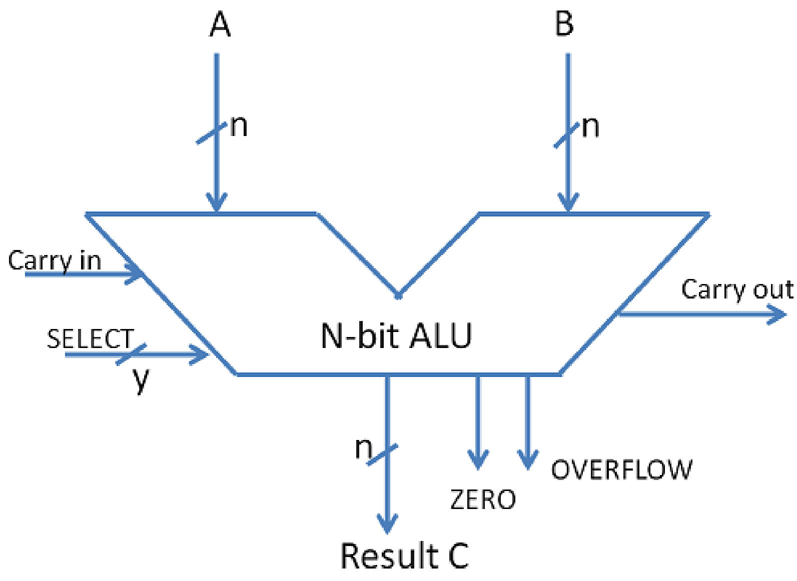
mirafra
TECHNOLOGIES

Project Report On

Design and Verification of an 8-Bit Parameterized Arithmetic Logic Unit (ALU) in Verilog



Submitted by:

Prithviraj Varma Konduru

Abstract:

This report presents a comprehensive overview of the design, implementation, and verification of an 8-bit Arithmetic Logic Unit (ALU) developed in Verilog.

The ALU supports a wide range of arithmetic and logical operations, including addition, subtraction (with and without carry), increment/decrement functions, comparisons, bitwise logic operations, shifts, rotates, and multiplication. Detailed explanations of the Verilog source code and the corresponding testbench are provided, elucidating the structure, control flow, and functional behaviour.

Simulation results confirm the correctness of the design, and future improvements are suggested to enhance performance and functionality.

Introduction:

An Arithmetic Logic Unit (ALU) is a critical component in digital systems, responsible for performing arithmetic and logical operations on input operands. In modern processors, the ALU forms the heart of the data path, executing instructions such as addition, subtraction, logical AND/OR/NOT, as well as comparison and shift/rotate operations. The ALU designed in this project is parameterized to operate on 8-bit data words, though the width can be modified via a parameter. It is implemented in Verilog, targeting synthesis on FPGA platforms, and includes a comprehensive verification environment to validate functional correctness across various scenarios.

The primary motivations for this design include:

- **Modularity**: Creating a reusable Verilog module that can be instantiated in larger system designs.
- **Functional Completeness**: Supporting a broad set of arithmetic and logical instructions that cover most typical ALU needs.
- **Flag Generation**: Producing status outputs such as overflow, carry-out, and comparator flags (greater, less, equal).
- **Testability**: Providing a robust testbench infrastructure that systematically applies stimulus vectors and checks outputs against expected results.

Objectives:

The specific objectives of the ALU project are as follows:

1. **Implement a Parameterized ALU**
   - Design a Verilog module (ALU) with a parameter w representing the operand width (default w = 8).
   - Allow for easy extension to wider data paths by adjusting the parameter.
2. **Support a Comprehensive Instruction Set**
   - **Arithmetic Operations**:
     - Unsigned addition, subtraction (with and without carry-in), increment/decrement of operands.
     - Multiplication of incremented or shifted operands.
     - Signed addition and subtraction with detection of signed overflow and comparator flags.
   - **Logical Operations**:
     - Bitwise AND, NAND, OR, NOR, XOR, XNOR.
     - Bitwise NOT for individual operands.
   - **Shift and Rotate Operations**:
     - Right and left shifts by one position.
     - Barrel-style rotate left/right by an amount specified in the lower bits of one operand.
3. **Generate Status Flags**
   - **Carry-Out (COUT)**: Indicates carry-out from the most significant bit in unsigned arithmetic.
   - **Overflow (OF)**: Detects signed overflow in signed arithmetic operations.

