# DESIGN DOCUMENT

## ARITHEMETIC LOGIC UNIT (ALU)

Pratiksha P Shetty
6093

# *INTRODUCTION*

An Arithmetic Logic Unit is a fundamental digital circuit used in every processor to perform basic arithmetic and logical operations. It handles tasks such as addition, subtraction, increment, decrement, and bitwise operations like AND, OR, XOR, and NOT. In modern digital systems, the ALU plays a key role not only in performing calculations but also in comparisons and decision-making processes. This makes it a vital component for executing instructions and ensuring smooth processor operations.

In the heart of every processor lies this powerful engine, the ALU. Acting as the brain behind decision-making and calculations, the ALU executes critical arithmetic and logical tasks that drive computing forward.

The ALU designed in this project is parameterized and supports a variety of command codes under both arithmetic and logical modes. It includes features such as comparison outputs (greater, less, equal), overflow and carry detection, and input validation control. The design also ensures robust handling of invalid conditions through an error flag.

This project focuses on designing and building a custom ALU using Verilog. The goal is not only to make it work correctly but also to keep the code easy to understand, reusable, and testable using simulations. By combining basic concepts from digital electronics with practical design in Verilog, this project shows how simple logic circuits can be turned into useful and reliable parts of a digital system.

# *OBJECTIVES*

The main objective of this project is to design and implement a parameterized ALU using Verilog Hardware Description Language. The ALU is expected to support a wide variety of arithmetic operations such as addition, subtraction, increment, and decrement, as well as logical operations including AND, OR, XOR, NAND, NOR, and NOT. Another important objective is to enable mode selection between arithmetic and logical functions using control signals.

The design aims to incorporate status flag outputs such as carry-out, overflow, greater-than, less-than, equal-to, and an error flag for handling invalid operations. Additionally, proper input validation through the INP_VALID signal is implemented to ensure correct functionality and avoid undefined behaviour. This project also focuses on developing modular, readable, and synthesizable Verilog code that can be tested using simulation tools and waveform analysis for verifying correctness and reliability.

# *SPECIFICATION*

This section provides the technical details of the ALU design, including the input and output signals, supported operations, and command codes for both arithmetic and logical modes.

i. *INPUT SIGNALS*

- CLK : Clock signal is used for synchronization of the circuit [1 bit].
- RST : Asynchronous, active high reset signal to initialize the system [1 bit].
- CE : Clock Enable signal to control active processing [1 bit].
- MODE : Control signal for selecting the operation mode [1-bit].
  - 1: Arithmetic operation.
  - 0: Logical operation.
- INP_VALID: Specifies the validity of input operands [2-bit].
  - 00: Both operands are invalid
  - 01: Only A is valid.
  - 10: Only B is valid.
  - 11: Both are valid.
- CMD: Parameterized command. Based on mode the command is selected [in this project its 4 bits].
- OP_A: Parameterized operand [8-bit in this project].
- OP_B: Parameterized operand [8-bit in this project].
- CIN: Carry-in signal for arithmetic operations involving carry [1-bit].

