EE123 VLSI Design Lab: Experiment 3 (ALU)

Saurav Kumar Pandey
System on chip design
IIT Palakkad
152502019@smail.iitpkd.ac.in

October 2025

***Abstract*—This experiment explores the design and synthesis of an 8-bit Arithmetic Logic Unit (ALU). We begin by developing two distinct models in Verilog: a high-level behavioral implementation and a more detailed structural implementation built upon a Ripple Carry Adder (RCA). Both designs are then functionally verified against a comprehensive testbench to ensure correctness. Following simulation, we synthesize both ALU versions using Synopsys Design Compiler and a 180nm standard-cell library. The core objective is to analyze and contrast the final synthesis reports, comparing the behavioral and structural approaches in terms of their real-world area, power consumption, and timing performance.**

***Index Terms*—ALU, Verilog, ASIC, Synthesis, Design Compiler, RTL, Gate-Level Netlist, 180nm.**

I. Introduction

The Arithmetic Logic Unit (ALU) is a fundamental building block in nearly every digital processor. It acts as the computational heart of the system, responsible for performing all arithmetic operations, such as addition and subtraction, as well as bitwise logical operations like AND, OR, XOR, and NOT.

Because of this central role, the ALU is a critical component in Central Processing Units (CPUs), Digital Signal Processors (DSPs), and various hardware accelerators. It is the hardware that executes the core calculations required for any instruction-level or algorithmic process.

In this experiment, we explore the design of an ALU from two different perspectives. First, we design a *behavioral* model, which describes the high-level functionality of the ALU. Second, we build a *structural* model, which defines the ALU in terms of its constituent components, specifically using a ripple-carry adder for arithmetic. Both versions will be verified through simulation and then synthesized using the Synopsys Design Compiler and an SCL 180 nm library. This allows us to compare the final implementation results and understand the practical trade-offs in area, power, and timing between the two design styles.

A. Behavioural ALU

Implemented directly using Verilog operators.

B. Structural ALU

Implemented using Ripple carry adder (RCA). And RCA is implemented using Full Adder (FA). RCA is used to perform addition and subtraction operations.

TABLE I
ALU Operations

Operation Code	Operation	Description
000	ADD	$A + B$
001	SUB	$A - B$
010	AND	A & B
011	OR	A | B
100	XOR	$A \wedge B$
101	NOT	$\sim A$
110	SHL	$A \ll 1$
111	SHR	$A \gg 1$

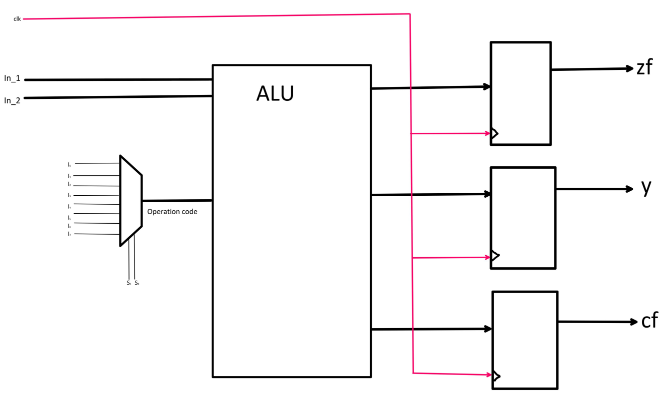

Fig. 1. RTL Diagram of ALU Behavioural

II. Verilog Code

A. Behavioural ALU

```verilog
module alu #(parameter N = 8)(clk, in_1, in_2,
    op_code, y, cf, zf);
  input wire clk;
  input wire [N-1:0] in_1, in_2;
  input wire [2:0] op_code;
  output reg [N-1:0] y;
  output reg cf, zf;

  always @(posedge clk)
  begin
    cf = 0;
    zf = 0;

    if (op_code == 3'b000) begin
      {cf, y} <= in_1 + in_2;
    end
    else if (op_code == 3'b001) begin
      {cf, y} <= in_1 - in_2;
    end
    else if (op_code == 3'b010) begin
```

