ALU PROJECT

EMP ID -6091

PRATHIKSHA

INTRODUCTION

The Arithmetic Logic Unit (ALU) is an important part of any processor that performs basic calculations like addition, subtraction, and logical operations such as AND, OR, and XOR. These operations are essential for the functioning of digital systems. In this project, the ALU was designed using Verilog HDL, a hardware description language used to model digital circuits.

This ALU is made to be flexible by allowing parameters like data width and supported operations to be easily adjusted. It can handle both signed and unsigned numbers, which makes it suitable for different types of calculations. In addition to producing results, the ALU also generates useful status flags such as carry-out, overflow, and comparison indicators like greater than, equal, and less than.

To make sure the ALU works correctly, a testbench was created to verify its behavior. The testbench reads input values and expected results from a file and compares them with the actual output of the ALU. This helps to catch any errors and ensures that all operations work as intended. The testing method also makes it easy to add more test cases in the future, which helps improve the overall quality and reliability of the design.

OBJECTIVES

- Design and build a multi-purpose ALU using Verilog HDL that can handle both arithmetic and logic operations.
- Support several operations like addition, subtraction, multiplication, increment, decrement, and bitwise operations (AND, OR, XOR).
- Include extra features like shifting and rotating bits, which are useful in many digital applications.
- Make the ALU **configurable** so it can work with different data sizes (like 8-bit, 16-bit, or 32-bit), depending on system needs.
- Focus on a **modular design** so the ALU can be reused and easily updated or expanded in the future.
- Use simulation-based **testbenches** to check that the ALU works correctly with many different input combinations.
- Ensure the ALU is accurate and reliable under all test cases.

- Provide a strong starting point for use in larger projects like processors, embedded systems, or custom digital circuits.

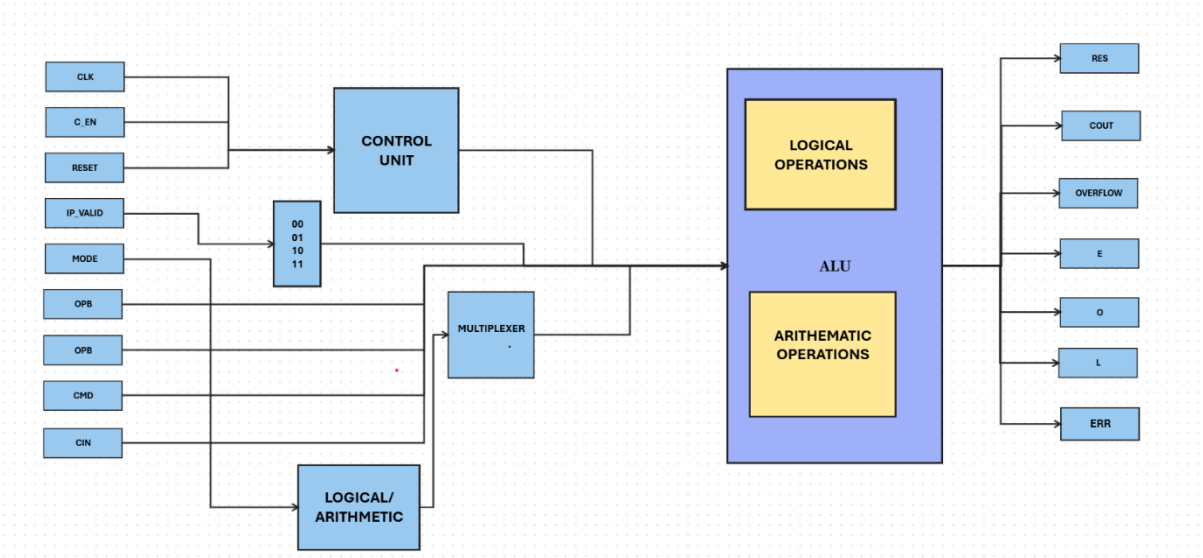
ARCHITECTURE

DESIGN:

The ALU works as a combinational logic block, meaning it does its calculations instantly without using a clock inside. But to fit well with other parts of a digital system that use a clock, the ALU is placed between registers that store inputs and outputs on clock signals. Inputs like numbers and commands are saved in input registers at the clock's rising edge. The ALU then processes the data, and the results are stored in output registers on the next clock edge. This adds a one-clock cycle delay, which helps keep the timing steady and makes the system more reliable.