- **Comparator Flags (G, L, E)**: Indicate “Greater than”, “Less than”, and “Equal” results when comparing operands.
- **Error (ERR)**: Flags invalid input conditions (e.g., insufficient valid operands, illegal shift/rotate amounts).

4. **Provide Clear Input Interface**
    - Use control signals to indicate when operands are valid (IN_valid), specify operation modes (MODE), and select instructions via a 4-bit opcode (CMD).
    - Include clock enable (CE), synchronous reset (rst), and carry-in (Cin) for arithmetic operations.
5. **Develop a Robust Testbench**
    - Create a Verilog testbench that reads stimulus vectors from an external file (stimulus.txt), applies them to the DUT (Device Under Test), and compares DUT outputs to expected results.
    - Implement driver, monitor, and scoreboard tasks to automate the testing of all defined instructions.
    - Report pass/fail results for each test case and maintain a count of failed cases.
6. **Verify Functional Correctness**
    - Simulate waveforms to confirm accurate operation for each supported instruction.
    - Validate correct flag behaviour under edge conditions (e.g., overflow boundaries, shift/rotate error cases).

Architecture:

Serial no	Pin name	Direction	No of bits	Function
1	OPA	INPUT		Parameterized operand 1
2	OPB	INPUT		Parameterized operand 2
3	CIN	INPUT	1	This is the active high carry in input signal of 1-bit
4	CLK	INPUT	1	This is the clock signal to the design and it is edge sensitive
5	RST	INPUT	1	This is the active high asynchronous reset to the design
6	CE	INPUT	1	This is the active high clock enable signal 1-bit
7	MODE	INPUT	1	MODE signal 1 bit is high, then this is an Arithmetic Operation otherwise it is logical operation
8	INP_VALID	INPUT	2	Operands are valid as per below table 00 : No operand is valid 01: Operand A is valid 10: Operand B is valid

Serial no	Pin name	Direction	No of bits	Function
9	CMD	INPUT		Parametrizd (4 bit default) Arithematic Commands CMD = 0 :ADD , 1: SUB, 2: ADD_CIN, 3: SUB_CIN, 4 :INC_A , 5:DEC_A , 6: INC_B 7: DEC_B, 8: CMP, 9: oprand A and B both incr by 1 and then multiplication performed. 10: Operand A left shift by 1 and then multiply with B. 11: Operand A and Operand B can be signed and unsigned, perform addition of two numbers, based on last bit of result it will raise cout, overflow flag, based on result neg and zero will raise and greater than, less than or equal to will happen based on input operands . 12:Operand A and Operand B can be signed and unsigned, perform a substraction of two numbers, based on last bit of result it will raise cout, overflow flag, based on result neg and zero will raise and greater

Serial no	Pin name	Direction	No of bits	Function
	CMD	INPUT		Logical Commands CMD = 0: AND, 1: NAND, 2: OR, 3: NOR, 4: XOR, 5: XNOR, 6: NOT_A, 7: NOT_B, 8: SHR1_A, 9: SHL1_A, 10: SHR1_B, 11: SHL1_B,

