Final Step

Group: Group 2

Members: Robert Tyler Jones, Javon Portis, Daniel Arreguin

July 2024

1 Introduction

The following is an extension of the previous steps. In this phase of the project, the previously defined functions will be integrated as modules and controlled by a control circuit. Operations will be tested with different bit sizes, and waveforms will be generated to verify the correctness of the logic operations.

2 Verilog

This Verilog code defines a parameterized Arithmetic Logic Unit (ALU) capable of performing various arithmetic and logical operations, including AND, NAND, OR, NOR, XOR, XNOR, NOT, shifting, addition, subtraction, multiplication, and division. The ALU supports multiple bit-widths (4-bit, 8-bit, 16-bit, and 32-bit) through parameterization, allowing flexibility in its application. It includes a testbench to verify functionality, which cycles through different operations and bit-widths, displaying results for each operation. The code integrates these components into a single file for streamlined simulation and testing.

2.1 Binary Logic Operation Modules

The following defines Logic operations, such as AND, NAND, OR, NOR, XOR and NOT.

```
// Parameterized AND
module and_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = a & b;
endmodule

// Parameterized NAND
module nand_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = ~(a & b);
endmodule

// Parameterized OR
module or_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = a | b;
endmodule

// Parameterized NOR
module nor_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = ~(a | b);
endmodule

// Parameterized XOR
module xor_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = a ^ b;
endmodule

// Parameterized XNOR
module xnor_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] y
);
    assign y = ~(a ^ b);
endmodule
```

```
54
55 // Parameterized NOT
56 module not_param #(parameter WIDTH = 4) (
57     input [WIDTH-1:0] a,
58     output [WIDTH-1:0] y
59 );
60     assign y = ~a;
61 endmodule
```

Listing 1: Binary Logic Operations Module

2.2 Shifter

The parameterized shifter takes an input "a" and shifts it based on the shift value. If shift is 00, a stays the same. If 01, it shifts left with a zero at the end. If 10, it shifts right with a zero at the start. Any other shift value results in all zeros.

```
1  // Parameterized Shifter
2  module shifter_param #(parameter WIDTH = 4) (
3      input [WIDTH-1:0] a,
4      input [1:0] shift,
5      output [WIDTH-1:0] y
6  );
7      assign y = (shift == 2'b00) ? a :
8                 (shift == 2'b01) ? {a[WIDTH-2:0], 1'b0} :
9                 (shift == 2'b10) ? {1'b0, a[WIDTH-1:1]} :
10                                   {WIDTH{1'b0}}; // Undefined shift
      , outputs 0
11 endmodule
```

Listing 2: Shifter

2.3 Arithmetic Integer Operations Modules

The following code defines arithmetic operations, such as add, subtraction, multiplication, and division.

```
1  // Parameterized Addition
2  module add_param #(parameter WIDTH = 4) (
3      input [WIDTH-1:0] a,
4      input [WIDTH-1:0] b,
5      input cin,
6      output [WIDTH-1:0] sum,
7      output cout
8  );
9      assign {cout, sum} = a + b + cin;
10 endmodule
11
12 // Parameterized Subtraction
13 module sub_param #(parameter WIDTH = 4) (
14     input [WIDTH-1:0] a,
15     input [WIDTH-1:0] b,
16     input cin,
```

```
    output [WIDTH-1:0] diff,
    output cout
);
    assign {cout, diff} = a - b - cin;
endmodule

// Parameterized Multiplication
module mul_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [(2*WIDTH)-1:0] product
);
    assign product = a * b;
endmodule

// Parameterized Division
module div_param #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    output [WIDTH-1:0] quotient,
    output [WIDTH-1:0] remainder
);
    assign quotient = a / b;
    assign remainder = a % b;
endmodule
```

