ALU Documentation

Dua Kaurejo

Overview

This arithmetic logic unit (ALU) is implemented in Verilog and performs a wide range of signed and unsigned arithmetic, logical, comparison, and bitwise shift operations. The ALU supports both 8-bit inputs (default configuration) and customizable input widths. The design includes overflow detection, carry flag generation, and flag updates to enable more sophisticated arithmetic and logical computations.

The ALU functionality is verified through a comprehensive testbench to ensure correctness for edge cases, signed and unsigned operations, and specific overflow scenarios.

Input Parameters

- **A**: NUM_BITS-bit input operand (interpreted as signed or unsigned based on the mode).
- **B**: NUM_BITS-bit input operand (interpreted as signed or unsigned based on the mode).
- **Opcode**: 4-bit code defining the operation to perform (e.g., addition, subtraction, etc.).
- **Signed Mode**: Determines whether the inputs are treated as signed or unsigned integers.
- **NUM_BITS**: the number of bits in inputs A & B, defaults to 8-bits

Output Parameters

- **Result:** *NUM_BITS*-bit result of the operation performed on A and B.
- **Flags:**
    - **Z (Zero)**: Set to 1 if the result is 0.
    - **N (Negative)**: Set to 1 if the result is negative in signed mode.
    - **C (Carry)**: Set to 1 if there is a carry/borrow out for unsigned **operations.**
    - **V (Overflow)**: Set to 1 if there is signed overflow in arithmetic operations or division by zero.

ALU Code Explanation

Arithmetic Operations

1. **Addition** (Opcode: 0000):

- If signed_mode is enabled, signed addition is performed using a temporary signed variable. Overflow is detected if the sign of the result differs from the expected sign based on the operands.
- If unsigned, the carry-out is computed and stored in the C flag.

2. **Subtraction** (Opcode: 0001):
   - For signed mode, subtraction uses a temporary signed variable, and overflow is detected based on operand and result sign mismatch.
   - For unsigned mode, the carry-out (borrow) is computed and stored in the C flag.
3. **Multiplication** (Opcode: 0010):
   - Signed multiplication is supported using a signed 16-bit intermediate result (s_mult_result). The overflow is detected if the upper bits differ from the sign extension of the lower bits.
   - For unsigned multiplication, carry-out is set if upper bits are non-zero.
4. **Division** (Opcode: 0011):
   - Handles division by zero by setting the overflow flag.
   - Signed division checks for special cases like dividing the minimum signed value by -1, which causes overflow.

Logical and Bitwise Operations

1. **AND** (Opcode: 0100): Performs bitwise AND on A and B.
2. **OR** (Opcode: 0101): Performs bitwise OR on A and B.
3. **XOR** (Opcode: 0110): Performs bitwise XOR on A and B.
4. **NOR** (Opcode: 0111): Performs bitwise NOR on A and B.
5. **NOT** (Opcode: 1000): Performs bitwise NOT on A.

Comparison Operations

1. **Greater Than** (Opcode: 1001):
   - Checks if A > B. For signed mode, signed comparison is performed.
2. **Less Than** (Opcode: 1010):
   - Checks if A < B. For signed mode, signed comparison is performed.
3. **Equality** (Opcode: 1011):
   - Checks if A == B.

ALU Documentation

Dua Kaurejo

Shift Operations

1. **Logical Left Shift** (Opcode: 1100):
    - Shifts A left by 1 bit and stores the MSB in the carry flag.
2. **Logical Right Shift** (Opcode: 1101):
    - Shifts A right by 1 bit, filling the MSB with 0, and stores the LSB in the carry flag.
3. **Arithmetic Right Shift** (Opcode: 1110):
    - Performs a right shift while preserving the sign bit in signed mode.

Table 1: List of ALU Operations and Relevant Flags

Operation	Opcode	Description	Affected Flags
Addition	`0000`	Adds A and B	Z, C, V, N
Subtraction	`0001`	Subtracts B from A	Z, C, V, N
Multiplication	`0010`	Multiplies A and B	Z, V, N, C (unsigned)
Division	`0011`	Divides A by B	Z, V
AND	`0100`	Logical AND between A and B	Z
OR	`0101`	Logical OR between A and B	Z
XOR	`0110`	Logical XOR between A and B	Z
NOR	`0111`	Logical NOR between A and B	Z
NOT	`1000`	Logical NOT of A	Z
Greater Than	`1001`	Checks if A is greater than B	Z
Less Than	`1010`	Checks if A is less than B	Z
Equality	`1011`	Checks if A is equal to B	Z
Logical Left Shift	`1100`	Shifts A left by 1	C
Logical Right Shift	`1101`	Shifts A right by 1	C
Arithmetic Right Shift	`1110`	Shifts A right by 1 (preserves MSB if signed)	C

ALU Documentation

Dua Kaurejo

Testbench Code Explanation

Overview

The testbench (ALU_tb.v) is designed to rigorously verify the ALU by simulating a variety of edge cases and standard operations. It uses parameterized input widths to enable flexibility for different configurations of the ALU.

