Design and Simulation of 32-Function 32-Bit Arithmetic and Logical Unit (ALU)

Group ID: **CG7**

P K Koushik	CB.EN.U4ECE21240
Yoshni Nandha Kishore	CB.EN.U4ECE21265

Abstract

The goal of this project is the design and simulation of a 32-bit Arithmetic and Logic Unit, capable of producing 32 unique functions. The ALU is one of the most basic building blocks of a digital processor, and generally speaking it is the heart of the processor executing arithmetic and logical operations. All kinds of computations, from simple addition or subtraction to the bitwise manipulation and the data conversions, are to be included in this scope. ALU is part of processors and microcontrollers in almost all sorts of digital systems and therefore is an important constituent of modern computational architectures.

The design of ALU in this project would be based on Verilog HDL; that is to say, using this, the people will be able to make a highly flexible and scalable design. Hardware modelling, especially for the purpose of being very efficient in describing digital systems, uses Verilog. Our ALU can take two 32-bit operands as inputs for execution based on a 5-bit opcode, which selects one of the 32 available operations. These include basic arithmetic functions like addition, subtraction, multiplication, and division. It also carries out logical operations which feature to include AND, OR, XOR, and XNOR. Also given are advanced ones like left and right shifts, parity checking, as well as code conversions like binary-to-grey and binary-to-BCD.

It is made with efficiency, speed, and precision in mind to suit modern requirements. The design is optimized to support high performance with low power consumption that makes it preferably utilized in many digital systems and processors. The ALU scalability gives its adaptability to more intricate systems, and the needed additions of more functions can also be incorporated. The 32-bit width enables the ALU to process large data sets, which is increasingly critical in applications such as data processing, encryption, and communication systems.

We have simulated the ALU in Vivado, where we checked our design by verification to confirm that all the operations are carried out correctly. Vivado is an HDL-based tool for high-performance digital system verification. It provides detailed information regarding the behavior of the ALU during operation. A number of test cases have been applied to the ALU to check the correctness of every operation. The outcomes from the simulation are analyzed, and the outputs of every operation are captured to compare them with expected results.

Another innovation the ALU's design brings is the carry and zero flags, which are useful for operations involving conditional performances or flow-control in a

processor. This flag provides feedback about operation results, showing whether there has been an overflow or if the result is zero. These are crucial functions in making decisions in digital circuits.

In conclusion, the designed 32-bit ALU is versatile, efficient, and well suited for varying applications, from simple calculators up to complex processors in digital systems. This assumption of reliable functionality for modern computational tasks is taken by the ALU with the added design flexibility into the power of Verilog HDL and Vivado simulation. The project, thus, puts forward the critical application of ALUs in digital systems and illustrates the effectiveness of HDL-based design and simulation.

Objectives

1. Design 32-bit ALU, which should support 32 arithmetic and logical functions. It should be designed with the help of Verilog HDL.
2. Optimization of ALU at the same time in terms of high speed with low power consumption. It should also be scalable.
3. For the validation and simulation of the ALU, use Vivado, and for this, perform a number of test cases to verify all aspects of ALU.
4. Complex operations that will be performed by the ALU, such as left and right shift, parity checking, and code conversion.