Listing 3: Arithmetic Integer Operations Modules

2.4 ALU (Arithmetic Logic Unit) and Control Unit

This Verilog module implements a parameterized Arithmetic Logic Unit (ALU) capable of performing various operations based on a control signal. It integrates multiple operation modules (e.g., AND, OR, addition) and uses a control signal to select and execute the desired operation, producing outputs such as results, carry-out, product, or remainder. The module supports different bit-widths and generates the appropriate outputs for each operation type, including logical functions, arithmetic operations, shifting, and division.

```
// Top-level ALU module
module alu #(parameter WIDTH = 4) (
    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    input cin,
    input [1:0] shift,
    input [3:0] control,  // Control signal to select operation
    output reg [WIDTH-1:0] result,
    output reg cout,
    output reg [WIDTH-1:0] remainder, // Only for division
    output reg [(2*WIDTH)-1:0] product // Only for multiplication
);
    // Intermediate signals
    wire [WIDTH-1:0] y_and, y_nand, y_or, y_nor, y_xor, y_xnor,
    y_not, y_shift, sum, diff, quotient, temp_remainder;
    wire cout_add, cout_sub;
    wire [(2*WIDTH)-1:0] temp_product;
```

```
17
18      // Instantiate all operation modules
19      and_param #(WIDTH) u_and (.a(a), .b(b), .y(y_and));
20      nand_param #(WIDTH) u_nand (.a(a), .b(b), .y(y_nand));
21      or_param #(WIDTH) u_or (.a(a), .b(b), .y(y_or));
22      nor_param #(WIDTH) u_nor (.a(a), .b(b), .y(y_nor));
23      xor_param #(WIDTH) u_xor (.a(a), .b(b), .y(y_xor));
24      xnor_param #(WIDTH) u_xnor (.a(a), .b(b), .y(y_xnor));
25      not_param #(WIDTH) u_not (.a(a), .y(y_not));
26      shifter_param #(WIDTH) u_shifter (.a(a), .shift(shift), .y(
        y_shift));
27      add_param #(WIDTH) u_add (.a(a), .b(b), .cin(cin), .sum(sum), .
        cout(cout_add));
28      sub_param #(WIDTH) u_sub (.a(a), .b(b), .cin(cin), .diff(diff),
         .cout(cout_sub));
29      mul_param #(WIDTH) u_mul (.a(a), .b(b), .product(temp_product))
        ;
30      div_param #(WIDTH) u_div (.a(a), .b(b), .quotient(quotient), .
        remainder(temp_remainder));
31
32      // Control circuit to select the operation
33      always @(*) begin
34          case (control)
35              4'b0000: begin result = y_and; cout = 0; end      //
        AND
36              4'b0001: begin result = y_nand; cout = 0; end     //
        NAND
37              4'b0010: begin result = y_or; cout = 0; end       // OR
38              4'b0011: begin result = y_nor; cout = 0; end      //
        NOR
39              4'b0100: begin result = y_xor; cout = 0; end      //
        XOR
40              4'b0101: begin result = y_xnor; cout = 0; end     //
        XNOR
41              4'b0110: begin result = y_not; cout = 0; end      //
        NOT
42              4'b0111: begin result = y_shift; cout = 0; end    //
        Shifter
43              4'b1000: begin result = sum; cout = cout_add; end //
        Addition
44              4'b1001: begin result = diff; cout = cout_sub; end //
        Subtraction
45              4'b1010: begin product = temp_product; cout = 0; end
        // Multiplication
46              4'b1011: begin result = quotient; cout = 0; remainder =
         temp_remainder; end // Division
47              default: begin result = {WIDTH{1'b0}}; cout = 0; end
48          endcase
49      end
50  endmodule
```

Listing 4: ALU (Arithmetic Logic Unit)