ii. *OUTPUT SIGNALS*

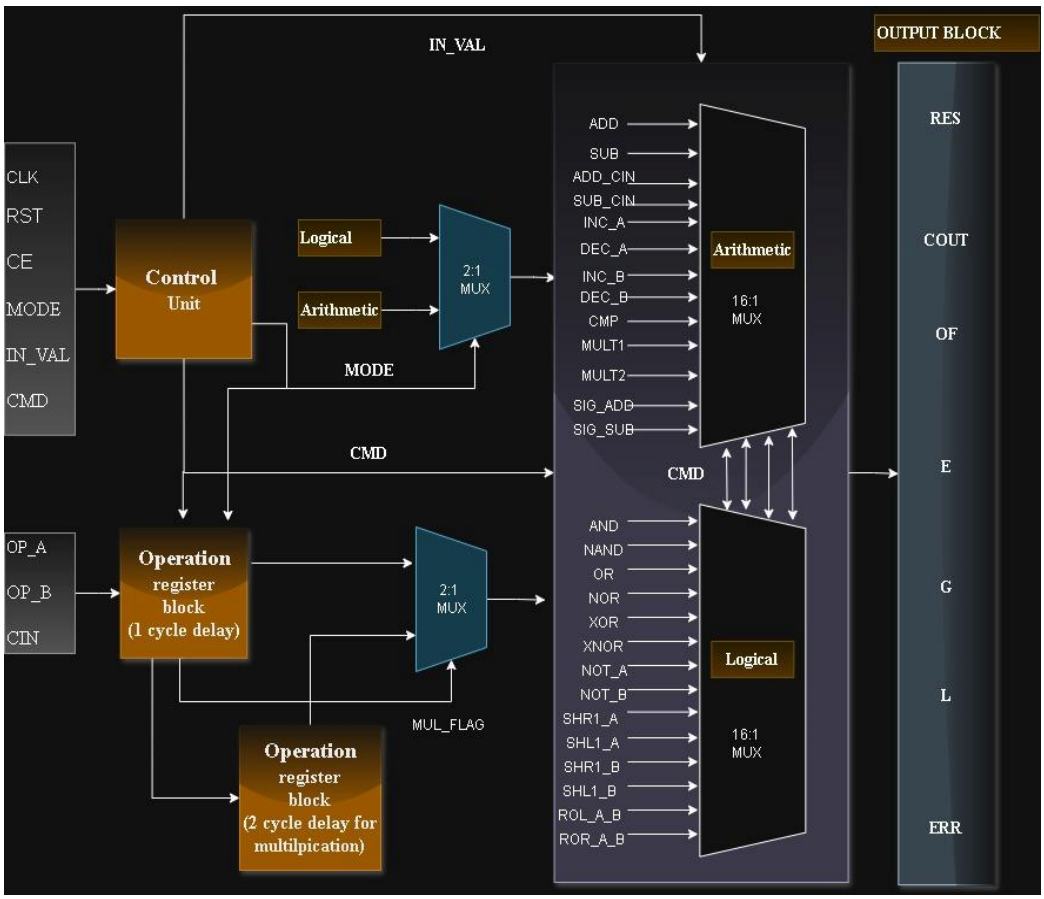
- RES: Parameterized result of the operation (bit-width is wider than input operands for all operations but for multiplication its 2 times the operand length).
- OFLOW: Overflow flag to indicate arithmetic overflow [1-bit].
- COUT: Carry-out signal from arithmetic operations [1-bit].
- G: Comparator outputs for greater-than result [1-bit].
- L: Comparator outputs for less-than result [1-bit].
- E: Comparator outputs for equal-to results [1-bit].
- ERR: Error flag set during invalid operations [1-bit].

The operations are carried out based on the mode and the command.

a. MODE =1
    - 0: ADD
    - 1: SUB
    - 2: ADD_CIN
    - 3: SUB_CIN
    - 4: INC_A (increment by 1)
    - 5: DEC_A (decrement by 1)
    - 6: INC_B (increment by 1)
    - 7: DEC_B (decrement by 1)
    - 8: CMP
    - 9:INC_MUL (increment both operands by 1 and then multiply them)
    - 10: SHIFT_MUL (left shift the operand A by 1 and then multiply by B).
    - 11: SIGNED_ADD
    - 12: SIGNED_SUB
b. MODE =0
    - 0: AND (bitwise)
    - 1: NAND
    - 2: OR (bitwise)
    - 3: NOR
    - 4: XOR
    - 5: XNOR
    - 6: NOT_A
    - 7: NOT_B
    - 8: SHR1_A (shifts right A by 1)
    - 9: SHL1_A (shifts left A by 1)
    - 10: SHR1_B (shifts right B by 1)
    - 11: SHL1_B (shifts left B by 1)
    - 12: ROL_A_B (rotate left the operand A based on the value of B. If the value of B goes beyond the length (operand_length-1) then error flag is raised).
    - 13: ROR_A_B B (rotate right the operand A based on the value of B. If the value of B goes beyond the length (operand_length-1) then error flag is raised).