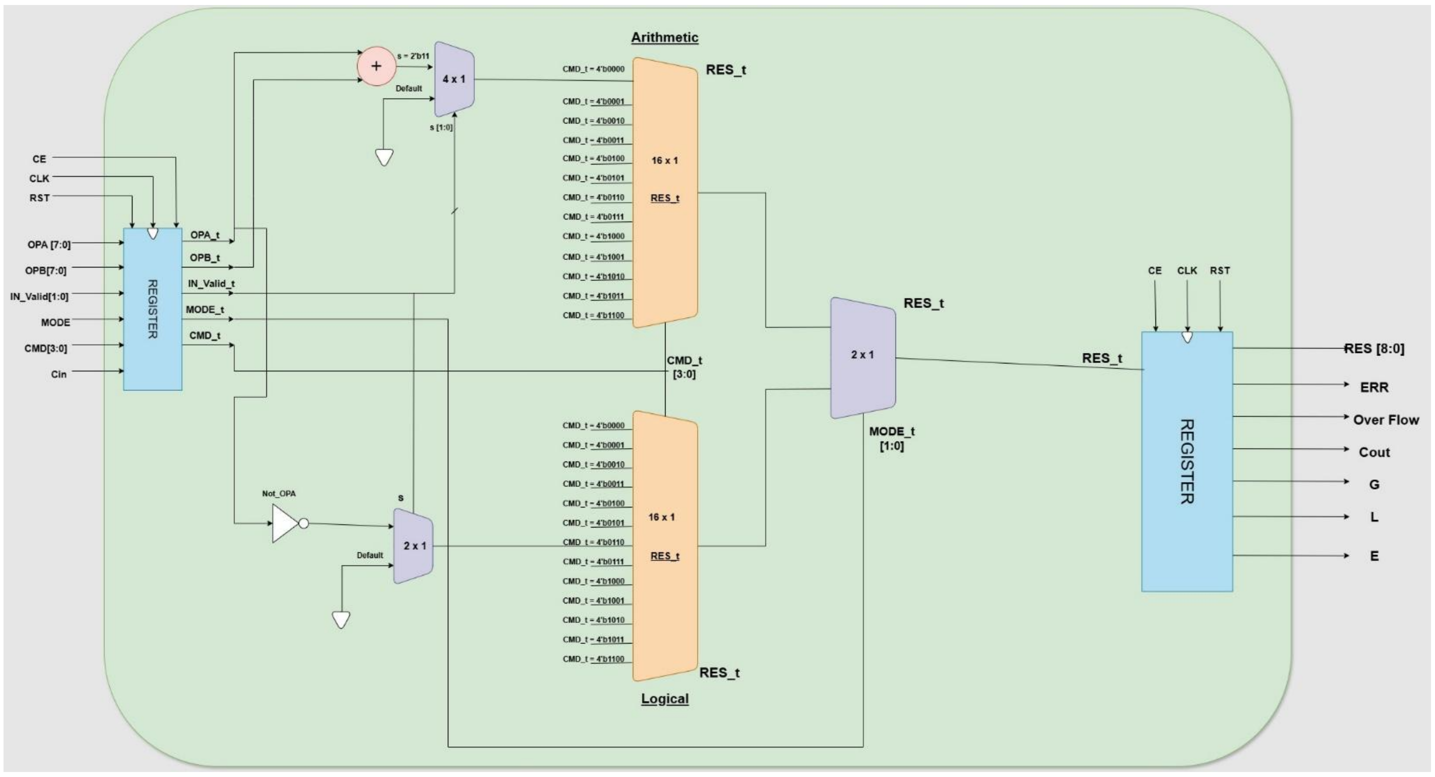
Seria l no	Pin name	Direction	No of bits	Function

9	CMD	INPUT		12: ROL_A_B
				if operandB 0000_X000 : output is operandA
				if operandB 0000_X001 : output is operandA left rotate by 1
				if operandB 0000_X010 : output is operandA left rotate by 2
				if operandB 0000_X011 : output is operandA left rotate by 3
				if operandB 0000_X100 : output is operandA left rotate by 4
				if operandB 0000_X101 : output is operandA left rotate by 5
				if operandB 0000_X110 : output is operandA left rotate by 6
				if operandB 0000_X111 : output is operandA left rotate by 7
				if operandB [7:4] any bit is 1 then its error whereas output will be as per [2:0 as mentioned above]