2.5 Test Bench

This testbench module defines and tests a range of bit-widths (4, 8, 16, 32) for various logical, arithmetic, and shift operations. It uses registers to set inputs and wires to capture the outputs of these operations. For each bit-width, different logical operations (AND, OR, XOR, etc.), arithmetic operations (addition, subtraction, multiplication, division), and shift operations are instantiated and tested. The initial block sets up test cases for each bit-width, displays the inputs and results, and saves the waveform data to a file.

```
// Testbench
module testbench;
    // Inputs
    reg [31:0] a, b;
    reg cin;
    reg [1:0] shift;
    reg [3:0] control;

    // Outputs
    wire [31:0] result;
    wire cout;
    wire [31:0] remainder;
    wire [63:0] product;

    // Instantiate the Unit Under Test (UUT)
    alu #(32) uut (
        .a(a),
        .b(b),
        .cin(cin),
        .shift(shift),
        .control(control),
        .result(result),
        .cout(cout),
        .remainder(remainder),
        .product(product)
    );

    initial begin
        // Initialize Inputs
        $dumpfile("finalStep.vcd");
        $dumpvars(0, testbench);

        // Test 4-bit ALU
        a = 4'b0001; b = 4'b0010; cin = 1'b0; shift = 2'b00;

        // Test each operation
        control = 4'b0000; // AND
        #10; $display("4-bit Test - AND: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0001; // NAND
        #10; $display("4-bit Test - NAND: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0010; // OR
        #10; $display("4-bit Test - OR: a = %b, b = %b, result = %b
```

```
    ", a, b, result);

        control = 4'b0011; // NOR
        #10; $display("4-bit Test - NOR: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0100; // XOR
        #10; $display("4-bit Test - XOR: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0101; // XNOR
        #10; $display("4-bit Test - XNOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0110; // NOT
        #10; $display("4-bit Test - NOT: a = %b, result = %b", a,
    result);

        shift = 2'b01; control = 4'b0111; // Shifter
        #10; $display("4-bit Test - Shifter: a = %b, shift = %b,
    result = %b", a, shift, result);

        cin = 1'b1; control = 4'b1000; // Addition
        #10; $display("4-bit Test - Addition: a = %b, b = %b, cin =
     %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1001; // Subtraction
        #10; $display("4-bit Test - Subtraction: a = %b, b = %b,
    cin = %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1010; // Multiplication
        #10; $display("4-bit Test - Multiplication: a = %b, b = %b,
     product = %b", a, b, product);

        control = 4'b1011; // Division
        #10; $display("4-bit Test - Division: a = %b, b = %b,
    quotient = %b, remainder = %b", a, b, result, remainder);

        // Test 8-bit ALU
        a = 8'h01; b = 8'h02; control = 4'b0000; // AND
        #10; $display("8-bit Test - AND: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0001; // NAND
        #10; $display("8-bit Test - NAND: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0010; // OR
        #10; $display("8-bit Test - OR: a = %b, b = %b, result = %b
    ", a, b, result);

        control = 4'b0011; // NOR
        #10; $display("8-bit Test - NOR: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0100; // XOR
        #10; $display("8-bit Test - XOR: a = %b, b = %b, result = %
```

```
    b", a, b, result);

        control = 4'b0101; // XNOR
        #10; $display("8-bit Test - XNOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0110; // NOT
        #10; $display("8-bit Test - NOT: a = %b, result = %b", a,
    result);

        shift = 2'b01; control = 4'b0111; // Shifter
        #10; $display("8-bit Test - Shifter: a = %b, shift = %b,
    result = %b", a, shift, result);

        cin = 1'b1; control = 4'b1000; // Addition
        #10; $display("8-bit Test - Addition: a = %b, b = %b, cin =
     %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1001; // Subtraction
        #10; $display("8-bit Test - Subtraction: a = %b, b = %b,
    cin = %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1010; // Multiplication
        #10; $display("8-bit Test - Multiplication: a = %b, b = %b,
     product = %b", a, b, product);

        control = 4'b1011; // Division
        #10; $display("8-bit Test - Division: a = %b, b = %b,
    quotient = %b, remainder = %b", a, b, result, remainder);

        // Test 16-bit ALU
        a = 16'h0001; b = 16'h0002; control = 4'b0000; // AND
        #10; $display("16-bit Test - AND: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0001; // NAND
        #10; $display("16-bit Test - NAND: a = %b, b = %b, result =
     %b", a, b, result);

        control = 4'b0010; // OR
        #10; $display("16-bit Test - OR: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0011; // NOR
        #10; $display("16-bit Test - NOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0100; // XOR
        #10; $display("16-bit Test - XOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0101; // XNOR
        #10; $display("16-bit Test - XNOR: a = %b, b = %b, result =
     %b", a, b, result);

        control = 4'b0110; // NOT
        #10; $display("16-bit Test - NOT: a = %b, result = %b", a,
```

```
    result);

        shift = 2'b01; control = 4'b0111; // Shifter
        #10; $display("16-bit Test - Shifter: a = %b, shift = %b,
    result = %b", a, shift, result);

        cin = 1'b1; control = 4'b1000; // Addition
        #10; $display("16-bit Test - Addition: a = %b, b = %b, cin
    = %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1001; // Subtraction
        #10; $display("16-bit Test - Subtraction: a = %b, b = %b,
    cin = %b, result = %b, cout = %b", a, b, cin, result, cout);

        control = 4'b1010; // Multiplication
        #10; $display("16-bit Test - Multiplication: a = %b, b = %b
    , product = %b", a, b, product);

        control = 4'b1011; // Division
        #10; $display("16-bit Test - Division: a = %b, b = %b,
    quotient = %b, remainder = %b", a, b, result, remainder);

        // Test 32-bit ALU
        a = 32'h00000001; b = 32'h00000002; control = 4'b0000; //
    AND
        #10; $display("32-bit Test - AND: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0001; // NAND
        #10; $display("32-bit Test - NAND: a = %b, b = %b, result =
     %b", a, b, result);

        control = 4'b0010; // OR
        #10; $display("32-bit Test - OR: a = %b, b = %b, result = %
    b", a, b, result);

        control = 4'b0011; // NOR
        #10; $display("32-bit Test - NOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0100; // XOR
        #10; $display("32-bit Test - XOR: a = %b, b = %b, result =
    %b", a, b, result);

        control = 4'b0101; // XNOR
        #10; $display("32-bit Test - XNOR: a = %b, b = %b, result =
     %b", a, b, result);

        control = 4'b0110; // NOT
        #10; $display("32-bit Test - NOT: a = %b, result = %b", a,
    result);

        shift = 2'b01; control = 4'b0111; // Shifter
        #10; $display("32-bit Test - Shifter: a = %b, shift = %b,
    result = %b", a, shift, result);

        cin = 1'b1; control = 4'b1000; // Addition
```