The ALU can do different math and logic operations depending on a mode signal and a 4-bit command code. In arithmetic mode, it can add, subtract, multiply, and do increment or decrement operations. In logic mode, it can perform AND, OR, NOT, XOR, shifts, and rotations. It can work with either one or two input values, controlled by a small signal that tells if one or two inputs are valid. This lets the ALU handle simple operations (like NOT or increment) and more complex ones (like addition or comparison).

Inside, the ALU uses internal registers to hold temporary results and status signals. These status signals indicate things like overflow (when numbers get too big), carry out, comparisons (whether one number is greater, less, or equal to another), and errors. For signed numbers (which can be positive or negative), special checks make sure the results don't overflow incorrectly. The ALU also supports advanced operations like signed multiplication and shift-multiply. The one-clock delay helps the data move smoothly and keeps the ALU running reliably within larger digital systems.



TESTBENCH:

Testbench or Verification Environment (Simplified Explanation)

A **testbench** is used to make sure the **Design Under Test (DUT)** works correctly. It sends a series of inputs to the design, captures the output, and checks if the output matches what we expect.

In this project, the ALU design is tested using such a verification setup.

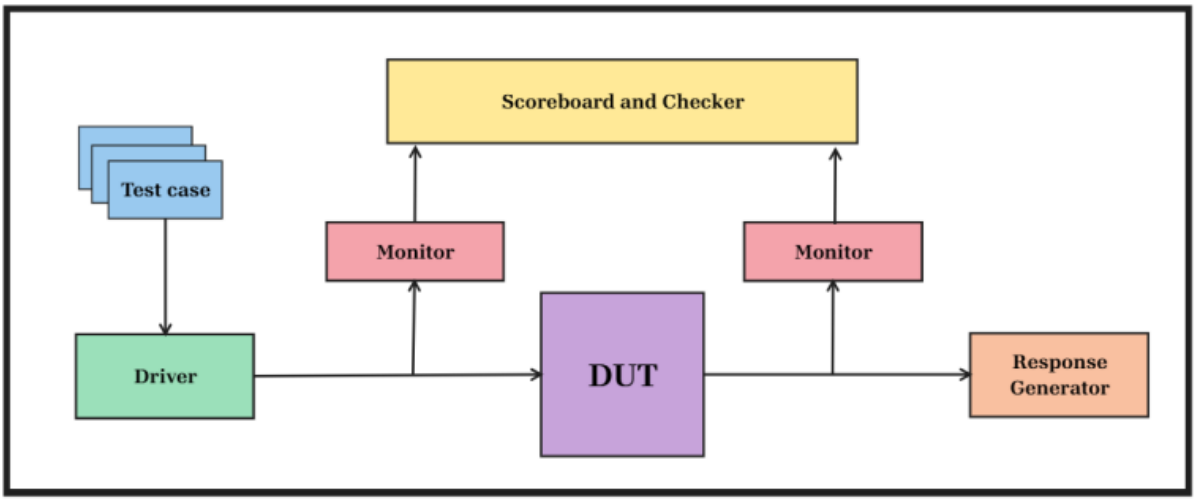
Main Components in the Testbench:

- **Test Cases:**
  These are sets of predefined input values used to check how the DUT behaves. Each test case sends specific data to the DUT and expects a certain result.
- **Driver:**
  The driver takes inputs from the test cases and sends them to the DUT as signals (bit-level/pin-level format).
- **DUT (Design Under Test):**
  This is the hardware module being tested. It receives inputs and generates outputs based on its internal logic.
- **Monitor:**
  The monitor watches the signals going in and out of the DUT. It captures the DUT's outputs and converts them back into a readable format for checking.
- **Response Generator:**
  This block predicts what the DUT should output based on the inputs. It creates the "expected" results.