Seria l no	Pin name	Direction	No of bits	Function

9	CMD	INPUT		13:ROR_A_B.
				if operandB 0000_X000 : output is operandA
				if operandB 0000_X001 : output is operandA left rotate by 1
				if operandB 0000_X010 : output is operandA left rotate by 2
				if operandB 0000_X011 : output is operandA left rotate by 3
				if operandB 0000_X100 : output is operandA left rotate by 4
				if operandB 0000_X101 : output is operandA left rotate by 5
				if operandB 0000_X110 : output is operandA left rotate by 6
				if operandB 0000_X111 : output is operandA left rotate by 7
				if operandB [7:4] any bit is 1 then its error whereas output will be as per [2:0 as mentioned above]

Serial no	Pin name	Direction	No of bits	Function
10	RES	OUT		This is the total parameterized plus 1 bits result of the instruction performed by the ALU.
11	OFLOW	OUT	1	This 1-bit signal indicates an output overflow, during Addition/Subtraction
12	COUT	OUT	1	This is the carry out signal of 1-bit, during Addition/Subtraction
13	G	OUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is greater than the value of OPB
14	L	OUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is lesser than the value of OPB
15	E	OUT	1	This is the comparator output of 1-bit,which indicates that the value of OPA is equal to the value of OPB
16	ERR	OUT	1	When Cmd is selected as 12 or 13 and mode is logical operation , if 4th ,5th ,6th and 7th bit of OPB are 1, then ERR bit will be 1 else it is high impedance .



The ALU is designed as a combinational logic block placed between clocked input and output registers for seamless integration in synchronous systems. Inputs are latched on the rising clock edge, processed in a combinational way, and results are stored in output registers, introducing a one-cycle latency.

It supports both arithmetic and logical operations, selected via a mode signal, 4-bit command code and 2-bit INP_VALID code. The ALU takes up to two operands based on a 2-bit input validity signal, enabling both single-input and dual-input operations depending on its status. Flags like overflow, carry out, comparison results, and error detection are generated alongside outputs. Signed operations, including shifts, rotates, and multiply instructions, are also supported within this ALU module. For multiplication commands a further 1 more delay is introduces meaning a total of 2 clock cycle delay before the output is received.

Working:

The core "working" of the ALU can be understood as a two-stage process: first capturing and stabilizing inputs in registers, then feeding those registered values into a purely combinational logic block that decodes the opcode and produces the arithmetic or logical result along with the appropriate status flags.

Input Registration (Clocked Stage)

- On every rising edge of the clock (when CE = 1 and rst = 0), the external inputs—namely operand A (OPA), operand B (OPB), carry-in (Cin), the 2-bit "valid" indicator (IN_valid), the mode bit (MODE), and the 4-bit opcode (CMD)—are all sampled into internal registers (OPA_t, OPB_t, Cin_t, IN_valid_t, MODE_t, CMD_t).
- If the asynchronous reset (rst) is asserted, all of those registered signals are cleared to zero immediately, ensuring a well-defined starting point when the ALU is taken out of reset.
- This registration stage guarantees that changes on the external input pins do not glitch the combinational logic in mid-computation. Only after a clock edge (with CE = 1) do stable input values propagate forward.

Combinational Processing (Purely Combinational Stage)

Once the registered inputs (OPA_t, OPB_t, Cin_t, IN_valid_t, MODE_t, CMD_t) are stable, the ALU enters a purely combinational block (an always @(*) block in Verilog). Here is how it proceeds:

a. **Clear Defaults**

- At the very start of the combinational block, all intermediate outputs (RES_t, COUT_t, OF_t, G_t, L_t, E_t, ERR_t) are initialized to zero. This guarantees that, if no case is matched or if an error condition arises, those outputs remain at zero (except for the error flag).

b. **Mode Selection**

- The single bit MODE_t selects between two broad categories of operations:
    1. **Arithmetic Mode** (MODE_t = 1)
    2. **Logical/Shift/Rotate Mode** (MODE_t = 0)

c. **Opcode Decoding and Operation Execution**

- Within each mode, a case (CMD_t) statement decodes exactly which instruction to execute.
- Before performing most operations, the block checks that the right operand-validity bits in IN_valid_t are asserted. If that check fails, it immediately sets ERR_t = 1 and leaves all other outputs at zero. Otherwise, it proceeds with the calculation.