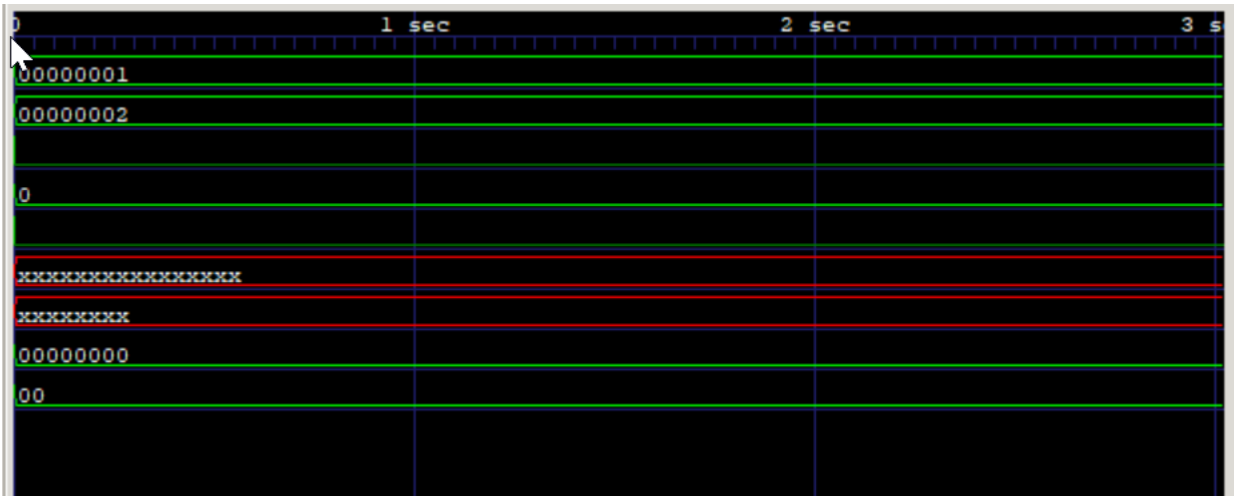
```
173         #10; $display("32-bit Test - Addition: a = %b, b = %b, cin
    = %b, result = %b, cout = %b", a, b, cin, result, cout);
174
175         control = 4'b1001; // Subtraction
176         #10; $display("32-bit Test - Subtraction: a = %b, b = %b,
    cin = %b, result = %b, cout = %b", a, b, cin, result, cout);
177
178         control = 4'b1010; // Multiplication
179         #10; $display("32-bit Test - Multiplication: a = %b, b = %b
    , product = %b", a, b, product);
180
181         control = 4'b1011; // Division
182         #10; $display("32-bit Test - Division: a = %b, b = %b,
    quotient = %b, remainder = %b", a, b, result, remainder);
183
184         $finish;
185     end
186 endmodule
```

Listing 5: Test Bench

3 Waveform Test

3.1 Wavefor

The following waveform visualizations that show the behavior and timing of inputs and outputs for the ALU module during simulation. It allows users to observe signal transitions and verify that the ALU performs operations correctly based on the control signals.



3.2 Results

3.2.1 4-bit Results

```
4-bit Test - AND: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000000
4-bit Test - NAND: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111111
4-bit Test - OR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000011
4-bit Test - NOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111100
4-bit Test - XOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000011
4-bit Test - XNOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111100
4-bit Test - NOT: a = 00000000000000000000000000000001, result = 1111111111
11111111111111111110
4-bit Test - Shifter: a = 00000000000000000000000000000001, shift = 01, res
ult = 00000000000000000000000000000010
4-bit Test - Addition: a = 00000000000000000000000000000001, b = 0000000000
00000000000000000010, cin = 1, result = 00000000000000000000000000000100,
 cout = 0
4-bit Test - Subtraction: a = 00000000000000000000000000000001, b = 0000000
00000000000000000000010, cin = 1, result = 1111111111111111111111111111111
10, cout = 1
4-bit Test - Multiplication: a = 00000000000000000000000000000001, b = 0000
00000000000000000000000010, product = 0000000000000000000000000000000000
00000000000000000000000010
4-bit Test - Division: a = 00000000000000000000000000000001, b = 0000000000
00000000000000000010, quotient = 00000000000000000000000000000000, remain
der = 00000000000000000000000000000001
```

