ARITHMETIC LOGIC UNIT DESIGN

PROJECT

Submitted by:

Kanthimathi C

6080

TABLE OF CONTENTS

1.INTRODUCTION:

Serving as the computational backbone of the CPU, the Arithmetic Logic Unit (ALU) manages essential arithmetic and logical processes. These include operations such as addition, subtraction, AND, OR, NOT, XOR, comparisons, and shifts. The ALU plays a central role in determining the computational power and efficiency of a processor. Its performance directly influences the overall speed, responsiveness, and capability of the system.

Key design features include an input valid signal to ensure reliable computation, a clock enable (ce) for synchronizing data latching, and command (cmd) and mode inputs for selecting between arithmetic and logical operations. The ALU can execute a wide array of operations, including signed and unsigned arithmetic, logical shifts, rotates, and comparison operations. On the output side, the ALU provides a 2×width result output to accommodate extended precision, particularly useful for multiplication and signed arithmetic. It also generates multiple status flags, including carry-out (Cout), overflow (oflow), and error (Err), alongside comparison flags like greater-than (G), less-than (L), and equal (E), making the ALU suitable for conditional operations and decision-making within control logic.

In summary, this project delivers a versatile and efficient ALU that meets the performance demands of modern digital systems. Its parameterized architecture promotes reusability, adaptability, and ease of integration, making it a strong candidate for both academic research and practical implementation in embedded or custom processor designs.

2.OBJECTIVE:

The objective of this project is to design, implement, and verify

- A multi-functional Arithmetic Logic Unit (ALU) using Verilog HDL.
- The ALU is intended to support a wide range of arithmetic and logical operations, including addition, subtraction, multiplication, increment, decrement, and various bitwise operations such as AND, OR, XOR, and shift/rotate functionalities.
- The goal is to create a parameterized and scalable ALU that can be configured to operate with different data widths (8-bit, 16-bit, 32-bit), offering flexibility for use in multiple digital systems and processor architectures.

The project emphasizes modular design principles, ensuring easy scalability and reusability. Verification is achieved through simulation testbenches, where the ALU is rigorously tested under various input conditions to confirm functional correctness and robustness. This design provides a strong foundation for further development in more complex CPUs, microcontrollers, or embedded system projects.

3.ARCHITECTURE:

The architecture of the ALU comprises input operand registers, control logic for opcode decoding, and output logic capable of generating status flags such as carry, overflow, error, and comparison flags (greater, less, equal). These components work in coordination to ensure correct operation execution based on the command and mode inputs.

3.1 Design Architecture:

The ALU is implemented as a combinational logic block that performs operations without depending on the clock signal internally. However, for smooth integration with synchronous digital systems, the ALU is placed between clocked input and output registers. Input operands and control signals are stored in input buffer registers on the rising clock edge, the combinational ALU logic processes the data through result buffers, and outputs are stored in output registers on the next clock edge. This creates a one-clock cycle delay that improves timing stability and reliability in larger digital systems.

The ALU executes arithmetic and logical operations based on a mode signal and 4-bit command code. In arithmetic mode, it performs addition, subtraction, increment, decrement, multiplication, and signed calculations, while in logical mode, functions like AND, OR, NOT, XOR, shifting, and rotation are applied. It accepts one or two operands (OPA, OPB), controlled by a 2-bit INP_VALID signal, allowing single-input (NOT, INC, DEC) or dual-input (ADD, SUB, CMP) operations. The combinational logic processes data based on the command and mode, storing intermediate results in internal registers.

Special status signals (OFLOW, COUT, G, L, E, ERR) handle overflow, carry propagation, comparison results (equal, greater, less), and error detection. Signed operations (S_ADD, S_SUB) ensure proper overflow management using sign bit checks and saturation limits. Additional functions, including signed multiplication, rotation, and shift-multiply, are supported, with a one-clock delay maintaining smooth data transitions and reliable execution in digital systems.