- **Scoreboard:**
  The scoreboard compares the actual outputs from the DUT (via the monitor) with the expected outputs from the response generator. If the values match, the test passes. If not, the test fails and the mismatch is logged for debugging.

How It Works (Flow):

1. The **test case** generates input data.
2. The **driver** sends this data to the **DUT**.
3. The **DUT** processes the data and produces an output.
4. The **monitor** captures the output and converts it back to a usable format.
5. The **response generator** produces what the output *should* be.
6. Both the DUT's actual output and the expected output are sent to the **scoreboard**.
7. The **scoreboard** compares them and logs the result.
8. If there's a mismatch, it flags it for debugging.



WORKING

DESIGN:

The ALU takes input values on the rising edge of the clock, but only when the clock enable (ce) signal is high. When this happens, it stores the inputs and control signals inside internal registers to keep them stable for processing.

Based on the mode and valid signals, the ALU uses a 4-bit command code (cmd) to decide which operation to perform, such as addition, subtraction, or logical operations like AND and OR.

Most operations are done in a single cycle using combinational logic, which quickly calculates the result and sets condition flags like overflow, carry, or comparison outcomes. However, multiplication is a bit different — it takes 2 clock cycles to complete because it's a more complex operation.

To handle this, the ALU uses a simple pipeline design for multiplication. This means the multiplication operation is split into stages: the first stage starts the calculation, and the second stage finishes it. While one multiplication is finishing, the ALU can start a new operation in the next cycle. This pipelining helps keep the ALU efficient and ensures smooth flow of data without delays in other operations.

After calculation, results and flags are stored in output registers and become available on the output pins in the next clock cycle for regular operations, or after two cycles for multiplication. This setup helps the ALU work reliably and fit well into larger digital systems.

TESTBENCH:

- **Parameter-driven architecture** matching the ALU module.
- Reads binary test vectors from stimulus.txt file.
- Tasks:
    - driver(): Decodes and drives inputs to DUT.
    - monitor(): Collects actual outputs post-latching.
    - score_board(): Compares expected vs actual; stores result in scb_stimulus_mem.
    - gen_report(): Dumps pass/fail log per feature ID into results.txt.

Scoring Logic

- Results (RES, COUT, EGL, OFLOW, ERR) are packed and compared against expected values.
- Pass/Fail decision is logged with Feature ID for traceability.

