ALU DESIGN REPORT

# Introduction:

An Arithmetic Logic Unit (ALU) is one of the fundamental building blocks of digital systems, responsible for performing arithmetic operations like addition, subtraction, multiplication, and logical operations such as AND, OR, and XOR. It plays a critical role in the datapath of CPUs, DSPs, and custom digital architectures.

This project focuses on designing an 8-bit ALU using Verilog Hardware Description Language (HDL). The ALU design is purely combinational in nature and is integrated with additional control and status features. The operations are selected based on an input opcode (cmd) and a mode signal to distinguish between arithmetic and logical operations.

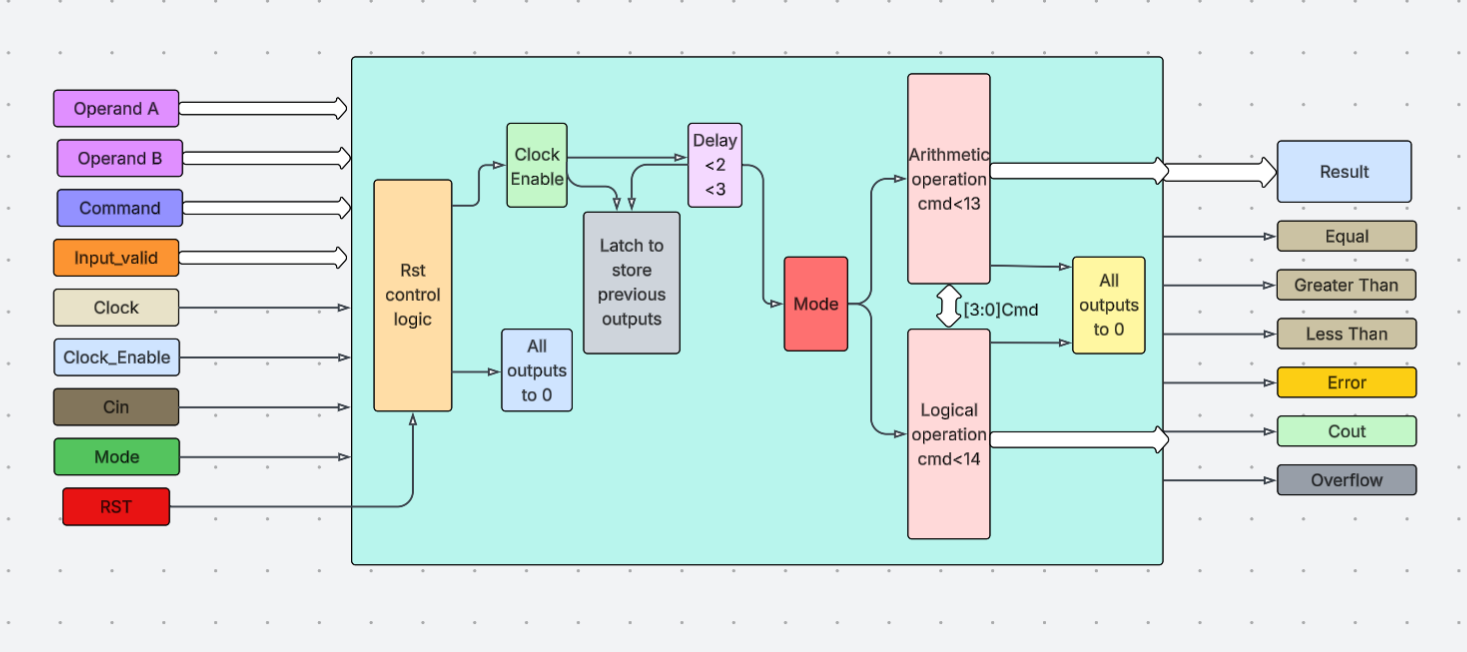
The goal of this project was to develop a synthesizable and simulation-verifiable ALU that can support a variety of basic and advanced operations. Functional correctness was tested through simulation using an extensive testbench in Verilog. Although no FPGA implementation was done, the ALU was designed in a modular and scalable way to support future extension.