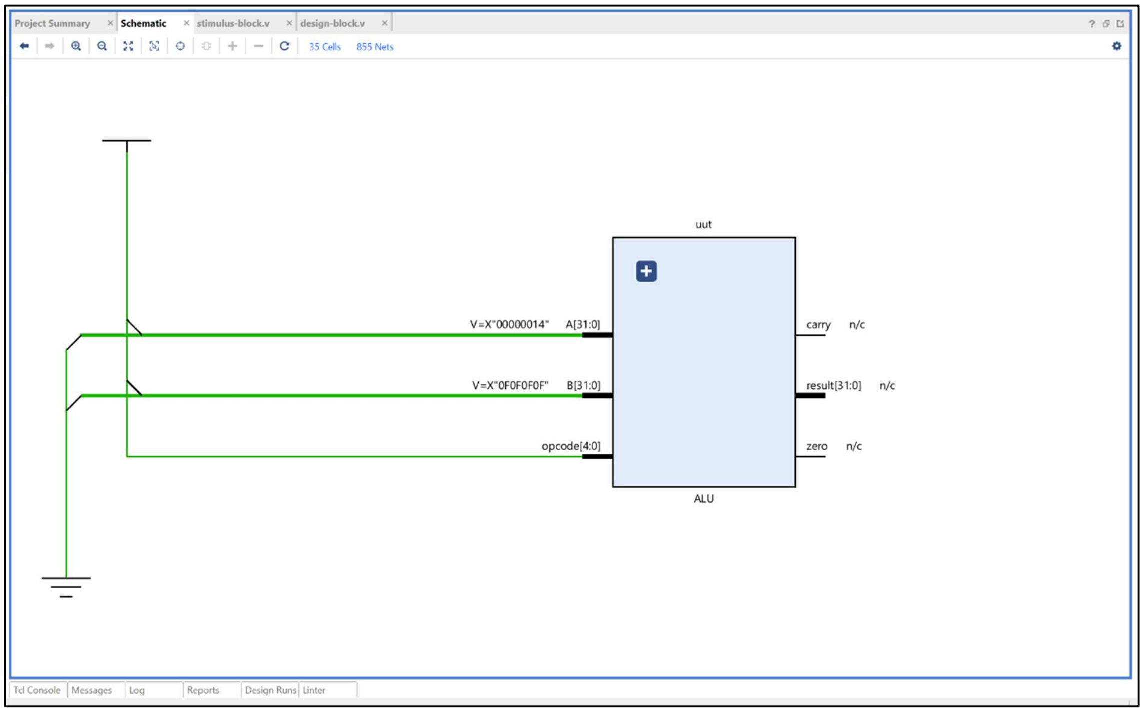
Methodology / Block Diagram

The ALU design follows an RTL approach, where each operation is selected based on a 5-bit opcode. The ALU has two 32-bit inputs (A and B) and produces a 32-bit result along with additional flags like carry and zero. The selection of the operation is done using a truth table corresponding to 32 operations, which are categorized as follows:

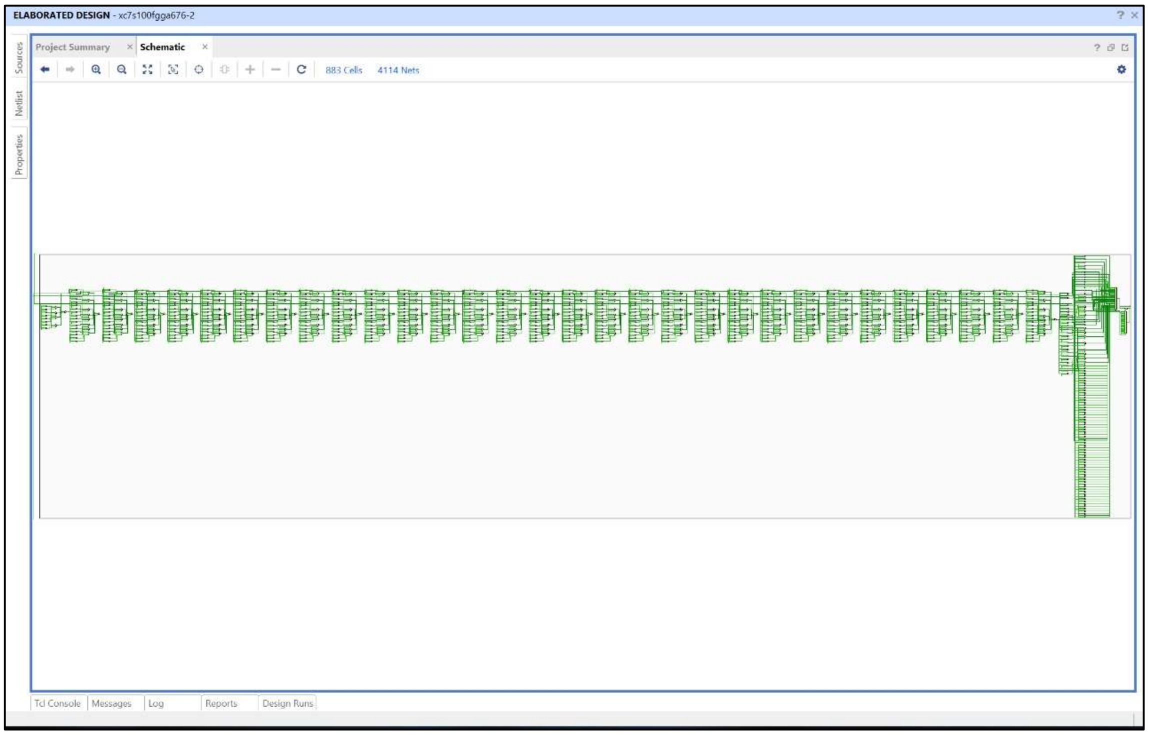
Category	Opcode	Operation	Description
Arithmetic Operations	5'b00001	Addition	A + B
	5'b00010	Subtraction	A - B
	5'b00011	Multiplication	A * B
	5'b00100	Division	A / B
	5'b00101	Increment A	A + 1
	5'b00110	Decrement A	A - 1
Logical Operations	5'b00111	Logical AND	A & B
	5'b01000	Logical OR	A | B
	5'b01001	Logical NOT A	~A
	5'b01010	Bitwise AND	A & B
	5'b01011	Bitwise OR	A | B
	5'b01100	Bitwise NAND	~(A & B)
	5'b01101	Bitwise NOR	~(A | B)
	5'b01110	Bitwise XOR	A ^ B
	5'b01111	Bitwise XNOR	~(A ^ B)
Shift Operations	5'b10110	Left Shift A by B positions	A << B
	5'b10111	Right Shift A by B positions	A >> B
Comparators	5'b11000	A == B	(A == B) ? 1 : 0
	5'b11001	A >= B	(A >= B) ? 1 : 0
	5'b11010	A <= B	(A <= B) ? 1 : 0
Code Conversion	5'b11011	Binary to Gray Code A	A ^ (A >> 1)
	5'b11100	Binary to Gray Code B	B ^ (B >> 1)
	5'b11101	Gray to Binary A	separate function
	5'b11110	Gray to Binary B	separate function
	5'b11111	Binary to BCD A	separate function
Parity Checking	5'b10000	Parity Checker A	^A
	5'b10001	Parity Checker B	^B
Complement Operations	5'b10010	1's Complement of A	~A
	5'b10011	1's Complement of B	~B
	5'b10100	2's Complement of A	-A
	5'b10101	2's Complement of B	-B
Other Operations	5'b00000	Reset	32'b0 (Reset)

The ALU is modelled using Verilog, with separate modules for the ALU and the testbench. The simulation tests a variety of cases to ensure that each function performs as expected.