# *ARCHITECTURE*

## System Architecture:




The ALU system architecture consists of multiple functional blocks that work together to perform arithmetic and logical operations based on the input control signals and operands. The design is organized into the following key components:

1. *Input Block*

   Includes all external inputs to the ALU:

- Control Signals: CLK, RST, CE, MODE, INP_VALID, CMD
- Operands: OPA, OPB, CIN

  These signals provide the necessary data and control to define what operation should be performed and when.

2. *Control Unit*

The Control Unit receives the control signals and interprets them. Based on the MODE signal:

- It selects between Arithmetic and Logical operation sets using a 2:1 multiplexer.
- It decodes the CMD input to activate the correct operation (e.g., ADD, SUB, AND, etc.).
- It controls operand selection, flag generation, and ensures that valid input combinations (INP_VALID) are processed.

3. *Operand Register Blocks*

There are two stages of operand registers:

- A 1-cycle delay block that buffers inputs (OPA, OPB, CIN) for stable operation.
- A 2-cycle delay block specifically used for multiplication operations, controlled by the MUL_FLAG. This ensures operands remain valid across multiple cycles required for multiply logic.
- These registers are essential for timing alignment and multi-cycle computations.

4. *Operation Logic*

The ALU supports:

- Arithmetic Operations (handled by a 16:1 multiplexer): ADD, SUB, INC, DEC, CMP, MULT1, MULT2, etc.
- Logical Operations (also handled by a 16:1 multiplexer): AND, OR, NOT, XOR, XNOR, NAND, NOR, SHL, SHR, Rotate, etc.
- The selected operation block is chosen using the MODE-based multiplexer, and the specific function is determined using the CMD signal.

5. *Output Block*

The final output block produces the following results:

- Result (RES)
- Carry Out (COUT)
- Overflow (OF)
- Comparison Flags: Greater (G), Less (L), Equal (E)
- Error Flag (ERR): Set when invalid conditions are detected (e.g., shift/rotate issues).

# *WORKING*

The ALU operates on the basis of clocked sequential logic, using always @(posedge clk) blocks to register input operands and control the timing of result generation. The control signals (clk, rst, ce, mode, cmd, and in_val) guide the behavior of the system and determine which operation is to be performed, when inputs are considered valid, and whether the operation belongs to arithmetic or logical mode.

*General Operation Flow*

1. Input Registration:

   On every rising clock edge (if ce = 1), the operand inputs (opa, opb) and carry-in (cin) are buffered into internal registers (tempa, tempb, tempcin). Signed versions of the operands (sign_a, sign_b) are also registered for signed arithmetic operations.

2. Operation Delay Management:

   For most arithmetic and all logical operations, the output is generated and stored in the res register on the second clock cycle after valid input.

   For multiplication operations (i.e., when cmd = 9 or cmd = 10 and mode = 1), operands are further delayed by one additional cycle (tempa_d1, tempb_d1) to ensure correct data stabilization. These operations provide output only on the third clock cycle from the time of valid input registration.

   *Arithmetic Mode (mode = 1):*

   *Based on the cmd signal and input validity:*

   Single operand operations like INC_A, DEC_A, INC_B, DEC_B use only tempa or tempb.

   Dual operand operations like ADD, SUB, ADD_CIN, CMP, SIG_ADD, SIG_SUB use both tempa and tempb, optionally tempcin, and update flags like cout, of, g, l, and e.

   Multiplication operations:

   - CMD 9: (tempa + 1) * (tempb + 1)
   - CMD 10: (tempa << 1) * tempb

These are executed using delayed operands tempa_d1 and tempb_d1.

*Logical Mode (mode = 0):*

Logical operations are selected based on the cmd and in_val. These include:

Bitwise operations: AND, OR, XOR, NAND, NOR, NOT

Shifting: logical shift left/right of tempa or tempb

Rotation: circular left or right shift of tempa using the shift amount from tempb[2:0]

An ERR flag is raised if invalid bits in tempb[7:4] are detected during rotation commands.