i. Arithmetic Mode (MODE_t = 1)

- **Unsigned Addition (CMD_t = 4'b0000)**
    - Condition: IN_valid_t == 2'b11 (both A and B valid).
    - Compute RES_t = OPA_t + OPB_t in (w+1)-bit precision so that bit w holds the carry-out.
    - Set COUT_t = RES_t[w] to capture that carry.
- **Unsigned Subtraction (CMD_t = 4'b0001)**
    - Condition: both operands valid.
    - Compute RES_t = OPA_t – OPB_t.
    - If OPA_t < OPB_t, there is effectively a borrow; set OF_t = 1 to indicate underflow.
- **Addition with Carry-in (CMD_t = 4'b0010)**
    - Condition: both valid.

    - Compute RES_t = OPA_t + OPB_t + Cin_t; set COUT_t = RES_t[w].
- **Subtraction with Borrow (CMD_t = 4'b0011)**
    - Condition: both valid.
    - Compute RES_t = OPA_t – OPB_t – Cin_t.
    - Set OF_t = 1 if OPA_t < (OPB_t + Cin_t), indicating signed underflow or borrow.
- **Increment A (CMD_t = 4'b0100)**
    - Condition: IN_valid_t[0] == 1 (only operand A must be valid).
    - Compute RES_t = OPA_t + 1.
    - If bit w of the result is 1, set COUT_t = 1 (overflow out of the MSB).
- **Decrement A (CMD_t = 4'b0101)**
    - Condition: IN_valid_t[0] == 1.
    - Compute RES_t = OPA_t – 1.
    - If the MSB of RES_t is 1, that implies signed underflow; set OF_t = 1.
- **Increment B (CMD_t = 4'b0110)**, **Decrement B (CMD_t = 4'b0111)**
    - Analogous to INC_A/DEC_A but operate on OPB_t.
    - INC_B sets COUT_t if (OPB_t + 1) overflows; DEC_B sets OF_t if (OPB_t – 1) underflows.
- **Compare (CMD_t = 4'b1000)**
    - Condition: both valid.
    - Instead of producing an arithmetic result, the ALU sets the comparator flags:

        - E_t = 1 if OPA_t == OPB_t
        - L_t = 1 if OPA_t < OPB_t
        - G_t = 1 if OPA_t > OPB_t
    - RES_t remains zero in this case.
- **Unsigned Multiply of (A+1)*(B+1) (CMD_t = 4'b1001)**
    - Condition: both valid.
    - Compute RES_t = (OPA_t + 1) * (OPB_t + 1). Since each operand is now effectively (w+1) bits, the product can be up to (2*w + 2) bits; the code allocates 2*w+1 bits for RES_t (dropping the very top bit if any).
    - No flags COUT_t or OF_t are set here; any overflow beyond the allocated width would simply be truncated.
    - Because multiplication tends to be a longer combinational path, the design writes this result first into a temporary register RES_m in one clock, and then into the output register RES in the following clock—creating a two-cycle latency.
- **Unsigned Multiply of (A<<1)B (CMD_t = 4'b1010)**
    - Condition: both valid.
    - Internally shift OPA_t left by one bit, multiply by OPB_t, and store in RES_t.
    - This is also pipelined through RES_m before reaching the final output so that the wide multiplier has time to settle.
- **Signed Addition (CMD_t = 4'b1011)**
    - Condition: both valid.
    - Cast OPA_t and OPB_t to signed values sOPA and sOPB.
    - Compute sRES = sOPA + sOPB; assign RES_t = sRES.