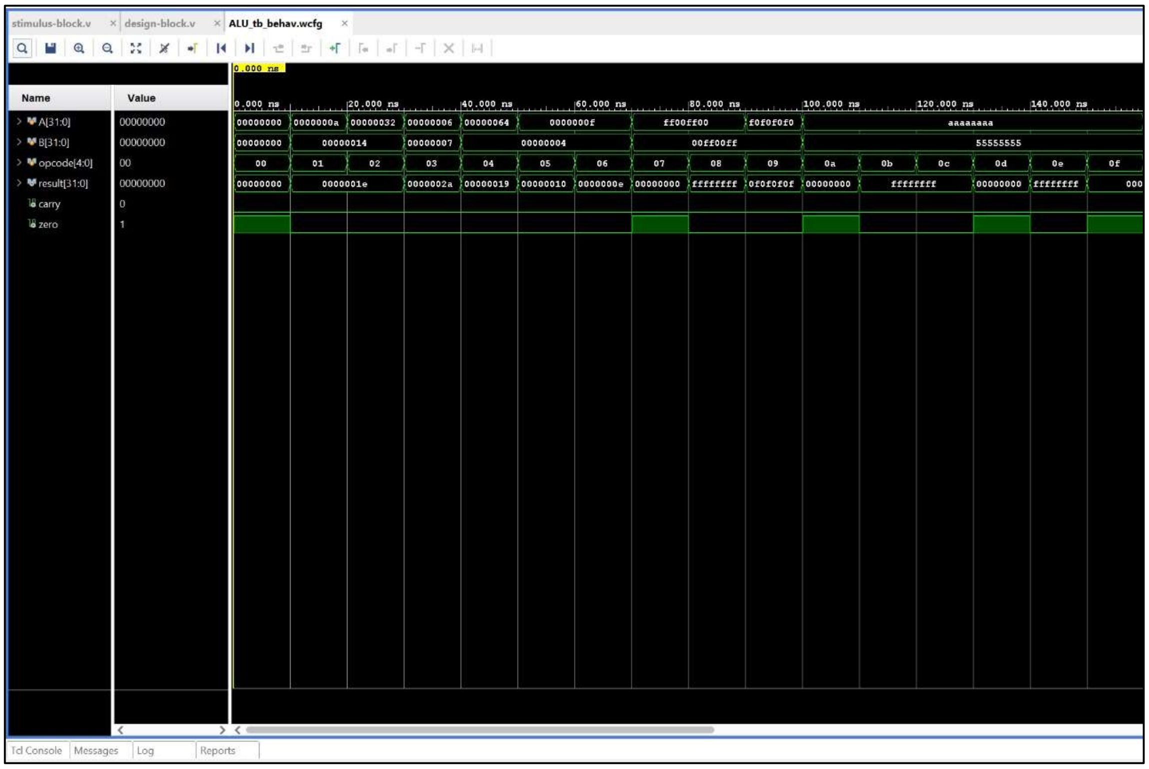
Block Diagram:

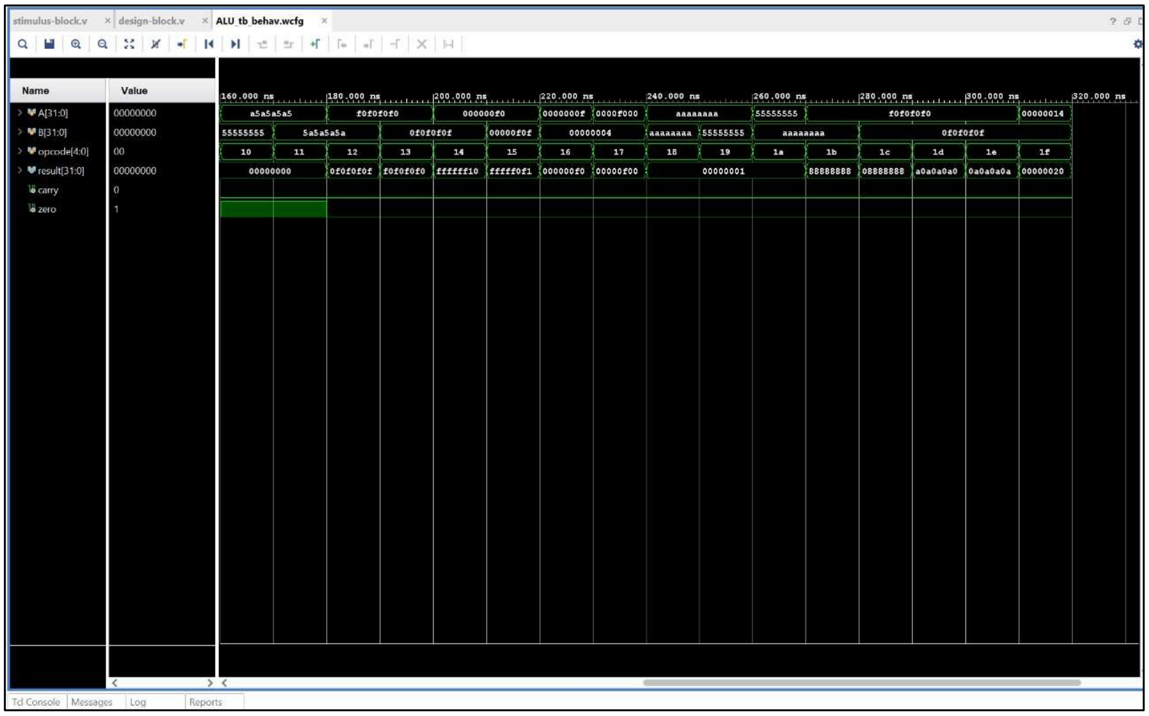


Detailed Schematic:



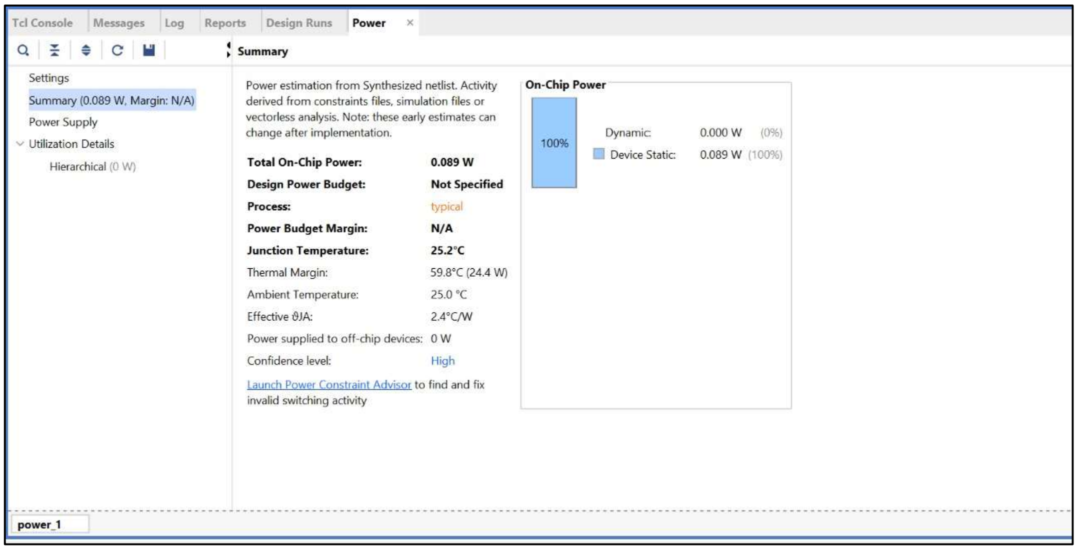
Simulation Results


stimulus-block.v
design-block.v
ALU_tb_behav.wcfg
Name
Value
A[31:0] 00000000
B[31:0] 00000000
opcode[4:0] 00
result[31:0] 00000000
carry 0
zero 1
0.000 ns
A: 00000000 0000000a 00000032 00000006 00000064 0000000f ff00ff00 f0f0f0f0 aaaaaaaa
B: 00000000 00000014 00000007 00000004 00ff00ff 55555555
opcode: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
result: 00000000 0000001e 0000002a 00000019 00000010 0000000e 00000000 ffffffff 0f0f0f0f 00000000 ffffffff 00000000 ffffffff 000
Tcl Console
Messages
Log
Reports


stimulus-block.v
design-block.v
ALU_tb_behav.wcfg
Name
Value
A[31:0] 00000000
B[31:0] 00000000
opcode[4:0] 00
result[31:0] 00000000
carry 0
zero 1
A: a5a5a5a5 f0f0f0f0 000000f0 0000000f 0000f000 aaaaaaaa 55555555 f0f0f0f0 00000014
B: 55555555 5a5a5a5a 0f0f0f0f 00000f0f 00000004 aaaaaaaa 55555555 aaaaaaaa 0f0f0f0f
opcode: 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
result: 00000000 0f0f0f0f f0f0f0f0 ffffff10 fffff0f1 000000f0 0000f00 00000001 88888888 08888888 a0a0a0a0 0a0a0a0a 00000020
Tcl Console
Messages
Log
Reports

Inference and Results

Power Estimation Results:



Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.089 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.2°C**
Thermal Margin:	59.8°C (24.4 W)
Ambient Temperature:	25.0 °C
Effective θJA:	2.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

100%

Dynamic:	0.000 W	(0%)
▨ Device Static:	0.089 W	(100%)

1. The 32-bit 32-function ALU was successfully implemented and simulated using Verilog HDL. All operations were verified against the expected results.
2. The design is flexible and scalable, meaning it can easily be extended to higher bit-widths or additional operations.
3. The inclusion of advanced functions like binary to grey code and BCD conversions increases the versatility of the ALU for various applications.
4. Simulations show that the ALU performs all operations within acceptable timing constraints, making it suitable for real-time applications.
5. The ALU can be optimized further by implementing signed arithmetic and floating-point operations.