Key Components

1. **Inputs**:
    - Raw values for A and B.
    - signed_mode to toggle between signed and unsigned operations.
    - opcode to specify the operation.
2. **Outputs**:
    - Result: The result of the operation.
    - Flags: Z, N, C, V.

Test Coverage

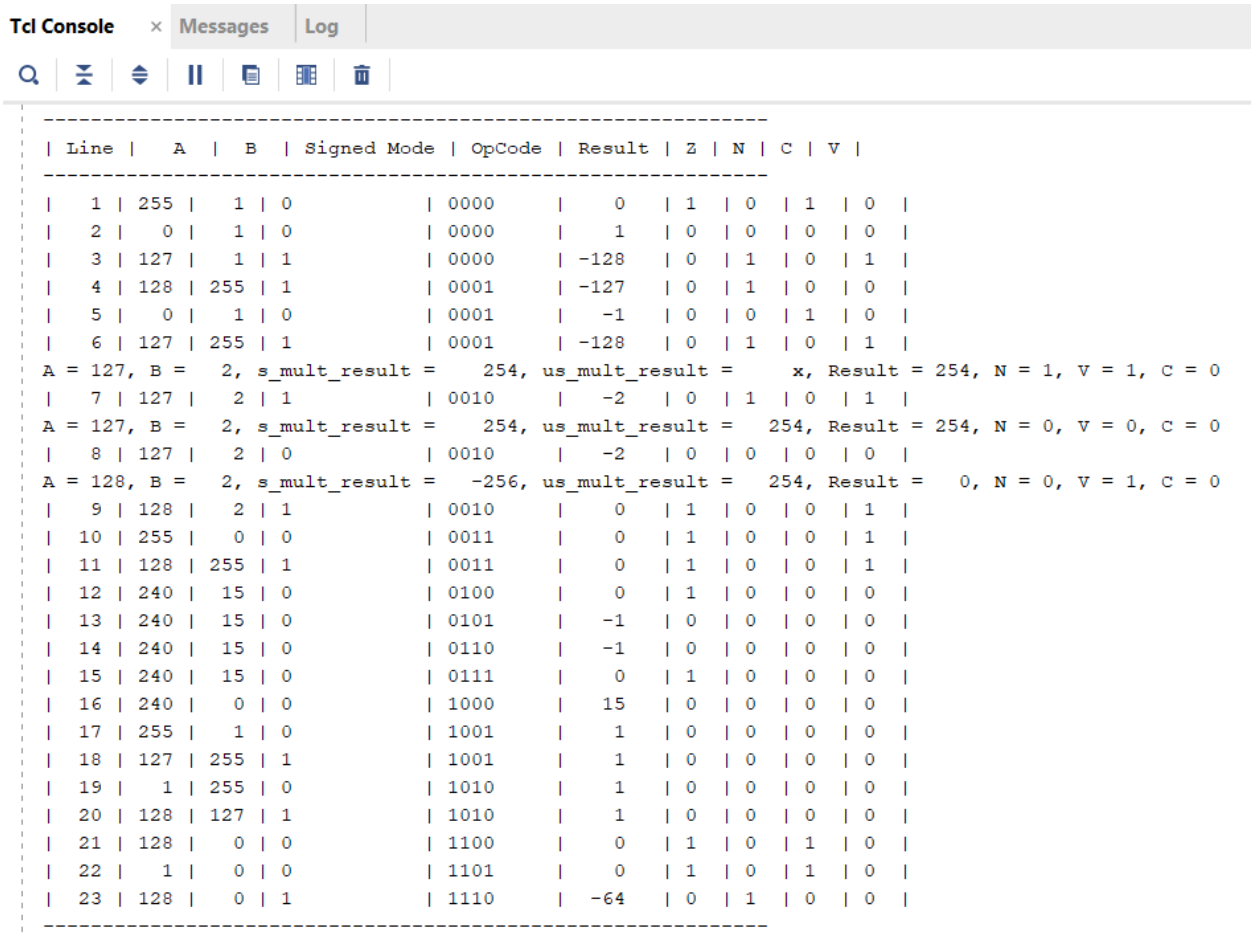
- **Arithmetic Tests**:
    - Includes edge cases such as addition overflow, subtraction borrow, and multiplication overflow.
- **Division Tests**:
    - Verifies proper handling of division by zero and signed division overflow.
- **Logical Tests**:
    - Covers basic AND, OR, XOR, NOR, and NOT operations.
- **Comparison Tests**:
    - Validates signed and unsigned greater-than, less-than, and equality comparisons.
- **Shift Tests**:
    - Ensures proper functionality of logical and arithmetic shifts with flag updates.