*Result and Flags:*

The result is stored in res. Output flags like cout (carry out), of (overflow), g, l, e (comparators), and err (error) are set based on the operation performed and the operand conditions.

*Reset Behavior:*

When rst is active (high), all internal registers and outputs are cleared to zero. This ensures the ALU returns to a known state.

*Timing:*

| Operation Type | Output Available At |
|---|---|
| Arithmetic / Logical ops | 2nd clock cycle |
| Multiplication (CMD 9/10) | 3rd clock cycle |

# *RESULT*

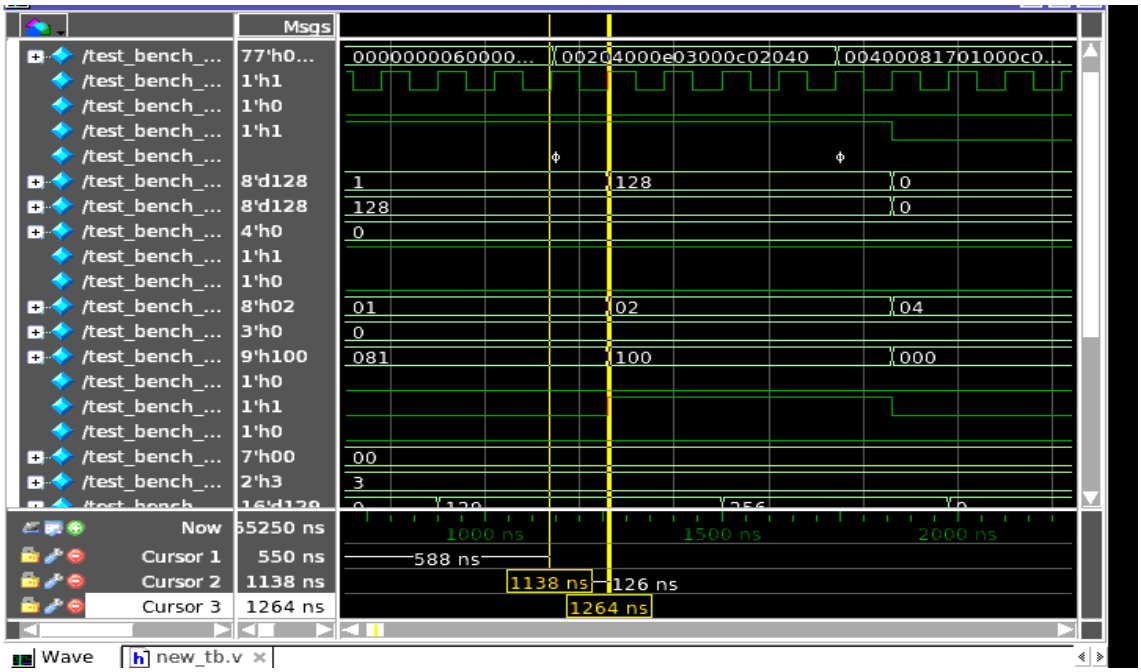



**Figure 1:** Output waveform for normal arithmetic operation

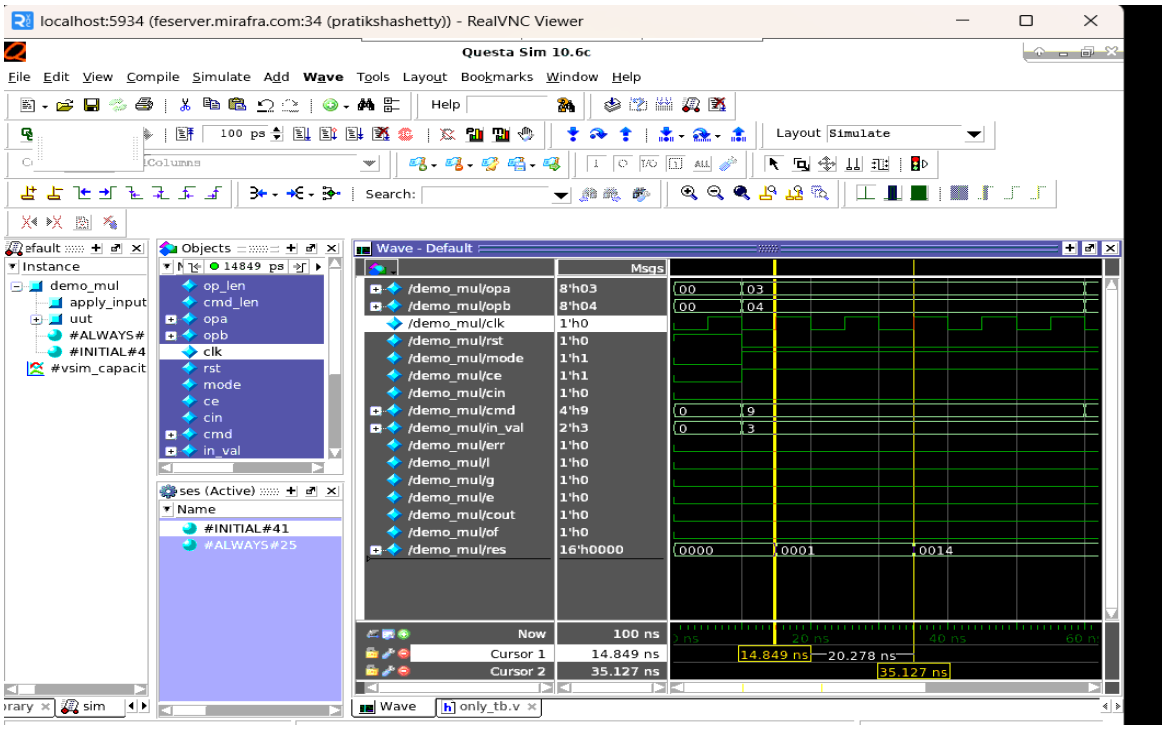



**Figure 2:** Output waveform for multiplication operation

The waveform captures shown illustrate the simulation results of the ALU for various operations. The first image demonstrates a normal arithmetic operation (add) where the output result (res) and relevant status flags are updated on the second clock cycle after valid input is provided. This confirms that the system processes most operations with a one-cycle delay due to input buffering.

The second waveform highlights a multiplication operation (CMD = 9 or 10 in arithmetic mode). As expected, the result is visible only on the third clock cycle, due to an extra one-cycle delay introduced by internal operand registers (tempa_d1, tempb_d1). This staged processing aligns with the design's behaviour and confirms that multiplication operations are handled in a pipelined manner.

Both waveforms confirm correct functionality and timing behavior as designed. The simulation validates the ALU's ability to handle different operations with appropriate delays and generate accurate result and flag outputs.

# *CONCLUSION*

The Verilog based ALU design successfully meets all the functional requirements outlined at the beginning of the project. It supports a wide range of arithmetic and logical operations, handles input validation effectively, and provides accurate result and flag outputs. The implementation uses a modular approach, with clearly defined control and data paths, making the design both readable and reusable. Simulation results confirm the correct behaviour of all supported operations, including the expected multi-cycle behaviour for multiplication instructions. Overall, the ALU design is fully testable and synthesizable, and it can be integrated into larger digital systems with minimal modifications.

# *FUTURE IMPROVEMENT*

In future versions of the ALU design, several enhancements are planned to improve performance, functionality, and reliability. One major improvement is the introduction of pipelined stages, which will enable the ALU to process multiple instructions in parallel by dividing the operation into smaller stages. This will significantly increase the throughput of the system. Additionally, more complex operations such as division, modulus, and floating-point arithmetic can be added to expand the range of supported functions. Increasing the operand width beyond the current 8-bit design will allow the ALU to handle larger data values, making it suitable for more advanced processing tasks. Lastly, formal verification techniques can be applied to mathematically prove the correctness of the design, ensuring robust and error-free operation across all possible input conditions.