Appendix

Module-1: ALU:

```
module ALU (
   input [31:0] A, B,          // 32-bit inputs A and B
   input [4:0] opcode,         // 5-bit opcode for 32 operations
   output reg [31:0] result,    // 32-bit result
   output reg carry,           // Carry output
   output reg zero             // Zero flag
);

   // Function for Gray to Binary Conversion
   function [31:0] gray_to_binary;
      input [31:0] gray;
      integer i;
      begin
         gray_to_binary[31] = gray[31];  // MSB remains the same
         for (i = 30; i >= 0; i = i - 1)
            gray_to_binary[i] = gray_to_binary[i+1] ^ gray[i];  // XOR of next MSB and current bit
      end
   endfunction

   // Updated Binary to BCD Conversion
   function [31:0] binary_to_bcd;
      input [31:0] binary;
      reg [31:0] bcd;  // BCD output
      integer i;
      begin
         bcd = 32'b0;  // Initialize BCD result to zero
         for (i = 31; i >= 0; i = i - 1) begin
            // Shift the result to the left by 1 bit
            bcd = bcd << 1;
            bcd[0] = binary[i];  // Shift in the next bit of binary

            // Add 3 to each BCD digit if it's greater than 4
            if (bcd[3:0] > 4)
               bcd[3:0] = bcd[3:0] + 3;
            if (bcd[7:4] > 4)
               bcd[7:4] = bcd[7:4] + 3;
            if (bcd[11:8] > 4)
               bcd[11:8] = bcd[11:8] + 3;
            if (bcd[15:12] > 4)
               bcd[15:12] = bcd[15:12] + 3;
            if (bcd[19:16] > 4)
               bcd[19:16] = bcd[19:16] + 3;
            if (bcd[23:20] > 4)
               bcd[23:20] = bcd[23:20] + 3;
            if (bcd[27:24] > 4)
               bcd[27:24] = bcd[27:24] + 3;
            if (bcd[31:28] > 4)
               bcd[31:28] = bcd[31:28] + 3;
         end
         binary_to_bcd = bcd;
      end
   endfunction
```

```
    always @(*) begin
        carry = 0;
        zero = 0;
        case (opcode)
            5'b00000: result = 32'b0;                   // Reset
            5'b00001: {carry, result} = A + B;          // Addition
            5'b00010: result = A - B;                   // Subtraction
            5'b00011: result = A * B;                   // Multiplication
            5'b00100: result = A / B;                   // Division
            5'b00101: result = A + 1;                   // Increment A
            5'b00110: result = A - 1;                   // Decrement A
            5'b00111: result = A & B;                   // Logical AND
            5'b01000: result = A | B;                   // Logical OR
            5'b01001: result = ~A;                      // Logical NOT A
            5'b01010: result = A & B;                   // Bitwise AND
            5'b01011: result = A | B;                   // Bitwise OR
            5'b01100: result = ~(A & B);                // Bitwise NAND
            5'b01101: result = ~(A | B);                // Bitwise NOR
            5'b01110: result = A ^ B;                   // Bitwise XOR
            5'b01111: result = ~(A ^ B);                // Bitwise XNOR
            5'b10000: result = ^A;                      // Parity Checker A (XOR all bits of A)
            5'b10001: result = ^B;                      // Parity Checker B (XOR all bits of B)
            5'b10010: result = ~A;                      // 1's Complement of A
            5'b10011: result = ~B;                      // 1's Complement of B
            5'b10100: result = -A;                      // 2's Complement of A
            5'b10101: result = -B;                      // 2's Complement of B
            5'b10110: result = A << B;                  // Left Shift A by B positions
            5'b10111: result = A >> B;                  // Right Shift A by B positions
            5'b11000: result = (A == B) ? 1 : 0;        // A == B
            5'b11001: result = (A >= B) ? 1 : 0;        // A >= B
            5'b11010: result = (A <= B) ? 1 : 0;        // A <= B
            5'b11011: result = A ^ (A >> 1);            // Binary to Gray Code A
            5'b11100: result = B ^ (B >> 1);            // Binary to Gray Code B
            5'b11101: result = gray_to_binary(A);       // Gray to Binary A
            5'b11110: result = gray_to_binary(B);       // Gray to Binary B
            5'b11111: result = binary_to_bcd(A);        // Binary to BCD A
            default: result = 32'hDEAD_BEEF;            // Default case (Invalid opcode)
        endcase

        // Set Zero flag
        if (result == 32'b0)
            zero = 1;
    end
endmodule
```