3.3 Extra Credit Results

3.3.1 8-bit Results

```
1         // Test 8-bit ALU
2         a = 8'h01; b = 8'h02; control = 4'b0000; // AND
```

Listing 6: 8-bit input

```
8-bit Test - AND: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000000
8-bit Test - NAND: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111111
8-bit Test - OR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000011
8-bit Test - NOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111100
8-bit Test - XOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 00000000000000000000000000000011
8-bit Test - XNOR: a = 00000000000000000000000000000001, b = 00000000000000
00000000000000010, result = 11111111111111111111111111111100
8-bit Test - NOT: a = 00000000000000000000000000000001, result = 1111111111
11111111111111111110
8-bit Test - Shifter: a = 00000000000000000000000000000001, shift = 01, res
ult = 00000000000000000000000000000010
8-bit Test - Addition: a = 00000000000000000000000000000001, b = 0000000000
00000000000000000010, cin = 1, result = 00000000000000000000000000000100,
 cout = 0
8-bit Test - Subtraction: a = 00000000000000000000000000000001, b = 0000000
00000000000000000000010, cin = 1, result = 1111111111111111111111111111111
10, cout = 1
8-bit Test - Multiplication: a = 00000000000000000000000000000001, b = 0000
00000000000000000000000010, product = 00000000000000000000000000000000
00000000000000000000000010
8-bit Test - Division: a = 00000000000000000000000000000001, b = 0000000000
00000000000000000010, quotient = 00000000000000000000000000000000, remain
der = 00000000000000000000000000000001
```

3.3.2 16-bit Results

```
1         // Test 16-bit ALU
2         a = 16'h0001; b = 16'h0002; control = 4'b0000; // AND
```

Listing 7: 16-bit input

```
16-bit Test - AND: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000000
16-bit Test - NAND: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111111
16-bit Test - OR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000011
16-bit Test - NOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111100
16-bit Test - XOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000011
16-bit Test - XNOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111100
16-bit Test - NOT: a = 00000000000000000000000000000001, result = 11111111111111111111111111111110
16-bit Test - Shifter: a = 00000000000000000000000000000001, shift = 01, result = 00000000000000000000000000000010
16-bit Test - Addition: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, cin = 1, result = 00000000000000000000000000000100, cout = 0
16-bit Test - Subtraction: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, cin = 1, result = 11111111111111111111111111111110, cout = 1
16-bit Test - Multiplication: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, product = 00000000000000000000000000000000000000000000000000000000000000010
16-bit Test - Division: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, quotient = 00000000000000000000000000000000, remainder = 00000000000000000000000000000001
```

3.3.3 32-bit Results

```
1         // Test 32-bit ALU
2         a = 32'h00000001; b = 32'h00000002; control = 4'b0000; // AND
```

Listing 8: 32-bit input

```
32-bit Test - AND: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000000
32-bit Test - NAND: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111111
32-bit Test - OR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000011
32-bit Test - NOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111100
32-bit Test - XOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 00000000000000000000000000000011
32-bit Test - XNOR: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, result = 11111111111111111111111111111100
32-bit Test - NOT: a = 00000000000000000000000000000001, result = 11111111111111111111111111111110
32-bit Test - Shifter: a = 00000000000000000000000000000001, shift = 01, result = 00000000000000000000000000000010
32-bit Test - Addition: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, cin = 1, result = 00000000000000000000000000000100, cout = 0
32-bit Test - Subtraction: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, cin = 1, result = 11111111111111111111111111111110, cout = 1
32-bit Test - Multiplication: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, product = 00000000000000000000000000000000
00000000000000000000000000000010
32-bit Test - Division: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, quotient = 00000000000000000000000000000000, remainder = 00000000000000000000000000000001
00000000000000000000000000000010
32-bit Test - Division: a = 00000000000000000000000000000001, b = 00000000000000000000000000000010, quotient = 00000000000000000000000000000000, remai
00000000000000000000000000000010
```

4 Conclusion

The implementation of the parameterized ALU module successfully integrates a range of arithmetic and logical operations, with flexibility for various bit-widths. The generated waveforms from the simulation in GTKWave confirm that the ALU operates correctly across different functions and input conditions. Observations of signal transitions and output results demonstrate that the ALU responds accurately to control inputs, performing operations such as addition, subtraction, and multiplication. Overall, the module's functionality is validated, showcasing its capability to handle diverse computational tasks effectively.