Notes

- The ALU currently limits multiplication and division to the lower NUM_BITS bits of the result. Overflow in these cases is indicated through the V flag.
- The signed interpretation of inputs is only active when signed_mode is set. Otherwise, all operations treat inputs as unsigned values.
- The testbench ensures coverage for common edge cases and thoroughly verifies flag behaviors.

ALU Documentation

Dua Kaurejo



Tcl Console × Messages Log

```
------------------------------------------------------------
| Line |   A  |  B  | Signed Mode | OpCode | Result | Z | N | C | V |
------------------------------------------------------------
|   1 | 255 |   1 | 0           | 0000   |    0   | 1 | 0 | 1 | 0 |
|   2 |   0 |   1 | 0           | 0000   |    1   | 0 | 0 | 0 | 0 |
|   3 | 127 |   1 | 1           | 0000   | -128   | 0 | 1 | 0 | 1 |
|   4 | 128 | 255 | 1           | 0001   | -127   | 0 | 1 | 0 | 0 |
|   5 |   0 |   1 | 0           | 0001   |   -1   | 0 | 0 | 1 | 0 |
|   6 | 127 | 255 | 1           | 0001   | -128   | 0 | 1 | 0 | 1 |
A = 127, B =   2, s_mult_result =    254, us_mult_result =     x, Result = 254, N = 1, V = 1, C = 0
|   7 | 127 |   2 | 1           | 0010   |   -2   | 0 | 1 | 0 | 1 |
A = 127, B =   2, s_mult_result =    254, us_mult_result =   254, Result = 254, N = 0, V = 0, C = 0
|   8 | 127 |   2 | 0           | 0010   |   -2   | 0 | 0 | 0 | 0 |
A = 128, B =   2, s_mult_result =   -256, us_mult_result =   254, Result =   0, N = 0, V = 1, C = 0
|   9 | 128 |   2 | 1           | 0010   |    0   | 1 | 0 | 0 | 1 |
|  10 | 255 |   0 | 0           | 0011   |    0   | 1 | 0 | 0 | 1 |
|  11 | 128 | 255 | 1           | 0011   |    0   | 1 | 0 | 0 | 1 |
|  12 | 240 |  15 | 0           | 0100   |    0   | 1 | 0 | 0 | 0 |
|  13 | 240 |  15 | 0           | 0101   |   -1   | 0 | 0 | 0 | 0 |
|  14 | 240 |  15 | 0           | 0110   |   -1   | 0 | 0 | 0 | 0 |
|  15 | 240 |  15 | 0           | 0111   |    0   | 1 | 0 | 0 | 0 |
|  16 | 240 |   0 | 0           | 1000   |   15   | 0 | 0 | 0 | 0 |
|  17 | 255 |   1 | 0           | 1001   |    1   | 0 | 0 | 0 | 0 |
|  18 | 127 | 255 | 1           | 1001   |    1   | 0 | 0 | 0 | 0 |
|  19 |   1 | 255 | 0           | 1010   |    1   | 0 | 0 | 0 | 0 |
|  20 | 128 | 127 | 1           | 1010   |    1   | 0 | 0 | 0 | 0 |
|  21 | 128 |   0 | 0           | 1100   |    0   | 1 | 0 | 1 | 0 |
|  22 |   1 |   0 | 0           | 1101   |    0   | 1 | 0 | 1 | 0 |
|  23 | 128 |   0 | 1           | 1110   |  -64   | 0 | 1 | 0 | 0 |
------------------------------------------------------------
```

Figure 1 Behavioral simulation output for ALU_tb on tcl console.