Module-2: Testbench

```
module ALU_tb;

  // Inputs
  reg [31:0] A, B;
  reg [4:0] opcode;

  // Outputs
  wire [31:0] result;
  wire carry, zero;

  // Instantiate the ALU module
  ALU uut (
    .A(A),
    .B(B),
    .opcode(opcode),
    .result(result),
    .carry(carry),
    .zero(zero)
  );

  // Initial block to apply stimulus
  initial begin
    // Test case 1: Reset (Opcode = 00000)
    A = 32'd0;
    B = 32'd0;
    opcode = 5'b00000;
    #10;
    $display("Opcode: %b, Result: %h, Carry: %b, Zero: %b", opcode, result, carry, zero);

    // Test case 2: Addition (Opcode = 00001)
    A = 32'd10;
    B = 32'd20;
    opcode = 5'b00001;
    #10;
    $display("Opcode: %b, A: %d, B: %d, Result: %d, Carry: %b, Zero: %b", opcode, A, B, result, carry, zero);

    // Test case 3: Subtraction (Opcode = 00010)
    A = 32'd50;
    B = 32'd20;
    opcode = 5'b00010;
    #10;
    $display("Opcode: %b, A: %d, B: %d, Result: %d, Zero: %b", opcode, A, B, result, zero);

    // Test case 4: Multiplication (Opcode = 00011)
    A = 32'd6;
    B = 32'd7;
    opcode = 5'b00011;
    #10;
    $display("Opcode: %b, A: %d, B: %d, Result: %d", opcode, A, B, result);

    // Test case 5: Division (Opcode = 00100)
    A = 32'd100;
    B = 32'd4;
    opcode = 5'b00100;
```

```
#10;
$display("Opcode: %b, A: %d, B: %d, Result: %d", opcode, A, B, result);

// Test case 6: Increment A (Opcode = 00101)
A = 32'd15;
opcode = 5'b00101;
#10;
$display("Opcode: %b, A: %d, Result: %d", opcode, A, result);

// Test case 7: Decrement A (Opcode = 00110)
A = 32'd15;
opcode = 5'b00110;
#10;
$display("Opcode: %b, A: %d, Result: %d", opcode, A, result);

// Test case 8: Logical AND (Opcode = 00111)
A = 32'hFF00_FF00;
B = 32'h00FF_00FF;
opcode = 5'b00111;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 9: Logical OR (Opcode = 01000)
A = 32'hFF00_FF00;
B = 32'h00FF_00FF;
opcode = 5'b01000;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 10: Logical NOT A (Opcode = 01001)
A = 32'hF0F0_F0F0;
opcode = 5'b01001;
#10;
$display("Opcode: %b, A: %h, Result: %h", opcode, A, result);

// Test case 11: Bitwise AND (Opcode = 01010)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01010;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 12: Bitwise OR (Opcode = 01011)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01011;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 13: Bitwise NAND (Opcode = 01100)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01100;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);
```

```
// Test case 14: Bitwise NOR (Opcode = 01101)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01101;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 15: Bitwise XOR (Opcode = 01110)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01110;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 16: Bitwise XNOR (Opcode = 01111)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b01111;
#10;
$display("Opcode: %b, A: %h, B: %h, Result: %h", opcode, A, B, result);