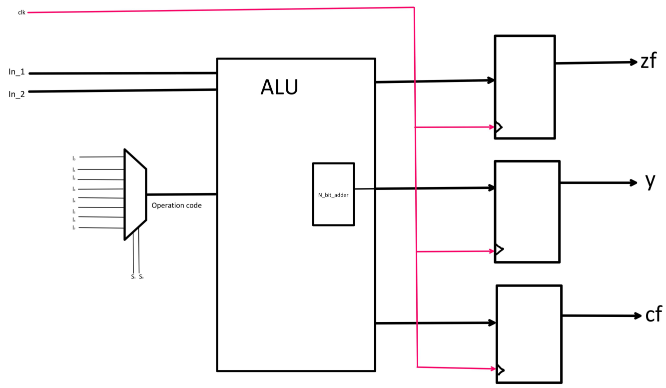

Fig. 2. RTL Diagram of ALU Structural

```
      y <= in_1 & in_2;
    end
    else if (op_code == 3'b011) begin
      y <= in_1 | in_2;
    end
    else if (op_code == 3'b100) begin
      y <= in_1 ^ in_2;
    end
    else if (op_code == 3'b101) begin
      y <= ~in_1;
    end
    else if (op_code == 3'b110) begin
      y <= in_1 << 1;
    end
    else if (op_code == 3'b111) begin
      y <= in_1 >> 1;
    end
    else begin
      y <= {N{1'b0}};
    end

    if (y == 0)
      zf <= 1;
    else
      zf <= 0;
  end
endmodule
```

Listing 1. Behavioural Verilog Code for 8-bit ALU

B. Structural ALU Verilog Code

1) Full Adder:

```
module full_add (a, b, cin, cout, sum);
  input wire a, b, cin;
  output wire cout, sum;
  assign sum = a ^ b ^ cin;
  assign cout = (a & b) | (b & cin) | (cin & a);
endmodule
```

Listing 2. Verilog Code for Full Adder (FA)

2) Ripple Carry Adder:

```
module n_bit_adder #(parameter N = 8) (A, B, Cin, Cout, Sum);
  input wire [N-1:0] A, B;
  input wire Cin;
  output wire [N-1:0] Sum;
  output wire Cout;
  wire [N:0] carry;
  assign carry[0] = Cin;
  genvar i;
  generate
    for (i = 0; i < N; i = i + 1)
    begin : adder
      full_add fa(
        .a(A[i]),
        .b(B[i]),
        .cin(carry[i]),
        .cout(carry[i+1]),
        .sum(Sum[i])
      );
    end
  endgenerate
  assign Cout = carry[N];
endmodule
```

Listing 3. Verilog Code for n-bit Ripple Carry Adder (RCA)

3) ALU using Ripple Carry Adder:

```
module alu #(parameter N = 8)(clk, in_1, in_2, cf, zf, op_code, y);
  input wire clk;
  input wire [N-1:0] in_1, in_2;
  input wire [2:0] op_code;
  output reg [N-1:0] y;
  output reg cf, zf;

  wire [N-1:0] b_mod;
  wire carry_in;
  wire [N-1:0] sum_out;
  wire carry_out;

  assign b_mod = (op_code == 3'b001) ? ~in_2 : in_2;
  assign carry_in = (op_code == 3'b001) ? 1'b1 : 1'b0;

  n_bit_adder #(N) nba_instance (
    .A(in_1),
    .B(b_mod),
    .Cin(carry_in),
    .Sum(sum_out),
    .Cout(carry_out)
  );

  always @(posedge clk)
  begin
    cf <= 0;
    zf <= 0;

    if (op_code == 3'b000) begin
      y  <= sum_out;
      cf <= carry_out;
    end
    else if (op_code == 3'b001) begin
      y  <= sum_out;
      cf <= carry_out;
    end
    else if (op_code == 3'b010) begin
      y  <= in_1 & in_2;
    end
    else if (op_code == 3'b011) begin
      y  <= in_1 | in_2;
    end
    else if (op_code == 3'b100) begin
      y  <= in_1 ^ in_2;
    end
    else if (op_code == 3'b101) begin
      y  <= ~in_1;
    end
    else if (op_code == 3'b110) begin
      y  <= in_1 << 1;
    end
    else if (op_code == 3'b111) begin
      y  <= in_1 >> 1;
    end
    else begin
      y  <= {N{1'b0}};
    end
```

```
    if (y == 0)
      zf <= 1;
    else
      zf <= 0;
  end
endmodule
```