# Objectives:

The main objectives of this ALU design project are as follows:

* To implement an Arithmetic Logic Unit (ALU) using Verilog with 8-bit operands and configurable width.
* To support a comprehensive set of arithmetic and logic operations through an opcode-driven control logic.
* To make the design modular and scalable using parameters (width and c\_w) for easy modification of operand and result size.
* To include status flags for result analysis: zero (zero), overflow (overflow), carry-out (cout), error (err), and comparators like greater (g), less (l), and equal (e).
* To use a mode signal to separate logic and arithmetic operations within the same architecture.
* To verify the ALU functionality using simulation with randomized and edge-case inputs.
* To handle operation delays where needed (like multiplication) using a delay1 signal to mimic pipeline latency or multi-cycle operation.
* To avoid FPGA synthesis and focus solely on simulation-based verification due to time and infrastructure constraints.
* To prepare the design for future FPGA implementation by maintaining synthesizability and clean coding practices.

# Architecture



The ALU architecture consists of the following core components:

**Inputs:**

* opa, opb: 8-bit input operands.
* cmd: 4-bit control signal that defines the specific operation to be performed.
* cin: 1-bit carry-in for operations like add-with-carry.
* input\_valid: 2-bit signal that validates whether operands are ready.
* mode: 1-bit signal that selects arithmetic (1) or logic (0) mode.
* clk, rst, ce: Clock, reset, and clock-enable signals.

**Internal Logic:**

* A case-based decoder that selects the operation depending on the mode and opcode.
* Control logic that checks input\_valid and controls delay using delay1.
* Separate arithmetic and logic blocks handled within the same always block.

**Outputs:**

* res: Result of the operation (16-bit to accommodate wide results like multiplication).
* cout, overflow: Arithmetic status flags.
* g, l, e: Comparator flags (greater than, less than, equal).
* err: Error flag indicating invalid operations or inputs.

**Delay Logic:**

* Some operations (like multiplication) introduce simulation-based latency using the delay1 register.
* The operation executes only when delay1 reaches a specified threshold.

# Working

The ALU design operates on the rising edge of the system clock. Here's a step-by-step explanation of how it works:

1. **Reset Phase:**  
   On reset (rst high), all outputs are cleared to 0. This ensures the ALU starts from a known state.
2. **Clock Enable (ce):**  
   When ce is asserted and delay logic allows (delay1 >= 1), the ALU checks mode to select either the arithmetic or logical block.
3. **Opcode (cmd) Decoding:**  
   Based on the 4-bit cmd, the corresponding operation is selected from a case structure. Arithmetic operations include:
   * Addition, subtraction (with or without cin)
   * Increment, decrement
   * Comparison
   * Multiplication

Logical operations include:

* + AND, NAND, OR, NOR, XOR, XNOR
  + NOT, right/left shifts
  + Rotate left/right

1. **Input Validation:**  
   If input\_valid is not set correctly, the operation is aborted and an error message is displayed with err = 1.
2. **Flag Calculation:**  
   Post-operation, the output flags are set based on the result:
   * cout for carry out
   * overflow for overflow detection
   * e, g, l for comparison flags
   * err for any exception
3. **Delay Handling:**  
   For expensive operations like multiplication, the design includes delay logic that requires delay1 to reach 2 before execution. This mimics pipelining or multi-cycle computation.

# Code Summary

The Verilog implementation is parameterized using width and c\_w, allowing easy scaling to higher bit-widths (e.g., 16-bit or 32-bit). The always @(posedge clk) block drives the operation control logic and the result generation.

The ALU includes:

* Arithmetic block (mode = 1)
* Logic block (mode = 0)
* Flag generation logic
* Delay handling logic
* Case-based operation decoding

Each operation is wrapped in if(input\_valid) checks to ensure correctness. Some key highlights:

* res = opa + opb for addition
* res = opa - opb - cin for subtract with carry
* Bitwise logic using &, |, ^, ~
* Signed comparisons using $signed()
* Manual overflow and carry detection
* Custom shift and rotate logic

Code is modular, clean, and synthesizable.

# Simulation Setup

* The ALU was simulated using Synopsy and Questasim and Verilog testbench.
* Randomized inputs for opa, opb, cmd, and cin were generated using $urandom.
* Each command was tested for valid and invalid inputs.
* The testbench included display logs and optional waveforms.
* Delay-based operations were verified by checking output only after delay cycles.
* No assertions or formal verification used.
* No FPGA testing was done.