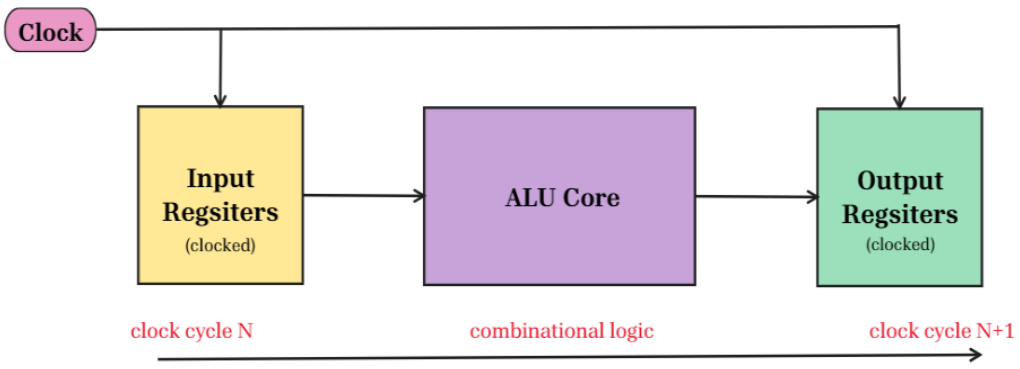

Figure 1. Overall Block diagram

This block diagram (Figure 1) describes my overall architecture, how the buffers are used in inputs and outputs to handle 1 clock cycle delay and synchronization. Below (Figure 2) diagram describes detailed connection of ALU design.

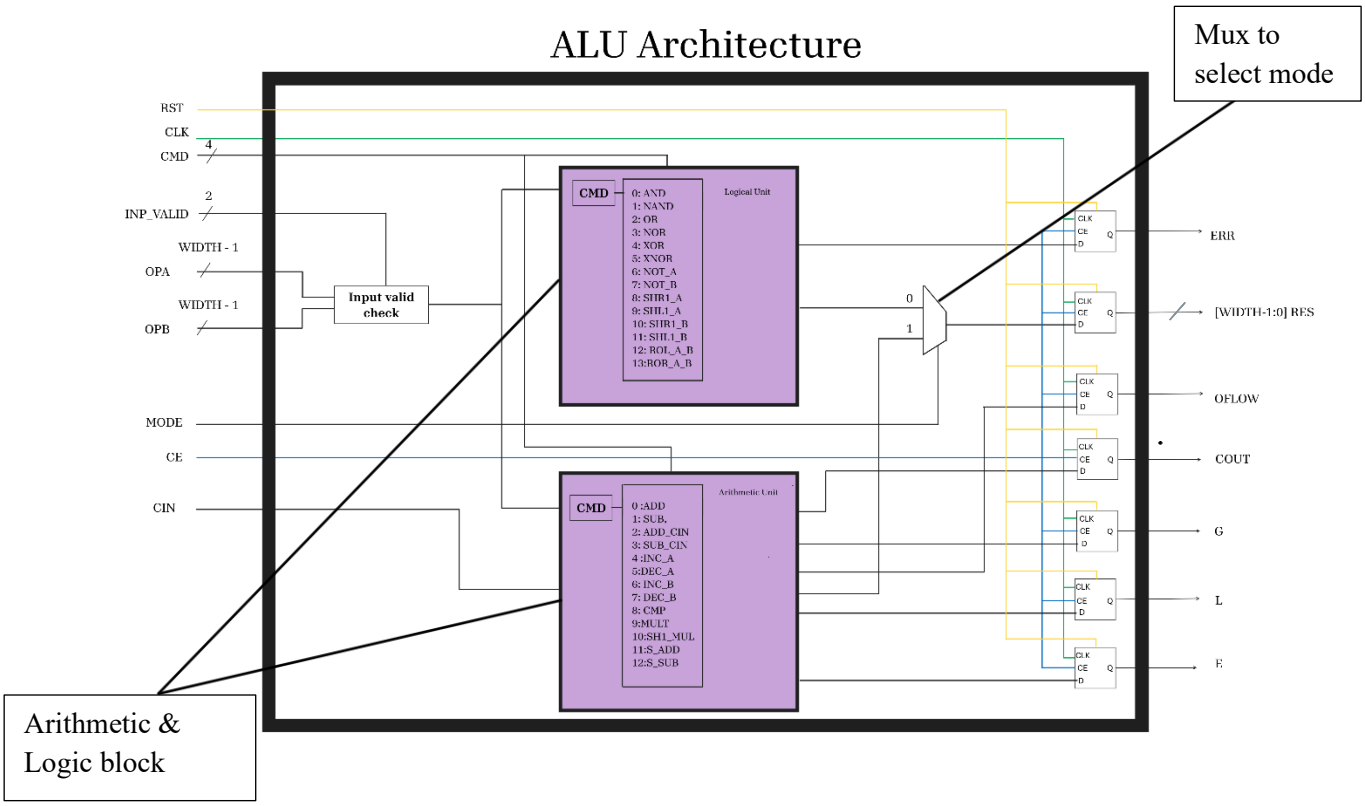

Figure 2.Design Architecture

Pin Description:

PIN	DESCRIPTION
RST	Asynchronous active-high reset, this is used to bring the system to a known state, here it is 0.
CLK	Provides the timing reference for sequential operations within the design.
CE	Used to latch inputs and outputs.
INP_VALID	Ensures the integrity of incoming data.
MODE	Determines the operational mode of the circuit. Used to switch between functional units. (arithmetic & logical)
CMD	specifies the exact operation to perform within the selected mode (e.g., ADD, SUB, AND, OR). Implemented as an opcode.
OPA	Input 1 operand of width bit wide for the processing unit.
OPB	Input 2 operand of width bit wide for the processing unit.
CIN	Carry Input for processing unit. Used in arithmetic operations like addition.

Table 1.Input pin description

PIN	DESCRIPTION
ERR	Indicates an error condition during operation, such as illegal command or invalid input. Useful for fault detection and handling.
RES	The primary result of alu of 2*width bit wide.
OFLOW	Indicates that the result of an arithmetic operation has exceeded the range representable by the data format
COUT	The carry-out bit from an arithmetic operation, typically used to indicate unsigned overflow or to propagate carry in multi-bit adders.
E	Set if the result of a comparison operation (OPA == OPB) is true. Used in conditional branching or logic decisions.
G	Set if the result of a comparison operation (OPA > OPB) is true. Used in conditional branching or logic decisions.
L	Set if the result of a comparison operation (OPA < OPB) is true. Used in conditional branching or logic decisions.

Table 2.Output pin description

These are my inputs and outputs to the ALU and their description given appropriately in table 3.1 and 3.2.

3.2 Testbench architecture:

Testbench or Verification Environment is used to check the functional correctness of the Design Under Test (DUT) by generating and driving a predefined input sequence to a design, capturing the design output, and comparing with-respect-to expected output. In this project, my design is tested using the verification environment as shown in figure 3.

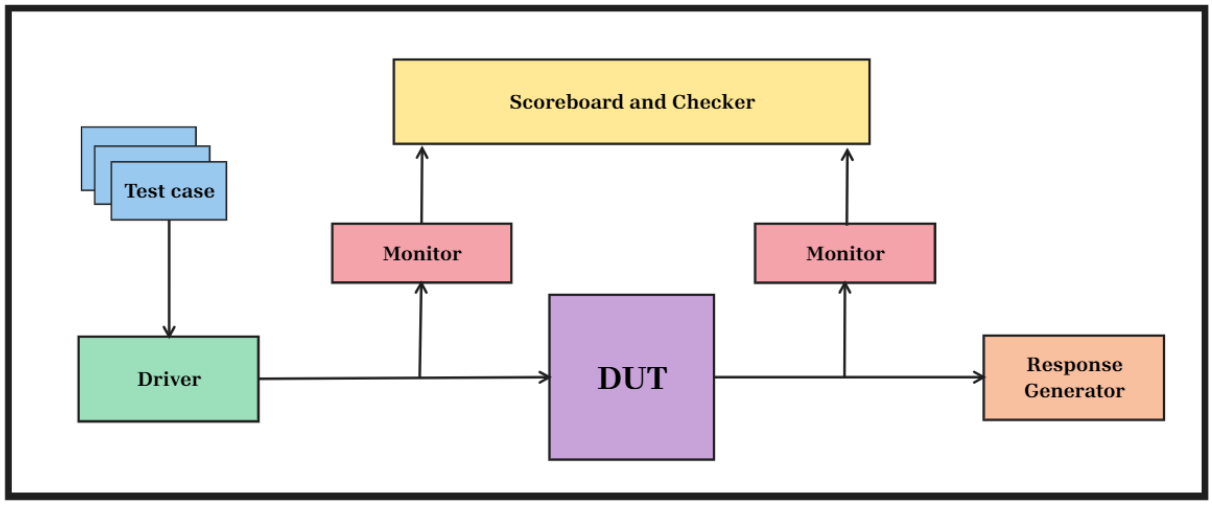

Figure 3.Test Bench architecture

Test cases: A simulation environment created to verify the functionality of a design (DUT - Design Under Test). It generates stimulus and checks the output of the DUT.