    - Detect signed overflow by checking if sOPA and sOPB had the same sign but sRES has a different sign—in that case, set OF_t = 1.
    - Also set comparator flags E_t, L_t, G_t based on the signed comparison of sOPA and sOPB.
- **Signed Subtraction (CMD_t = 4'b1100)**
    - Condition: both valid.
    - Compute sRES = sOPA – sOPB.
    - Assign RES_t = sRES.
    - Detect signed overflow if sOPA and sOPB have opposite signs and the result sign differs from sOPA; set OF_t accordingly.
    - Set comparator flags E_t, L_t, G_t for signed comparison.

ii. Logical / Shift / Rotate Mode (MODE_t = 0)

- **Bitwise AND (CMD_t = 4'b0000)**
    - Condition: both valid.
    - Compute RES_t = OPA_t & OPB_t.
- **Bitwise NAND (CMD_t = 4'b0001), NOR (4'b0011), XNOR (4'b0101)**
    - Condition: both valid.
    - Compute the appropriate bitwise complement of AND/OR/XOR.
    - Explicitly zero out the upper half of RES_t (bits [2*w-1 : w]) since only the lower w bits contain the meaningful result.
- **Bitwise OR (4'b0010), XOR (4'b0100)**

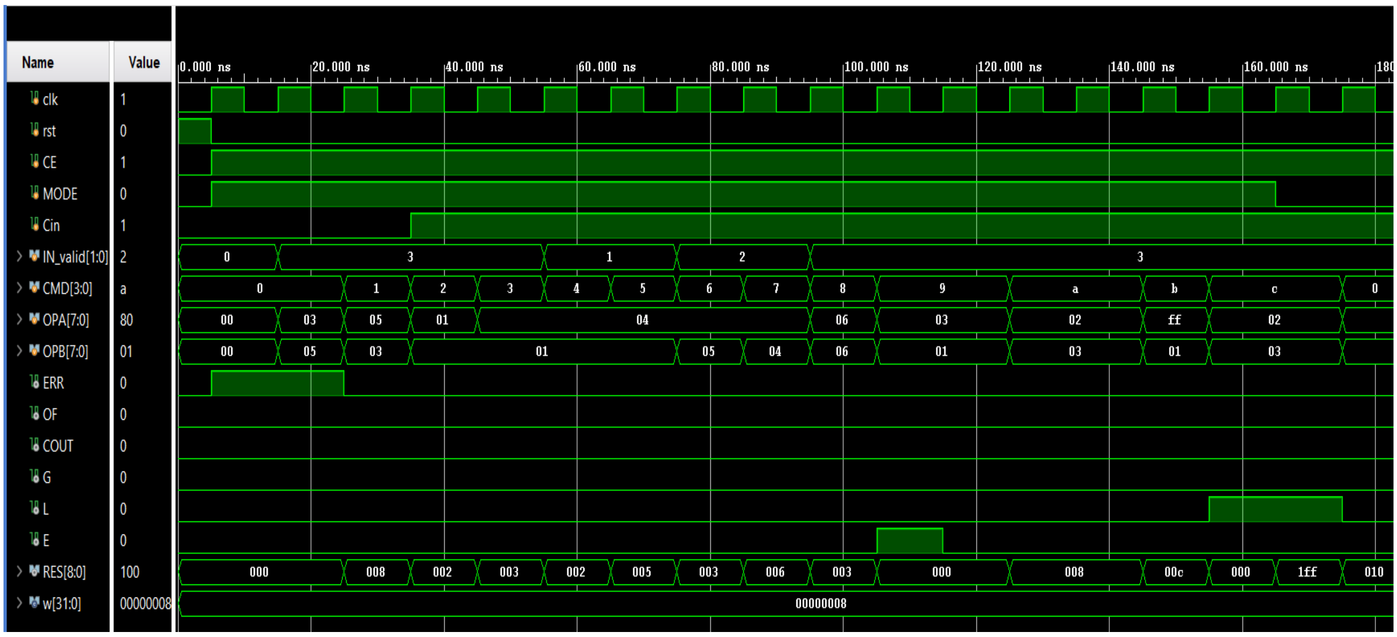
    - Condition: both valid.
    - Compute OPA_t | OPB_t or OPA_t ^ OPB_t, respectively.
    - Upper bits remain zero.
- **NOT_A (4'b0110)**, **NOT_B (4'b0111)**
    - Condition: only the relevant operand's valid bit must be 1 (IN_valid_t[0] for A, IN_valid_t[1] for B).
    - Compute bitwise complement of the single operand, zeroing the upper bits.
- **Shift Right A (CMD_t = 4'b1000)**, **Shift Left A (4'b1001)**
    - Condition: IN_valid_t[0] == 1.
    - Perform a one-bit shift of OPA_t.
- **Shift Right B (4'b1010)**, **Shift Left B (4'b1011)**
    - Condition: IN_valid_t[1] == 1.
    - Perform a one-bit shift of OPB_t.
- **Rotate Left A by Amount in B (4'b1100)**
    - Condition: both valid.
    - First check if higher bits of OPB_t (bits [w-1 : $clog2(w)]) are zero—if not, set ERR_t = 1 (invalid rotation amount).
    - Otherwise, let shift_amount = OPB_t[$clog2(w)-1 : 0] (for 8-bit, that's OPB_t[2:0]).
    - Compute a standard barrel-rotate: (OPA_t << shift_amount) | (OPA_t >> (w – shift_amount)), zero-extending so that the final RES_t is (2*w + 1) bits with a leading zero.
- **Rotate Right A by Amount in B (4'b1101)**
    - Same validity check on OPB_t as above.
    - Compute (OPA_t >> shift_amount) | (OPA_t << (w – shift_amount)), again zero-extended.