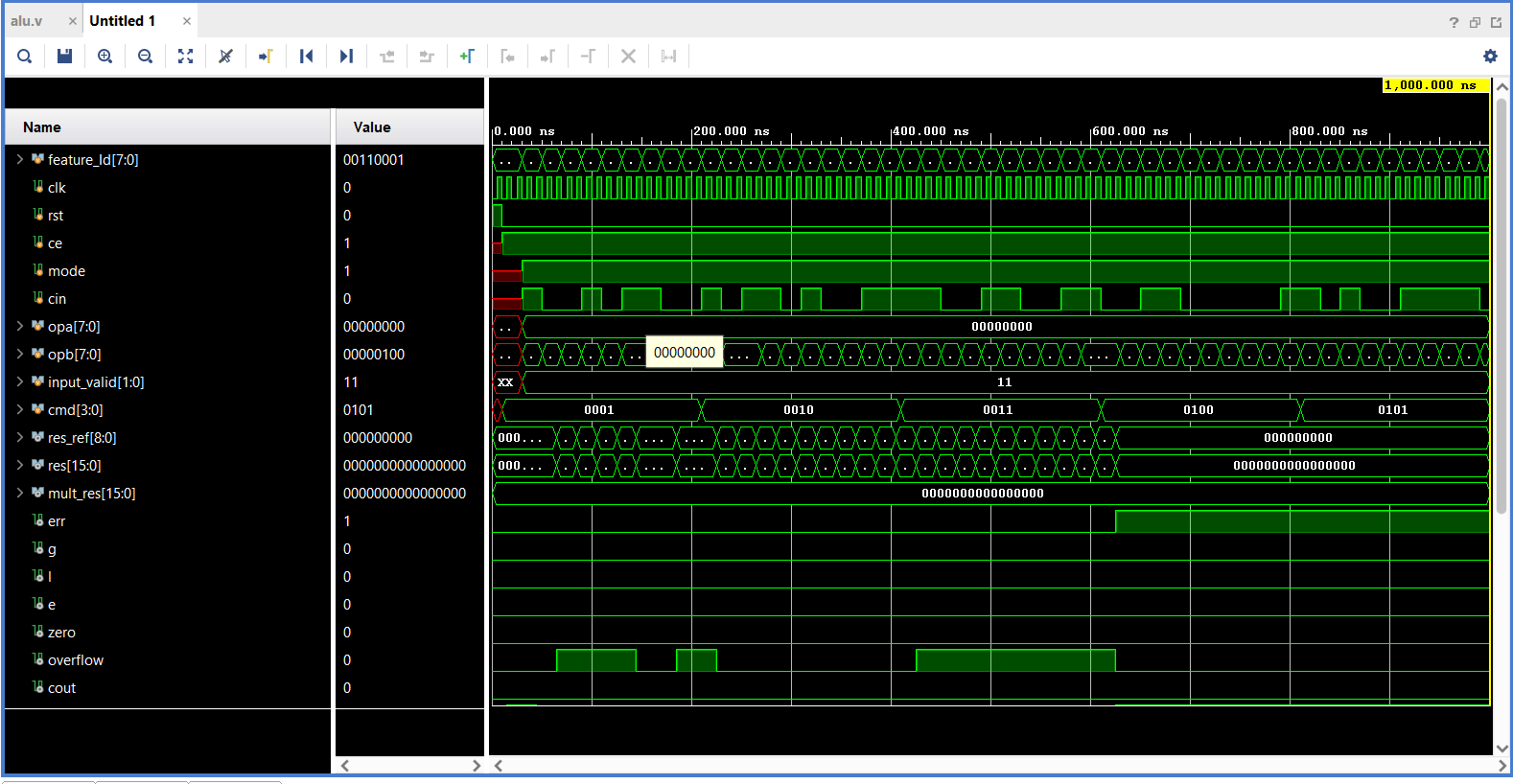
Simulation verified:

* Correct results
* Proper flags (g, l, e, overflow)
* Error handling
* Behaviour under invalid inputs

# Results

Simulation output confirmed:

* Arithmetic operations performed correctly for all valid input cases.
* Logical operations showed expected bitwise results.
* Status flags like overflow, greater, less, and equal worked as intended.
* Multiplication delay logic executed only after the right number of cycles.
* Error signal set correctly for invalid operations or unqualified inputs.



Submitted by:

Name:Vishnu Vardhan J

Emp Id:6077