ALU Documentation

Dua Kaurejo

Table 2: Summary of Test Cases Used in the Testbench

Line	Raw A	Raw B	Signed Mode	Opcode	Actual A	Actual B	Expected Result	Z	N	C	V
1	255	1	0	`0000`	255	1	0 (Overflow)	1	0	1	0
2	0	1	0	`0000`	0	1	1	0	0	0	0
3	127	1	1	`0000`	127	1	-128 (Overflow)	0	1	0	1
4	128	255	1	`0001`	-128	-1	-127	0	1	0	0
5	0	1	0	`0001`	0	1	255	0	0	1	0
6	127	255	1	`0001`	127	-1	-128 (Overflow)	0	1	0	1
7	127	2	1	`0010`	127	2	254	0	1	0	1
8	127	2	0	`0010`	127	2	254	0	0	0	0
9	128	2	1	`0010`	-128	2	0 (Overflow)	1	0	0	1
10	255	0	0	`0011`	255	0	0 (Division by Zero)	1	0	0	1
11	128	255	1	`0011`	-128	-1	0 (Overflow)	1	0	0	1
12	240	15	0	`0100`	240	15	0	1	0	0	0
13	240	15	0	`0101`	240	15	255	0	0	0	0
14	240	15	0	`0110`	240	15	255	0	0	0	0
15	240	15	0	`0111`	240	15	0	1	0	0	0
16	240	0	0	`1000`	240	0	15	0	0	0	0
17	255	1	0	`1001`	255	1	1	0	0	0	0
18	127	255	1	`1001`	127	-1	0	1	0	0	0
19	1	255	0	`1010`	1	255	1	0	0	0	0
20	128	127	1	`1010`	-128	127	1	0	0	0	0
21	128	0	0	`1100`	128	0	0 (Overflow)	1	0	1	0
22	1	0	0	`1101`	1	0	0 (Overflow)	1	0	1	0
23	128	0	1	`1110`	-128	0	-64	0	1	0	0

ALU Documentation

Dua Kaurejo

Code Screenshots

ALU.v

```
module ALU #(parameter NUM_BITS = 8) (
    input [NUM_BITS-1:0] A,
    input [NUM_BITS-1:0] B,
    input signed_mode,
    input [3:0] opcode,
    output reg [NUM_BITS-1:0] Result,
    output reg Z, N, C, V
);
    reg signed [2*NUM_BITS-1:0] s_mult_result;
    reg [2*NUM_BITS-1:0] us_mult_result;
    reg signed [NUM_BITS:0] temp_result;
    reg signed [NUM_BITS-1:0] signed_A, signed_B;

    always @(*) begin
        Z = 0; N = 0; C = 0; V = 0;
        Result = {NUM_BITS{1'b0}};

        signed_A = $signed(A);
        signed_B = $signed(B);

        case (opcode)
            // Addition
            4'b0000: begin
                if (signed_mode) begin
                    temp_result = signed_A + signed_B;
                    Result = temp_result[NUM_BITS-1:0];
                    V = ((signed_A[NUM_BITS-1] == signed_B[NUM_BITS-1])
                        && (signed_A[NUM_BITS-1] != Result[NUM_BITS-1]));
                end else begin
                    {C, Result} = A + B;
                end
            end

            // Subtraction
            4'b0001: begin
                if (signed_mode) begin
                    temp_result = signed_A - signed_B;
                    Result = temp_result[NUM_BITS-1:0];
                    V = (signed_A[NUM_BITS-1] != signed_B[NUM_BITS-1])
                        && (signed_A[NUM_BITS-1] != Result[NUM_BITS-1]);
                end else begin
                    {C, Result} = A - B;
                end
```