Driver: Receives the stimulus from a generator (ie. Test case) and drives the packet level data inside the transaction into pin level (to DUT).

DUT: The actual hardware module or system being tested and verified within the testbench.

Monitor: Observes pin level activity on interface signals and converts into packet level which is sent to the components such as scoreboard.

Scoreboard: Receives data items from monitors and compares them with expected values. Expected values can be either golden reference values or generated from the reference model.

Response generator: creates expected responses to DUT transactions, often predicting outputs. It helps validate DUT output correctness by enabling comparisons or driving return data.

The test case generates stimulus and sends it to the driver. The driver converts this into pin-level signals and drives them into the DUT. The DUT processes the inputs and produces outputs. These outputs are observed by the monitor, which converts them back to transaction-level data. The response generator creates expected responses based on the input transactions. Both actual and expected outputs are sent to the scoreboard, which compares them to check for correctness. The results are logged, and any mismatches are flagged for debugging. This flow ensures functional validation of the DUT through coordinated interaction of all components.

4. WORKING:

4.1 Execution of ALU operations (Design):

4.1.1 Input Handling:

The ALU receives two operands, OPA and OPB, which are the input values to be processed. These inputs are validation using the INPUT_VALID signal. The MODE signal determines whether the operation is arithmetic (MODE = HIGH) or logical (MODE = LOW). The CMD (Command) signal specifies the operation to be performed by the ALU, such as addition, subtraction, logical AND, logical OR, etc. Additional control signals, such as CIN (Carry Input), may be provided to assist certain operations, like addition with carry propagation.

INPUT COMMANDS

CMD	OPERATION	DESCRIPTION
0	ADD	Adds operand A and operand B.
1	SUB	Subtracts operand B from operand A.
2	ADD_CIN	Adds operand A, operand B, and a carry-in bit.
3	SUB_CIN	Subtracts operand B and carry-in from operand A.
4	INC_A	Increments operand A by 1.
5	DEC_A	Decrements operand A by 1.
6	INC_B	Increments operand B by 1.
7	DEC_B	Decrements operand B by 1.
8	CMP	Compares A and B; used for setting flags (like greater, equal, less).
9	INCx2_MUL	Increments both A and B by 1, then multiplies them.
10	SHL_Ax1_MUL_B	Left shifts A by 1, then multiplies with B.
11	ADD_FLAGS	Adds signed/unsigned A and B; sets flags (carry, overflow, zero, etc).
12	SUB_FLAGS	Subtracts signed/unsigned B from A; sets flags and compares A-B.

Table 3. Arithmetic mode commands

CMD	OPERATION	DESCRIPTION
0	AND	Bitwise AND of A and B.
1	NAND	Bitwise NAND of A and B.
2	OR	Bitwise OR of A and B.
3	NOR	Bitwise NOR of A and B.
4	XOR	Bitwise XOR of A and B.
5	XNOR	Bitwise XNOR of A and B.
6	NOT_A	Bitwise NOT of A.
7	NOT_B	Bitwise NOT of B.
8	SHR1_A	Shifts operand A right by 1 bit.
9	SHL1_A	Shifts operand A left by 1 bit.
10	SHR1_B	Shifts operand B right by 1 bit.
11	SHL1_B	Shifts operand B left by 1 bit.

Table 4. Logical mode commands

4.1.2. Clocking and Latching:

The ALU operation is synchronized with a clock signal, which provides the timing for the circuit. The CE (Clock Enable) signal is used to control when the input operands and the CMD signal are latched into the ALU. When CE is high, the input operands (OPA and OPB) and the CMD signal are captured and held within the ALU, allowing the operation to be performed. The CE signal acts as a gating mechanism, ensuring that the inputs are only processed when the clock is enabled, and the data is valid.

4.1.3. Processing stage:

The ALU receives two operands (OPA, OPB) along with a command (CMD) that defines the required operation and INPUT_VALID which ensures the data integrity. The mode signal (MODE) determines whether the ALU performs arithmetic (MODE = HIGH) or logical (MODE = LOW) operations. Additional control signals such as CIN (carry input) assist in arithmetic tasks like addition.

Arithmetic Operations (MODE = HIGH):

Addition (ADD): Uses binary addition, incorporating CIN if carry propagation is required.

Subtraction (SUB): Implements two's complement method where OPA - OPB is computed as OPA + (~OPB + 1).

Increment (INC) and Decrement (DEC): Modifies the operand by adding or subtracting 1.