RESULT

file:///fetools/work_area/frontend/Batch_10/prathikshagowda/final/covReport/ 70%

ocumentation Forums

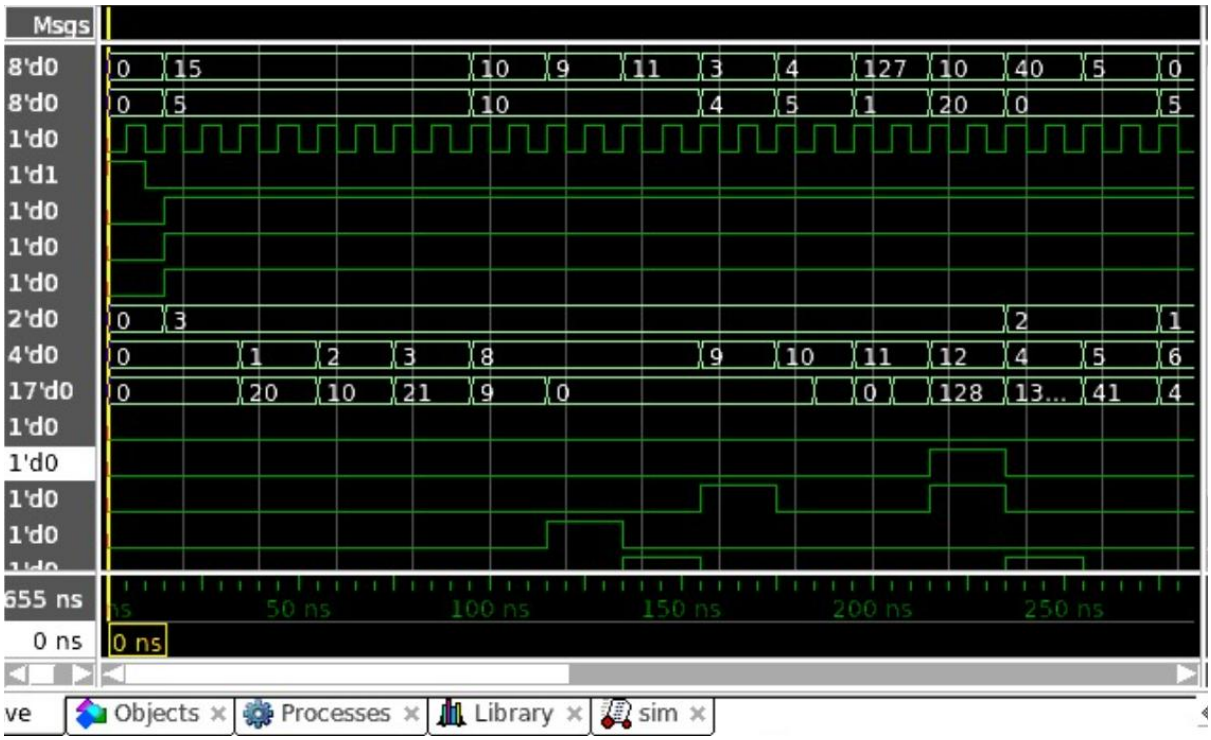
Questa Design Coverage