ALU Documentation

Dua Kaurejo

```
43                  end
44              end
45
46              // Multiplication
47              4'b0010: begin
48                  if (signed_mode) begin
49                      s_mult_result = signed_A * signed_B;
50                      Result = s_mult_result[NUM_BITS-1:0];
51                      V = (s_mult_result[2*NUM_BITS-1:NUM_BITS] != {NUM_BITS{s_mult_result[NUM_BITS-1]}});
52                      N = Result[NUM_BITS-1];
53                      $display("A = %d, B = %d, s_mult_result = %d, us_mult_result = %d, Result = %d, N = %b, V = %b, C = %b",
54                      A, B, s_mult_result, us_mult_result, Result, N, V, C);
55
56                  end else begin
57                      us_mult_result = A * B;
58                      Result = us_mult_result[NUM_BITS-1:0];
59                      C = |us_mult_result[2*NUM_BITS-1:NUM_BITS];
60                      N = 0;
61                      V = 0;
62                      $display("A = %d, B = %d, s_mult_result = %d, us_mult_result = %d, Result = %d, N = %b, V = %b, C = %b",
63                       A, B, s_mult_result, us_mult_result, Result, N, V, C);
64
65                  end
66              end
67
68
69              // Division
70              4'b0011: begin
71                  if (B == 0) begin
72                      V = 1; // Division by zero error
73                      Result = {NUM_BITS{1'b0}};
74                  end else if (signed_mode) begin
75                      if ((signed_A == -(1 << (NUM_BITS - 1))) && (signed_B == -1)) begin
76                          V = 1; // Overflow on signed division
77                          Result = {NUM_BITS{1'b0}};
78                      end else begin
79                          Result = signed_A / signed_B;
80                      end
81                  end else begin
82                      Result = A / B;
83                  end
```

ALU Documentation

Dua Kaurejo

```
            end


            // Logical Operations
            4'b0100: Result = A & B; // AND
            4'b0101: Result = A | B; // OR
            4'b0110: Result = A ^ B; // XOR
            4'b0111: Result = ~(A | B); // NOR
            4'b1000: Result = ~A; // NOT on A

            // Comparisons
            4'b1001: Result = (signed_mode ? (signed_A > signed_B) : (A > B)) ? 1 : 0; // Greater Than
            4'b1010: Result = (signed_mode ? (signed_A < signed_B) : (A < B)) ? 1 : 0; // Less Than
            4'b1011: Result = (A == B) ? 1 : 0; // Equality

            // Shifts
            4'b1100: begin // Logical Left Shift by 1
                Result = A << 1;
                C = A[NUM_BITS-1];
            end
            4'b1101: begin // Logical Right Shift by 1
                Result = A >> 1;
                C = A[0];
            end
            4'b1110: begin // Arithmetic Right Shift by 1 (Signed)
                Result = signed_mode ?
                        {A[NUM_BITS-1], A[NUM_BITS-1:1]} :  // Preserve sign bit (MSB) for signed mode
                        {1'b0, A[NUM_BITS-1:1]};           // Fill MSB with 0 for unsigned mode
                //(signed_A >>> 1) : (A >> 1);
                C = A[0];
            end

            default: Result = {NUM_BITS{1'b0}};
        endcase

        // Set flags based on Result
        Z = (Result == 0);
        N = signed_mode ? Result[NUM_BITS-1]:0; // Set Negative flag only

    end
endmodule
```

ALU_tb.v

ALU Documentation

Dua Kaurejo

```
`timescale 1ns/1ps

module ALU_tb;

    // Parameters
    parameter NUM_BITS = 8;

    // Testbench Signals
    reg [NUM_BITS-1:0] A, B;
    reg signed_mode;
    reg [3:0] opcode;
    wire [NUM_BITS-1:0] Result;
    wire Z, N, C, V;

    // Instantiate the ALU
    ALU #(NUM_BITS) alu_inst (
        .A(A),
        .B(B),
        .signed_mode(signed_mode),
        .opcode(opcode),
        .Result(Result),
        .Z(Z),
        .N(N),
        .C(C),
        .V(V)
    );

    // Task for running a single test case with signed/unsigned support
   integer test_counter=0;
   task run_test(
    input [NUM_BITS-1:0] a,
    input [NUM_BITS-1:0] b,
    input s_mode,
    input [3:0] op,
    input signed [NUM_BITS-1:0] expected_result,
    input expected_z,
    input expected_n,
    input expected_c,
    input expected_v
);
    reg signed [NUM_BITS-1:0] actual_result;
```