Multiplication (MUL): Employs shift-add multiplication or other relevant methods to calculate the product.

Logical Operations (MODE = LOW):

Bitwise Operations: AND, OR, XOR, and NOT functions are performed directly on the operand values.

Shifting Operations:

Shift Left (SHL) and Shift Right (SHR): Move bits accordingly while inserting zeros into vacant positions.

Rotate Left (ROL) and Rotate Right (ROR): Circularly shifts bits within a fixed-width register.

4.1.4 Status Flag Handling:

Overflow (OFLOW): Activated if the result of an arithmetic operation surpasses the representable range.

Carry-out (COUT): Propagates extra bits during multi-bit arithmetic operations.

Comparison Flags (E, G, L):

- **Equal (E)**: Set when OPA == OPB.
- **Greater (G)**: Set when OPA > OPB.
- **Less (L)**: Set when OPA < OPB.

Error (ERR): Raised if an invalid operation occurs, ensuring reliable fault detection and system integrity.

4.1.5 Output Handling:

Once computation completes, the result (RES) is stored in the output buffer registers. The relevant status flags are updated based on the outcome of the operation, including conditions like carry out, overflow, and comparison results. The output is

synchronized using clocked registers, ensuring a clean transition and preventing glitches in digital circuits.

4.2 Testbench

4.2.1 Reference model:

To verify the Design Under Test (DUT), a testbench is used that includes a reference model (golden model), which serves as a functional representation of the DUT. This reference model generates an expected response for a given stimulus, ensuring correctness by comparing actual outputs against predefined expected results

4.2.2 Stimulus generator:

A stimulus generator is responsible for creating input test cases formatted according to Figure 4. The generated stimulus data is saved as stimulus.txt, storing structured packets for verification. These packets contain different parameters required for DUT validation.

PACKET DESCRIPTION

TEST CASE FORMAT (Stimulus)

PACKET HEADER	FEATURE ID	RESERVED BIT	RST	CE	IP_VALID	MODE	CMD	OPA	OPB	CIN	EXPECTED RESULT	EXPECTED EGL	EXPECTED OFLOW	EXPECTED ERR	
WIDTH (in bits)	w	2	1	1	2	1	4	w	w	1	2*w	3	1	1	TOTAL = 6w + 18

RESPONSE PACKET FORMAT

PACKET HEADER	CURRENT TEST CASE	RESULT	COUT	EGL	OFLOW	ERR	
WIDTH (in bits)	6w + 18	2 * w	1	3	1	1	TOTAL = 8w + 24

Figure 4. Stimulus format

The stimulus file (stimulus.txt) is fed into the testbench (tb), where all relevant fields are extracted using indexing techniques. The DUT is instantiated in the testbench environment. A driver module then takes the extracted input fields and drives them to the DUT, ensuring correct data transmission.

Expected Data Packet: The testbench extracts expected results from the stimulus input file. These results are structured into an expected data packet that matches the format of the actual DUT output packet.

Actual Data Packet: This comes from the DUT outputs which are structured in a particular order like expected data packet. It is formatted as {RES, COUT, EGL, OFLOW, ERR}.

Scoreboard and Checker Mechanism:

The scoreboard receives both the actual output packet from the DUT and the expected data packet generated from the stimulus. The checker then compares these two packets, verifying accuracy and functionality. A report is generated indicating whether the test case passed or failed, allowing for debugging and validation of the DUT. Each test case can be uniquely identified using a Feature ID, ensuring traceability and systematic debugging.

By following this structured approach, the testbench effectively verifies DUT functionality, ensuring robust validation through comparison mechanisms, indexing methods, and structured stimulus processing.

5. RESULTS:

5.1 Waveform of ALU

Waveform of my ALU after applying stimulus:

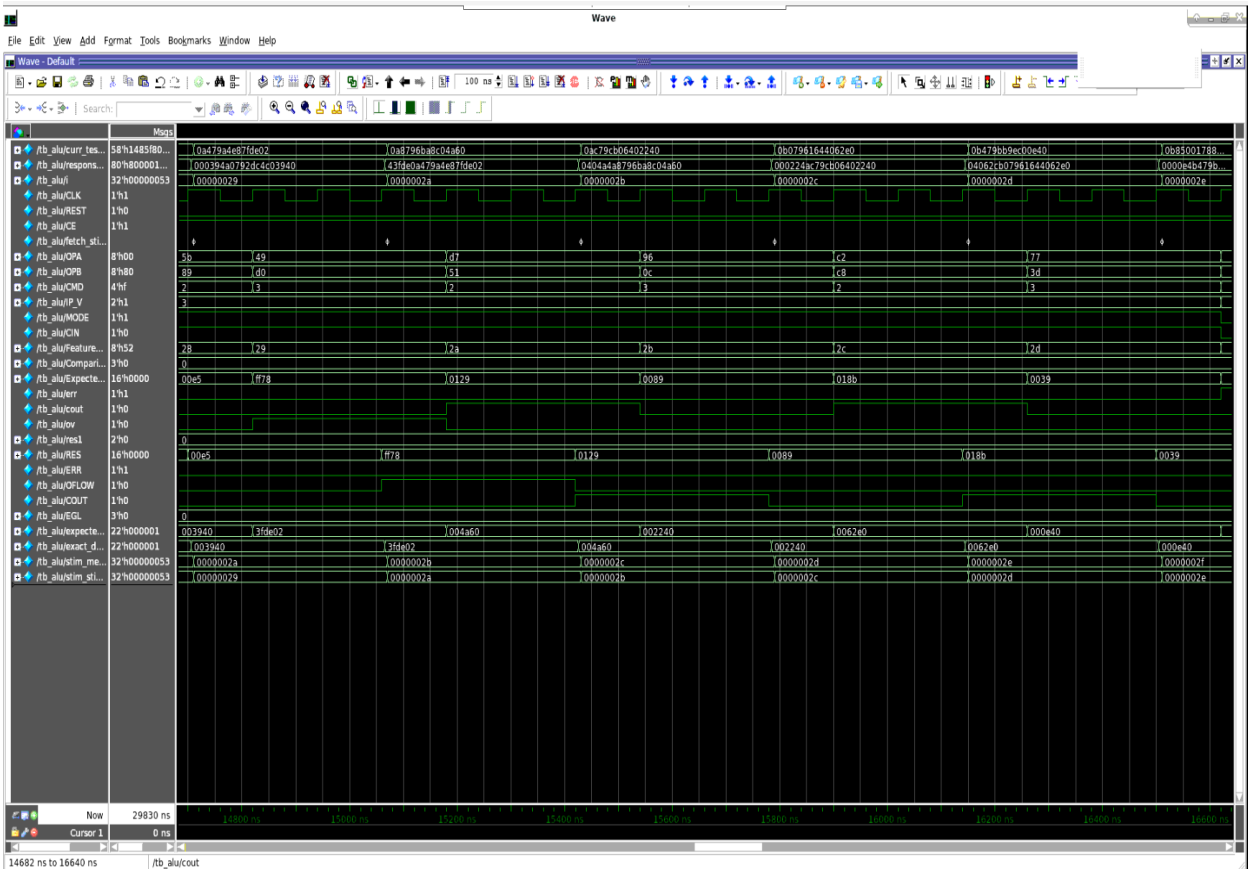

Figure 5. waveform 1

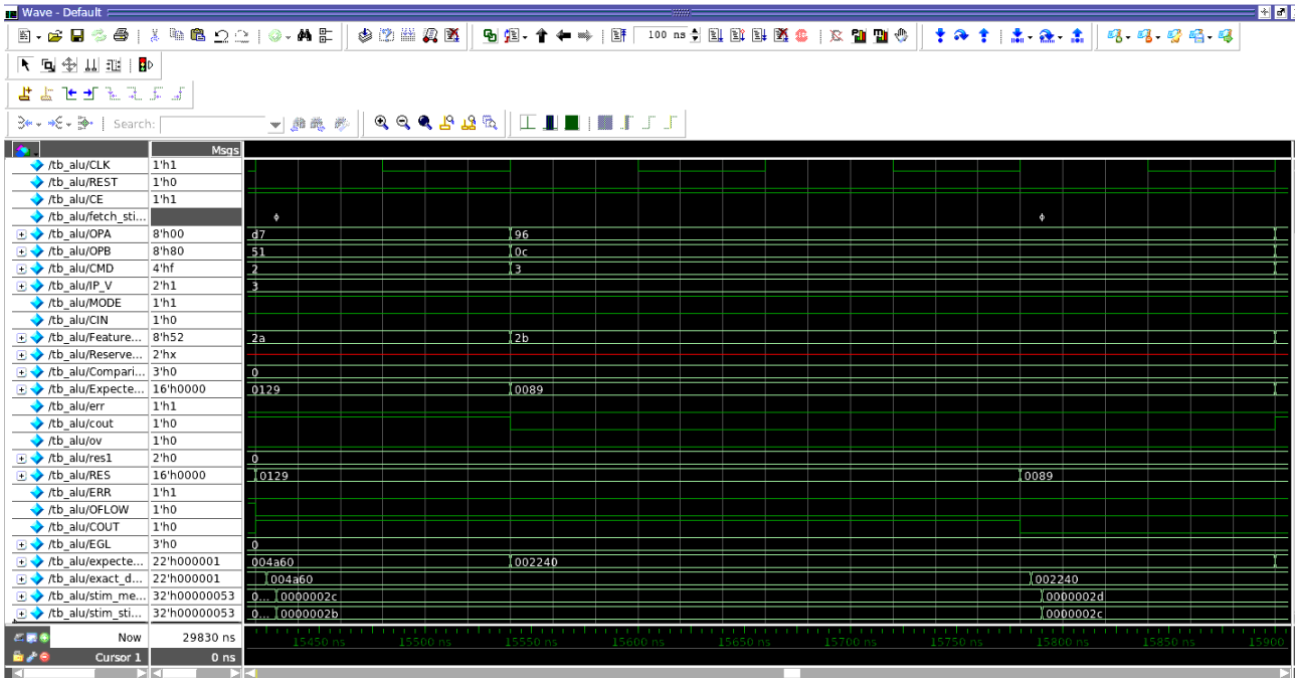

Figure 6. waveform 2

Individual commands:

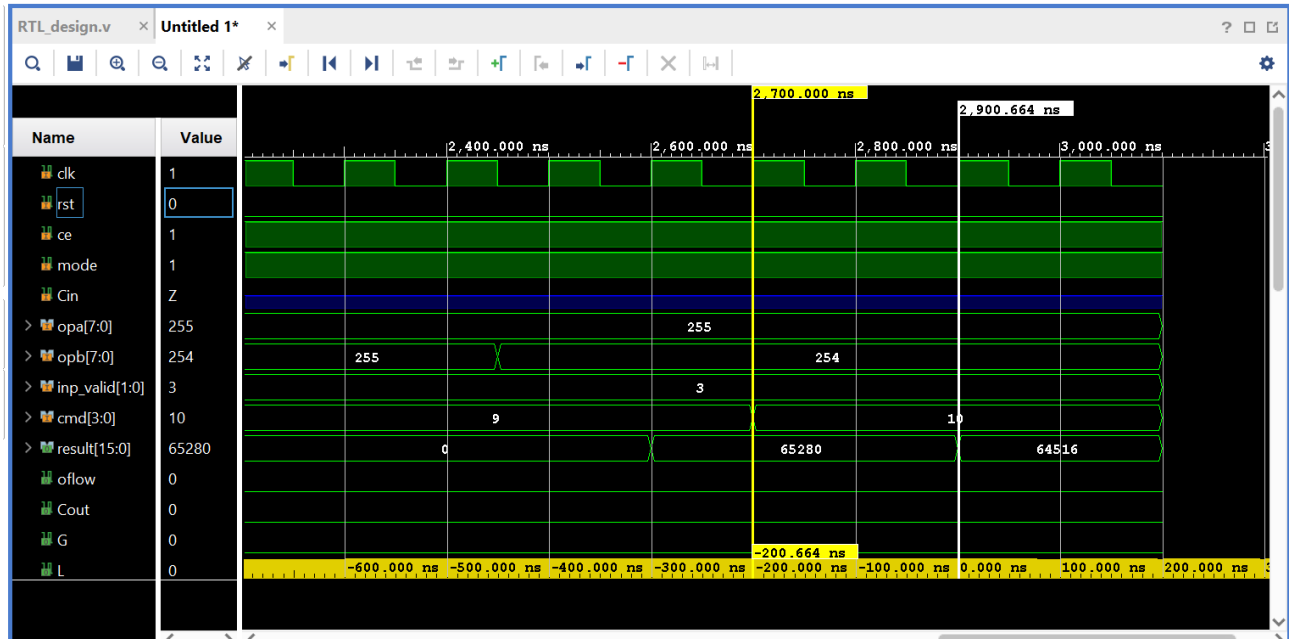

Figure 7. Multiplication results

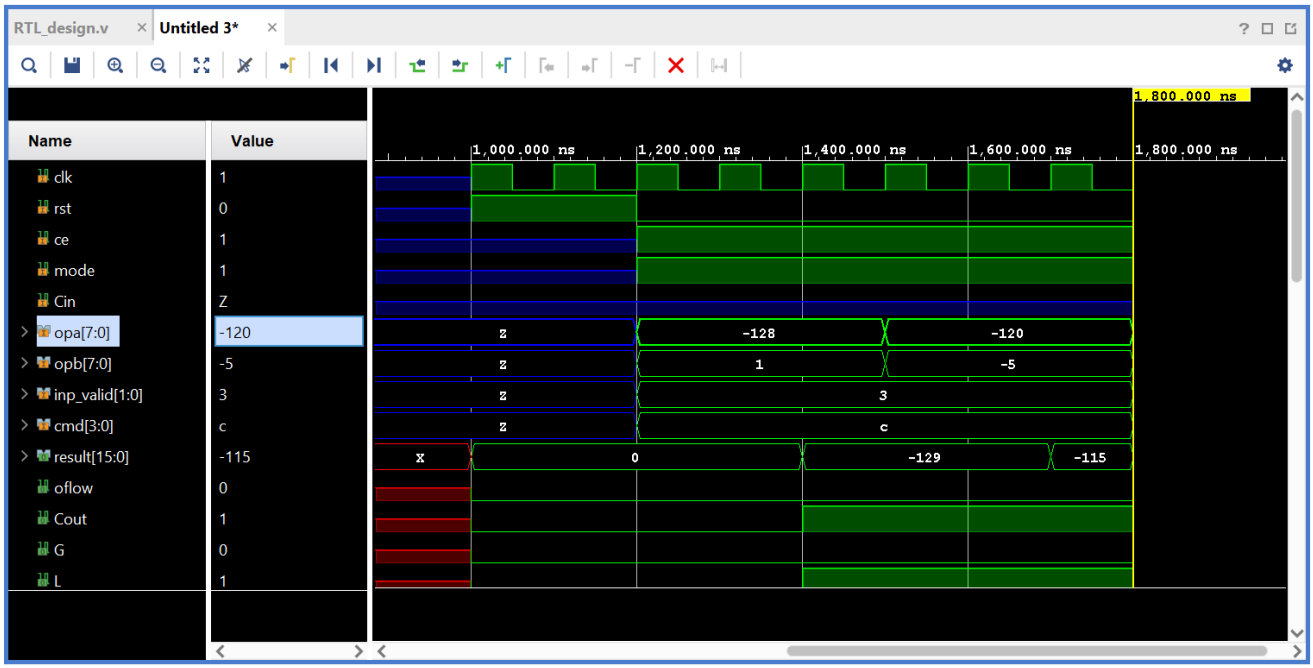

Figure 8. Signed subtraction

5.2 Code Coverage:

Questa Design Coverage

Scope: /tb_alu/inst_dut

Instance Path:
/tb_alu/inst_dut
Design Unit Name:
work.alu_design
Language:
Verilog
Source File:
tb_alu.v

Local Instance Coverage Details:

Total Coverage:					98.98%	**93.08%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	125	125	0	1	100.00%	**100.00%**
Branches	75	74	1	1	98.66%	**98.66%**
FEC Expressions	5	4	1	1	80.00%	**80.00%**
FEC Conditions	8	7	1	1	87.50%	**87.50%**
Toggles	278	276	2	1	99.28%	**99.28%**

6. CONCLUSION:

The implementation of the ALU and its verification through a structured testbench ensures that the design meets functional requirements. By handling both arithmetic and logical operations effectively, integrating status flag management, and employing a systematic verification approach, this ALU guarantees computational accuracy, reliability, and seamless integration within digital systems. The use of a stimulus-driven testbench, golden reference models, and comparison techniques with a scoreboard mechanism ensures robust validation and detection of potential errors, improving overall design efficiency.

7. FUTURE IMPROVEMENT:

- **High-Speed Optimization:** Improve architecture with pipeline execution to boost efficiency and reduce latency.
- **Expanded Functionality:** Add floating-point, complex number, and cryptographic operations for advanced computing.
- **Automated Verification:** Use AI-driven testbench generation and debugging tools to enhance accuracy and reduce development time.

By integrating these improvements, the ALU can evolve into a more powerful, efficient, and adaptive computing unit, catering to advanced digital applications while maintaining high accuracy and performance.