- **Default Case**
    - If CMD_t does not match any defined code in the selected mode, set ERR_t = 1 and leave RES_t and all flags zero.

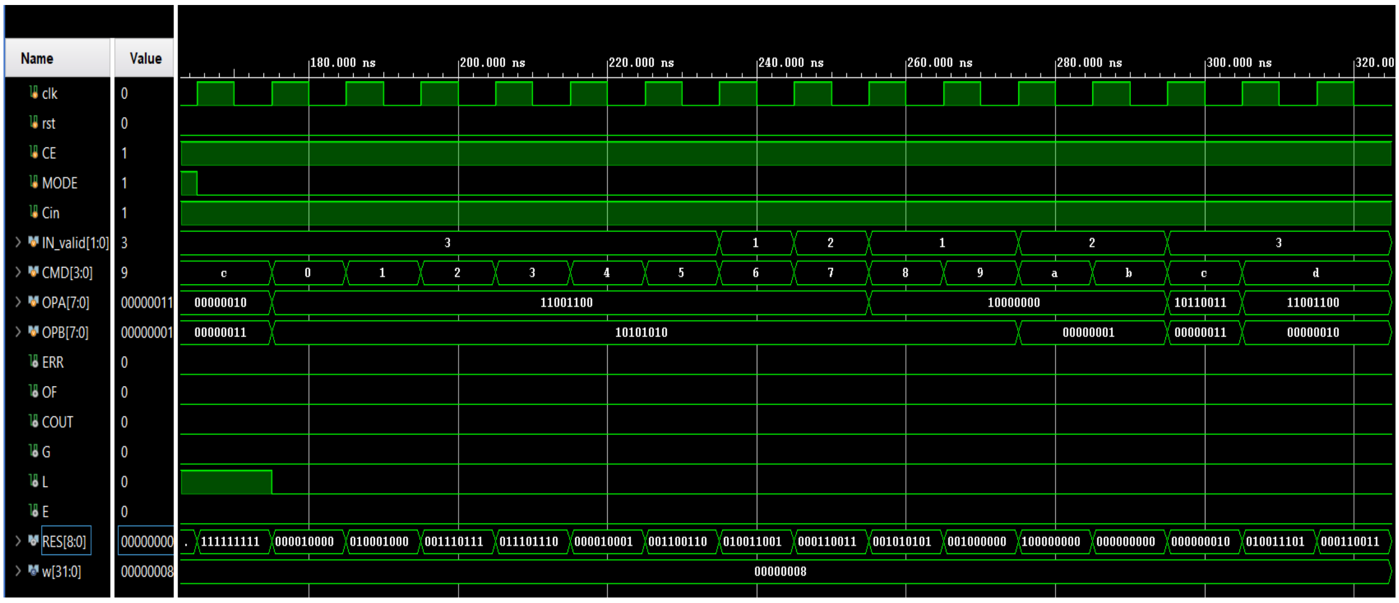
Output Registration (Clocked Stage)