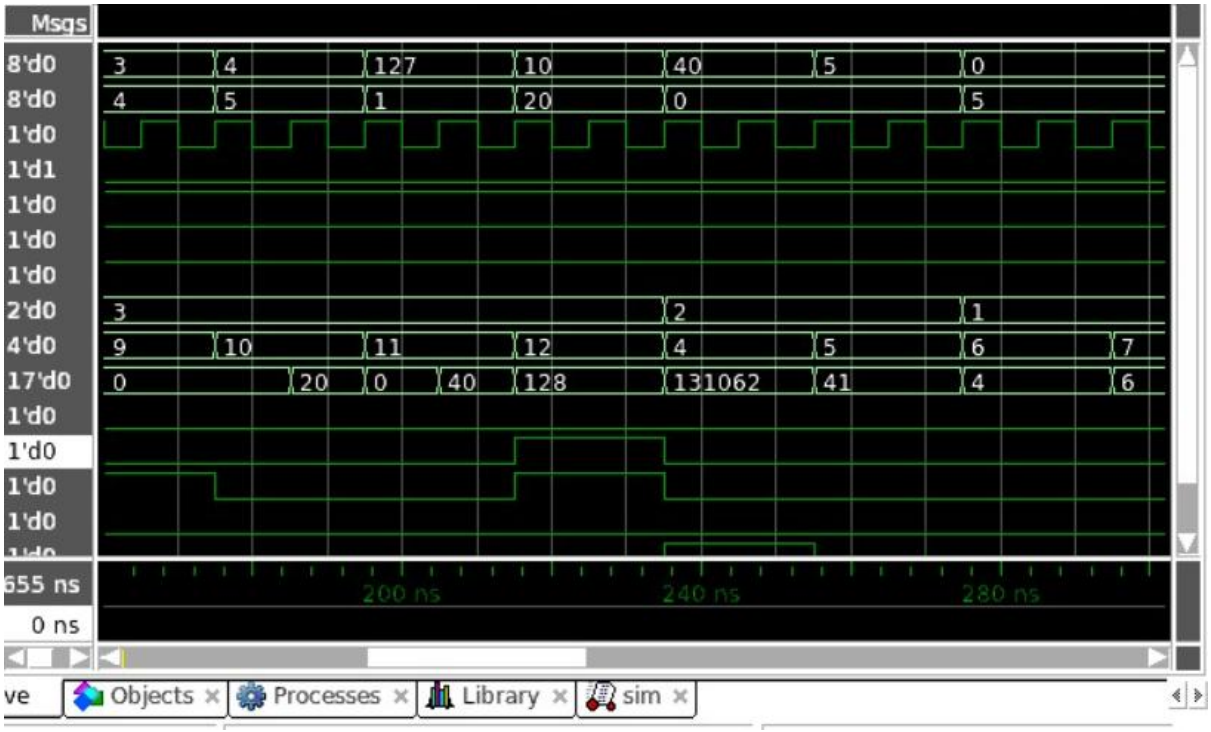
Scope: /test_bench_alu/dut

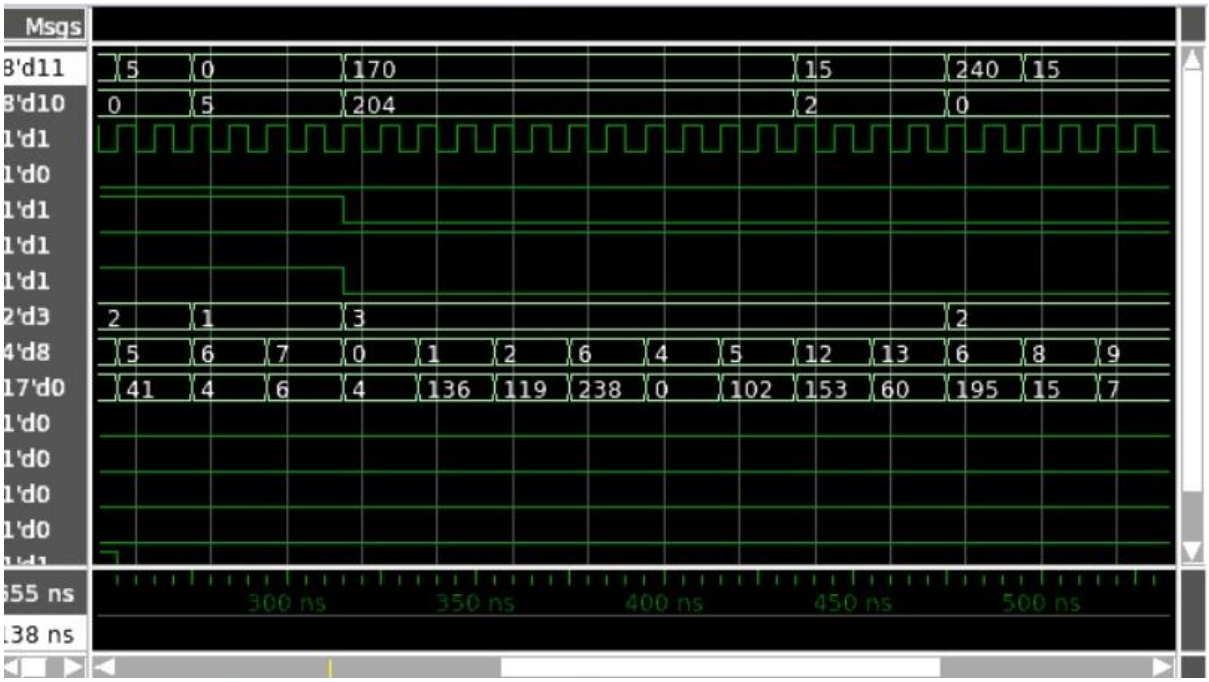
Instance Path:
/test_bench_alu/dut
Design Unit Name:
work.ALU
Language:
Verilog
Source File:
alu_tb.v