// Test case 17: Parity Checker A (Opcode = 10000)
A = 32'hA5A5_A5A5;
opcode = 5'b10000;
#10;
$display("Opcode: %b, A: %h, Parity A Result: %d", opcode, A, result);

// Test case 18: Parity Checker B (Opcode = 10001)
B = 32'h5A5A_5A5A;
opcode = 5'b10001;
#10;
$display("Opcode: %b, B: %h, Parity B Result: %d", opcode, B, result);

// Test case 19: 1's Complement of A (Opcode = 10010)
A = 32'hF0F0_F0F0;
opcode = 5'b10010;
#10;
$display("Opcode: %b, A: %h, 1's Complement Result: %h", opcode, A, result);

// Test case 20: 1's Complement of B (Opcode = 10011)
B = 32'h0F0F_0F0F;
opcode = 5'b10011;
#10;
$display("Opcode: %b, B: %h, 1's Complement Result: %h", opcode, B, result);

// Test case 21: 2's Complement of A (Opcode = 10100)
A = 32'h0000_00F0;
opcode = 5'b10100;
#10;
$display("Opcode: %b, A: %h, 2's Complement Result: %h", opcode, A, result);

// Test case 22: 2's Complement of B (Opcode = 10101)
B = 32'h0000_0F0F;
opcode = 5'b10101;
```

```
#10;
$display("Opcode: %b, B: %h, 2's Complement Result: %h", opcode, B, result);

// Test case 23: Left Shift A by B (Opcode = 10110)
A = 32'h0000_000F;
B = 32'd4;
opcode = 5'b10110;
#10;
$display("Opcode: %b, A: %h, B: %d, Left Shift Result: %h", opcode, A, B, result);

// Test case 24: Right Shift A by B (Opcode = 10111)
A = 32'h0000_F000;
B = 32'd4;
opcode = 5'b10111;
#10;
$display("Opcode: %b, A: %h, B: %d, Right Shift Result: %h", opcode, A, B, result);

// Test case 25: A == B (Opcode = 11000)
A = 32'hAAAA_AAAA;
B = 32'hAAAA_AAAA;
opcode = 5'b11000;
#10;
$display("Opcode: %b, A: %h, B: %h, Result (A==B): %d", opcode, A, B, result);

// Test case 26: A >= B (Opcode = 11001)
A = 32'hAAAA_AAAA;
B = 32'h5555_5555;
opcode = 5'b11001;
#10;
$display("Opcode: %b, A: %h, B: %h, Result (A>=B): %d", opcode, A, B, result);

// Test case 27: A <= B (Opcode = 11010)
A = 32'h5555_5555;
B = 32'hAAAA_AAAA;
opcode = 5'b11010;
#10;
$display("Opcode: %b, A: %h, B: %h, Result (A<=B): %d", opcode, A, B, result);

// Test case 28: Binary to Gray Code A (Opcode = 11011)
A = 32'hF0F0_F0F0;
opcode = 5'b11011;
#10;
$display("Opcode: %b, A: %h, Gray Code Result: %h", opcode, A, result);

// Test case 29: Binary to Gray Code B (Opcode = 11100)
B = 32'h0F0F_0F0F;
opcode = 5'b11100;
#10;
$display("Opcode: %b, B: %h, Gray Code Result: %h", opcode, B, result);

// Test case 30: Gray to Binary A (Opcode = 11101)
A = 32'hF0F0_F0F0;
opcode = 5'b11101;
#10;
$display("Opcode: %b, A: %h, Gray to Binary Result: %h", opcode, A, result);
```

```
        // Test case 31: Gray to Binary B (Opcode = 11110)
        B = 32'h0F0F_0F0F;
        opcode = 5'b11110;
        #10;
        $display("Opcode: %b, B: %h, Gray to Binary Result: %h", opcode, B, result);

        // Test case 32: Binary to BCD A (Opcode = 11111)
        A = 32'd20;
        opcode = 5'b11111;
        #10;
        $display("Opcode: %b, A: %d, Binary to BCD Result: %h", opcode, A, result);

        // End of the test
        $stop;
    end
endmodule
```