- After the combinational block calculates intermediate results (RES_t, COUT_t, OF_t, G_t, L_t, E_t, ERR_t), a second always @(posedge clk or posedge rst) block registers them onto the module's outputs (RES, COUT, OF, G, L, E, ERR).
- **Reset Behaviour**: If rst = 1, all outputs and the pipeline register RES_m are cleared to zero immediately.
- **Clock Enable (CE)**: When CE = 0, outputs hold their previous values (no register updates occur). When CE = 1, outputs are loaded as follows:
    - **For most opcodes**: RES <= RES_t (one-cycle result), and flags COUT, OF, G, L, E, ERR are updated to their combinational equivalents (..._t).
    - **For multiplication opcodes (1001, 1010 in arithmetic mode)**:
        1. On the first cycle, the combinational product appears in RES_t; the register RES_m is loaded with that value.
        2. On the second cycle, RES <= RES_m. Meanwhile, flags (COUT, OF, G, L, E, ERR) can be updated immediately from the combinational block. This two-cycle handshake ensures the wide multiplier completes its operation before the result is exposed on RES.

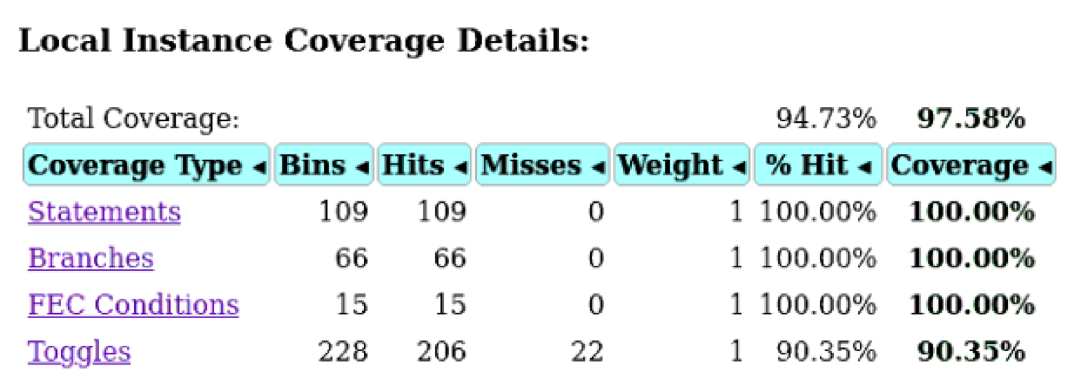
Result:



Output of Arithmetic Operations



Output of Logical Operations



Local Instance Coverage Details:

Total Coverage: 94.73% **97.58%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	109	109	0	1	100.00%	**100.00%**
Branches	66	66	0	1	100.00%	**100.00%**
FEC Conditions	15	15	0	1	100.00%	**100.00%**
Toggles	228	206	22	1	90.35%	**90.35%**

Design Coverage Report

Conclusion:

The Verilog ALU designed in this project successfully meets all functional requirements:

- **Versatility**: Supports 14 distinct opcodes in arithmetic mode and 14 in logical mode, covering a comprehensive range of operations.
- **Scalability**: Parameterized data width allows easy extension beyond 8 bits.
- **Correctness**: Testbench-driven verification, using 74 stimulus vectors, validated correct behaviour across all instructions, including edge cases and error conditions.
- **Flag Generation**: Accurate generation of carry-out, overflow, and comparator flags, essential for integration in larger processor designs.
- **Modularity**: Clear separation of input capture, combinational execution, and output registering simplifies maintenance and reuse.

Future Improvement:

- **Deepen Pipelining**: Split decode, execution, and flag generation into separate stages to boost clock speed.
- **Add New Ops**: Incorporate divide, modulus, and signed multiply.
- **Wider Data Width**: Parameterize for 16- or 32-bit with DSP-slice support.
- **Formal Verification**: Integrate SystemVerilog Assertions to prove correctness of arithmetic, logic, and error checks.