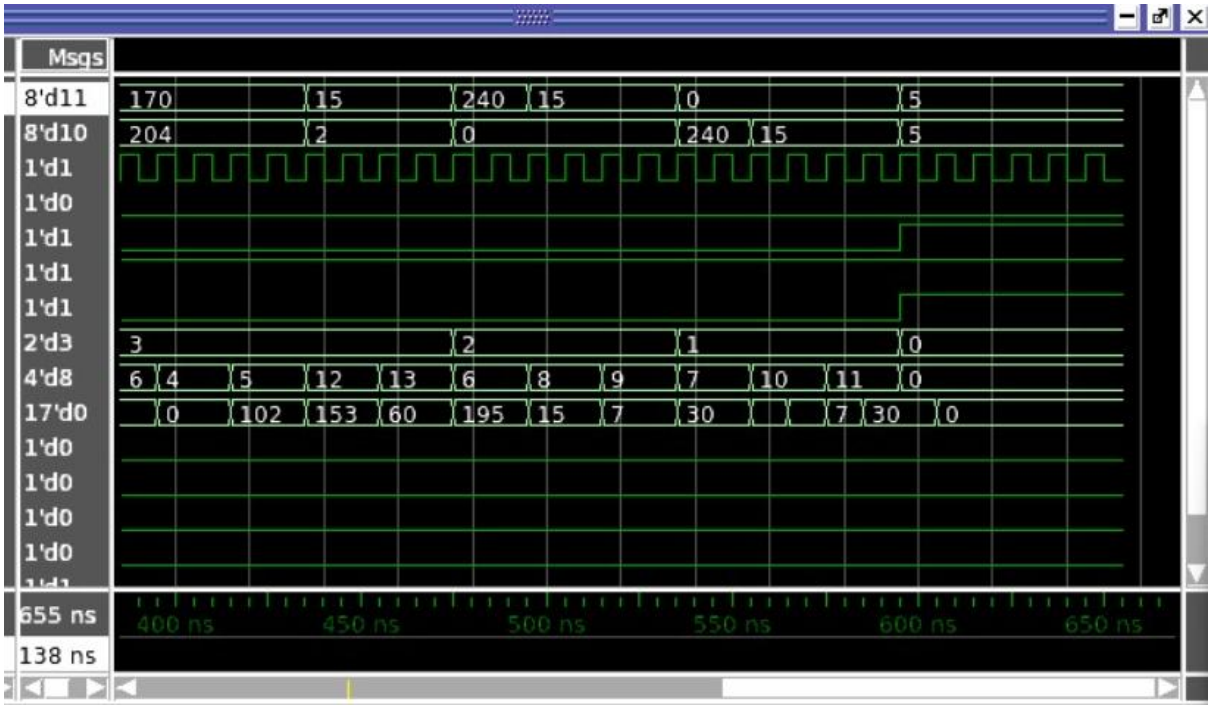
Local Instance Coverage Details:

Total Coverage:					100.00%	**100.00%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	135	135	0	1	100.00%	**100.00%**
Branches	84	84	0	1	100.00%	**100.00%**
FEC Expressions	6	6	0	1	100.00%	**100.00%**
FEC Conditions	3	3	0	1	100.00%	**100.00%**
Toggles	260	260	0	1	100.00%	**100.00%**




Msgs
8'd0
8'd0
1'd0
1'd1
1'd0
1'd0
1'd0
2'd0
4'd0
17'd0
1'd0
1'd0
1'd0
1'd0
655 ns
0 ns
200 ns
240 ns
280 ns
Objects
Processes
Library
sim


Msgs
8'd11
8'd10
1'd1
1'd0
1'd1
1'd1
1'd1
2'd3
4'd8
17'd0
1'd0
1'd0
1'd0
1'd0
655 ns
.38 ns
300 ns
350 ns
400 ns
450 ns
500 ns



CONCLUSION

The ALU has been successfully implemented and thoroughly tested to confirm that it meets all functional requirements. It is capable of performing a wide range of arithmetic and logical operations while also managing important status flags such as carry, overflow, and comparison results.

The verification process was carried out using a structured testbench that feeds in different test cases from an input file. Each output was compared with expected results using a scoreboard method to check for correctness. This approach helped in identifying any errors quickly and ensured that the ALU performs accurately and consistently under various input conditions.

Overall, the combination of a reliable design and a strong testing setup ensures that this ALU can be confidently integrated into larger digital systems, offering dependable performance and easy scalability for future improvements.

FUTURE IMPROVEMENT

- **Speed Optimization**: The ALU can be made faster by breaking it into smaller stages using pipelining. This helps handle more operations quickly and reduces delays.

- **More Features**: In the future, we can include support for floating-point numbers, complex number calculations, and even encryption-related operations to handle more advanced tasks.
- **Smarter Testing**: We can use intelligent tools to automatically create and run test cases. This would save time, catch more issues early, and make the debugging process much easier.

With these changes, the ALU can become more powerful and efficient, and be better suited for complex digital systems while still giving accurate and fast results.