Listing 4. Structural Verilog Code for 8-bit ALU

C. Testbench

```
`timescale 1ns/1ns

module alu_tb;
  parameter N = 8;
  reg clk;
  reg [N-1:0] in_1;
  reg [N-1:0] in_2;
  reg [2:0] op_code;
  wire [N-1:0] y;
  wire cf;
  wire zf;

  alu #(N) uut (
    .clk(clk),
    .in_1(in_1),
    .in_2(in_2),
    .op_code(op_code),
    .y(y),
    .cf(cf),
    .zf(zf)
  );

  initial clk = 0;
  always #5 clk = ~clk;

  initial begin
    $dumpfile("alu_waveform.vcd");
    $dumpvars(0, alu_tb);

    in_1 = 8'd10; in_2 = 8'd5;

    op_code = 3'b000; #10;
    op_code = 3'b001; #10;
    op_code = 3'b010; #10;
    op_code = 3'b011; #10;
    op_code = 3'b100; #10;
    op_code = 3'b101; #10;
    op_code = 3'b110; #10;
    op_code = 3'b111; #10;

    in_1 = 8'd15; in_2 = 8'd15; op_code = 3'b001;
     #20;

    $finish;
  end

  initial begin
    $monitor("Time=%0t | in_1=%0d | in_2=%0d | op=%b
     | y=%0d | cf=%b | zf=%b",
             $time, in_1, in_2, op_code, y, cf, zf);
  end
endmodule
```

Listing 5. Verilog Testbench for ALU

III. Waveform Output

IV. Synthesis Results

The synthesized netlists were analyzed using Synopsys Design Compiler to compare area, power, and timing metrics.

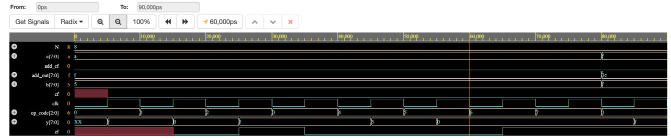

Fig. 3. Output Waveform

A. Discussion

The behavioral ALU used less area because the synthesis tool optimized arithmetic operations efficiently. However, it consumed more dynamic power due to higher switching activity. The RCA-based ALU required more area but consumed less dynamic power, illustrating the area–power trade-off between high-level and structural designs.

TABLE II
SYNTHESIS COMPARISON BETWEEN BEHAVIORAL AND STRUCTURAL ALU

Parameter	**Behavioral**	**RCA-Based**
Total Cell Area (μm^2)	3493.46	6124.65
Dynamic Power (µW)	1106.10	179.61
Leakage Power (nW)	65.37	120.39

V. Conclusion

In this experiment, an 8-bit ALU was successfully designed, verified, and implemented using two distinct Verilog methodologies: a high-level behavioral model and a structural model based on a Ripple Carry Adder. Functional verification confirmed that both designs met the operational specifications.

The subsequent synthesis and comparative analysis using a 180nm standard-cell library revealed a clear and significant **area-power trade-off**. The behavioral model, leveraging synthesis tool optimizations, resulted in a more compact **cell area** ($3493.46\mu m^2$). However, this came at the cost of high **dynamic power** (1106.10µW), likely due to the tool inferring a complex, high-switching-activity adder.

Conversely, the structural RCA-based ALU, while consuming significantly more **area** ($6124.65\mu m^2$), offered substantially lower **dynamic power** (179.61µW). This demonstrates that explicit structural design, while more complex, can grant finer control over the final hardware, leading to a more power-efficient implementation. This experiment successfully illustrates the critical impact of RTL design choices on the final physical metrics of an ASIC design.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.