  
This module compares the actual DUT outputs captured by the monitor with the expected results derived from the test vector. It logs the result of each comparison (pass or fail) into the readmem\_scb[55:0] array and highlights any mismatches.

8)**Report Generator**  
At the end of the simulation, this module compiles a summary of all test results and writes them to a report file (results.txt). The report clearly indicates which test cases passed or failed, providing a comprehensive overview of ALU verification status.

**Working** -

A screenshot of a diagram

AI-generated content may be incorrect.

Figure 4: Input Selector Block

Whenever the Pos edge clock is triggered, it starts to undergo a input\_valid selection operation, which will able to select the temporary register operands. The following selection operation are as follows:

* **When selector is 00**, both the temporary register in A and B is set to zero.
* **When selector is 01**, the temporary register of A stores the input operand A, and the temporary register of B is set to zero
* **When selector is 10,** the temporary register of B stores the input operand B, and the temporary register of A is set to zero
* **When selector is 11**, the temporary register of A stores the input operand A and the temporary register of B stores the input operand B.

A diagram of a flowchart

AI-generated content may be incorrect.

Figure 5:Flowchart of ALU operation

Once the input operand values are store in the temporary register, simultaneously it will perform the operation depicted in the picture above. As soon as the posedge of the clock signal is triggered, it undergoes the following process:

1)  **Reset** : the reset only works when the rst signal is set to 1, thereby clearing the internal register and the outputs stored.

2) **Clock Enable:** the clock enable only works when the CE is set to 1, thereby performing the ALU operation. If the CE is disabled, then the outputs stored will latch onto previous output values.

3) **Mode:** The ALU supports two operational modes: **Arithmetic** and **Logical**. The mode is determined by a control signal:

* If **MODE = 1**, the ALU executes **arithmetic operations**.
* If **MODE = 0**, it performs **logical operations**.

The specific operation within each mode is selected using a **4-bit CMD signal**, as detailed below:

**Arithmetic Commands:**

|  |  |  |
| --- | --- | --- |
| CMD | OPERATION | DESCRIPTION |
| 0 | ADD | Unsigned addition of Operand\_A and Operand\_B. |
| 1 | SUB | Unsigned subtraction of Operand\_A and Operand\_B |
| 2 | ADD\_CIN | Unsigned addition with carry-in. |
| 3 | SUB\_CIN | Unsigned subtraction with borrow-in. |
| 4 | INC\_A | |  | | --- | |  |  |  | | --- | | Increment Operand\_A by 1. | |
| 5 | DEC\_A | Decrement Operand\_A by 1. |
| 6 | INC\_B | Increment Operand\_B by 1. |
| 7 | DEC\_B | Decrement Operand\_B by 1. |
| 8 | CMP | Compare Operand\_A and Operand\_B. |
| 9 | SPECIAL1 | Increment both operands by 1, then multiply the results. |
| 10 | SPECIAL 2 | |  | | --- | |  |  |  | | --- | | Shift Operand\_A left by 1, then multiply with Operand\_B. | |
| 11 | SPECIAL 3 | Perform signed addition and comparison. |
| 12 | SPECIAL 4 | Perform signed subtraction and comparison. |

4) **Outputs:**

Once the computation is dome in the arithmetic or logical operation, the results are stored and update and the next clock cycle, including the flags and errors.

**Result—**

A screenshot of a document

AI-generated content may be incorrect.

Figure 6: coverage report from Questa sim

The ALU successfully performed all operations correctly and was fully synthesizable. It samples the input on the first rising edge of the clock and outputs the result on the second rising edge for all operations except multiplication. The design was validated using 87 test cases, all of which passed. As shown in Figure 6, a code coverage of 96.42% was achieved.

**Conclusion**

The ALU is implemented and verified using a structured testbench to ensure it meets all functional requirements. It efficiently handles both arithmetic and logical operations while incorporating status flag management for enhanced control. A systematic verification methodology guarantees computational accuracy, reliability, and smooth integration into digital systems. The use of a stimulus-based testbench, and a scoreboard for result comparison enables thorough validation and effective error detection, thereby enhancing the overall design robustness and efficiency.

**Future Improvement** –

* The addition of pipelined stages to enable multiple instructions to be processed concurrently.
* Reducing ALU operations to execute within just two clock cycles.
* Expanding the ALU to support a greater number of arithmetic and logical functions.
* To better implement multiplication operation in order to achieve result at 3 clock cycle operation.
* To optimize the ALU design by minimizing the use of internal registers.
* Formal verification techniques to ensure correctness and reliability of operations.