ALU Documentation

Dua Kaurejo

```
42      begin
43          test_counter = test_counter+1;
44          A = a;
45          B = b;
46          signed_mode = s_mode;
47          opcode = op;
48          #10; // Wait for the result to settle
49
50          if (signed_mode) begin
51              actual_result = $signed(Result); // Interpret Result as signed
52          end else begin
53              actual_result = Result; // Treat Result as unsigned
54          end
55
56          // Verification and display
57          $display("| %3d | %3d | %3d | %1b           | %04b     | %4d   | %1b  | %1b  | %1b  | %1b  |",
58                   test_counter, A, B, signed_mode, opcode, actual_result, Z, N, C, V);
59
60          if (actual_result !== expected_result) begin
61              $fatal("Error: Result mismatch. Expected %d, got %d", expected_result, actual_result);
62          end
63          if (Z !== expected_z) begin
64              $fatal("Error: Z mismatch. Expected %b, got %b", expected_z, Z);
65          end
66          if (N !== expected_n) begin
67              $fatal("Error: N mismatch. Expected %b, got %b", expected_n, N);
68          end
69          if (C !== expected_c) begin
70              $fatal("Error: C mismatch. Expected %b, got %b", expected_c, C);
71          end
72          if (V !== expected_v) begin
73              $fatal("Error: V mismatch. Expected %b, got %b", expected_v, V);
74          end
75      end
76  endtask
77
78
79      initial begin
80          // Print the header for the truth table
81          $display("-------------------------------------------------------------");
82          $display("| Line |   A  |  B  | Signed Mode | OpCode | Result | Z | N | C | V |");
```

```
83          $display("------------------------------------------------------------");
84
85          // Edge Cases for Addition
86          run_test(255, 1, 0, 4'b0000, 0, 1, 0, 1, 0);     // Unsigned addition
87          run_test(0, 1, 0, 4'b0000, 1, 0, 0, 0, 0);       // Unsigned addition
88          run_test(127, 1, 1, 4'b0000, -128, 0, 1, 0, 1); // Signed addition
89
90          // Edge Cases for Subtraction
91          run_test(128, 255, 1, 4'b0001, -127, 0, 1, 0, 0); // Signed subtraction
92          run_test(0, 1, 0, 4'b0001, 255, 0, 0, 1, 0);      // Unsigned subtraction
93          run_test(127, 255, 1, 4'b0001, -128, 0, 1, 0, 1); // Signed subtraction
94          // passed above
95
96          // Edge Cases for Multiplication
97          run_test(127, 2, 1, 4'b0010, 254, 0, 1, 0, 1);    // Signed multiplication
98          run_test(127, 2, 0, 4'b0010, 254, 0, 0, 0, 0);    // Unsigned multiplication
99          run_test(128, 2, 1, 4'b0010, 0, 1, 0, 0, 1);   // Signed multiplication
100
101         // Edge Cases for Division
102         run_test(255, 0, 0, 4'b0011, 0, 1, 0, 0, 1);      // Unsigned division by zero
103         run_test(128, 255, 1, 4'b0011, 0, 1, 0, 0, 1); // Signed division
104
105         // Logical Operations
106         run_test(240, 15, 0, 4'b0100, 0, 1, 0, 0, 0);     // AND
107         run_test(240, 15, 0, 4'b0101, 255, 0, 0, 0, 0);   // OR
108         run_test(240, 15, 0, 4'b0110, 255, 0, 0, 0, 0);   // XOR
109         run_test(240, 15, 0, 4'b0111, 0, 1, 0, 0, 0);     // NOR
110         run_test(240, 0, 0, 4'b1000, 15, 0, 0, 0, 0);     // NOT (only A)
111
112         // Comparisons
113         run_test(255, 1, 0, 4'b1001, 1, 0, 0, 0, 0);    // Unsigned Greater Than (True)
114         run_test(127, 255, 1, 4'b1001, 1, 0, 0, 0, 0);  // Signed Greater Than (False)
115         run_test(1, 255, 0, 4'b1010, 1, 0, 0, 0, 0);    // Unsigned Less Than (True)
116         run_test(128, 127, 1, 4'b1010, 1, 0, 0, 0, 0);  // Signed Less Than (True)
117
118         // Shift Operations
119         run_test(128, 0, 0, 4'b1100, 0, 1, 0, 1, 0);      // Logical Left Shift
120         run_test(1, 0, 0, 4'b1101, 0, 1, 0, 1, 0);        // Logical Right Shift
121         run_test(128, 0, 1, 4'b1110, 192, 0, 1, 0, 0);    // Arithmetic Right Shift
122
123         $display("------------------------------------------------------------");

124         $finish;
125     end
126 endmodule
127
```

Future Improvements

- Extend the testbench to cover additional bit widths beyond 8-bit configurations.
- Implement pipelining or parallel processing for higher performance.
- Add support for floating-point operations.
- Develop more advanced testing