Experiment 3b
Design, Simulation, and Synthesis of an 8-bit ALU

Kaushik Balaji M S
M.Tech in System-on-chip Design
Indian Institute of Technology, Palakkad
152502010@smail.iitpkd.ac.in

***Abstract*—The objective of this experiment is to design, simulate and synthesize an 8-bit Arithmetic Logic Unit (ALU) using Verilog HDL. Both behavioral and structural architectures are implemented to understand the trade-offs between abstraction levels in digital design. The designs are synthesized using the Synopsys Design Compiler with the SCL 180 nm CMOS standard-cell library. Key synthesis metrics such as area, power, and timing are analyzed and compared. The experiment also includes post-synthesis gate-level simulation to verify functional equivalence between the RTL and synthesized designs.**

I. Introduction

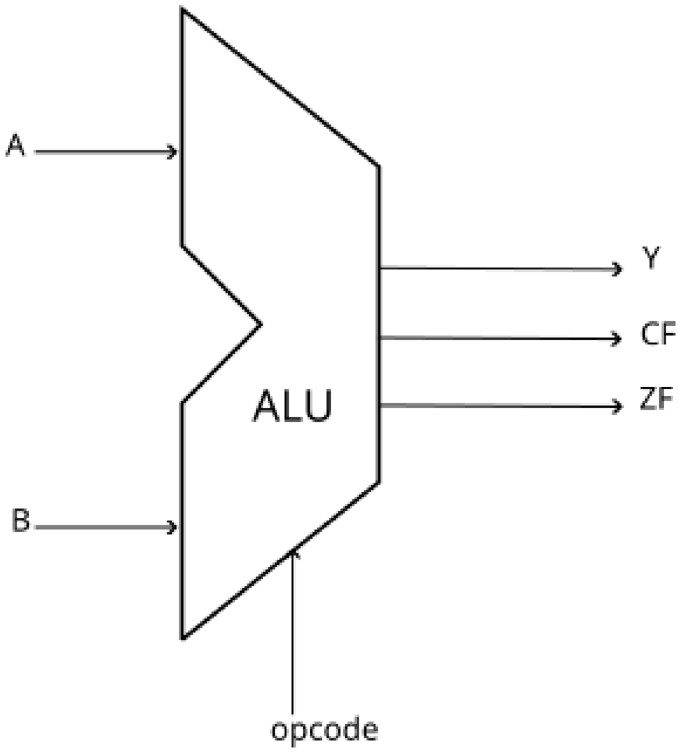


Fig. 1. Simple block diagram of an ALU

An Arithmetic Logic Unit (ALU) is a fundamental component of any digital processing system, performing arithmetic and logic operations on binary operands. It serves as the computational core of microprocessors, digital signal processors (DSPs), and other hardware accelerators.

Typical ALU operations include addition, subtraction, bitwise logical operations (AND, OR, XOR, NOT), and shift operations. Modern ALUs are designed to be parameterized and efficient, allowing them to scale across various word lengths and architectures.

In this experiment, two versions of an ALU are developed and compared:

- *Behavioral ALU* — modeled using Verilog operators ($+.-, \&, |$, etc.) to describe the functionalities.
- *Structural ALU* — implemented using leaf modules such as a Ripple Carry Adder (RCA) for arithmetic operations like addition, subtraction.

Synthesis and timing analysis are performed using the Synopsys Design Compiler to explore how the implementation style affects area, power, and performance.

II. Design Details

Both implementations of the ALU have the same number of input and output ports

- **Inputs**: 8-bit operands A, B, and 3-bit opcode.
- **Outputs**: 8-bit result Y, carry flag CF, and zero flag ZF.

A. 8-bit Behavioral ALU

This design generates the Behavioral implementation of ALU using Verilog operators such as +, -, etc.

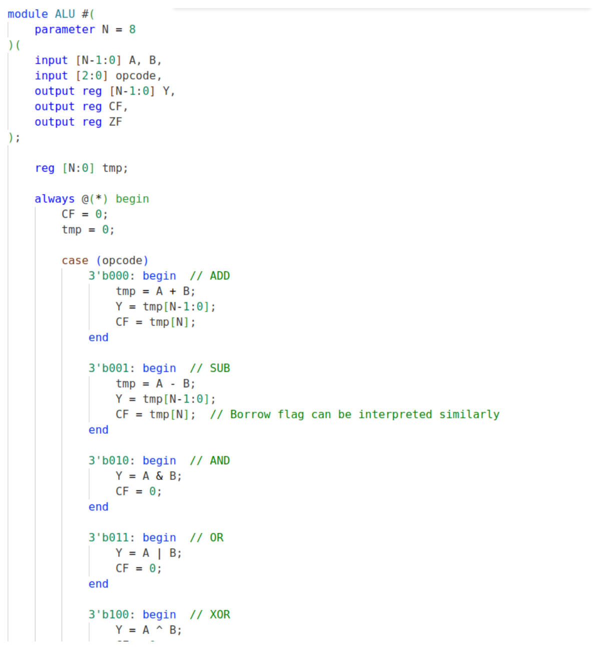
TABLE I
ALU Operation Codes and Descriptions

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

Table 1 gives the various opcodes and their corresponding operations to be performed.

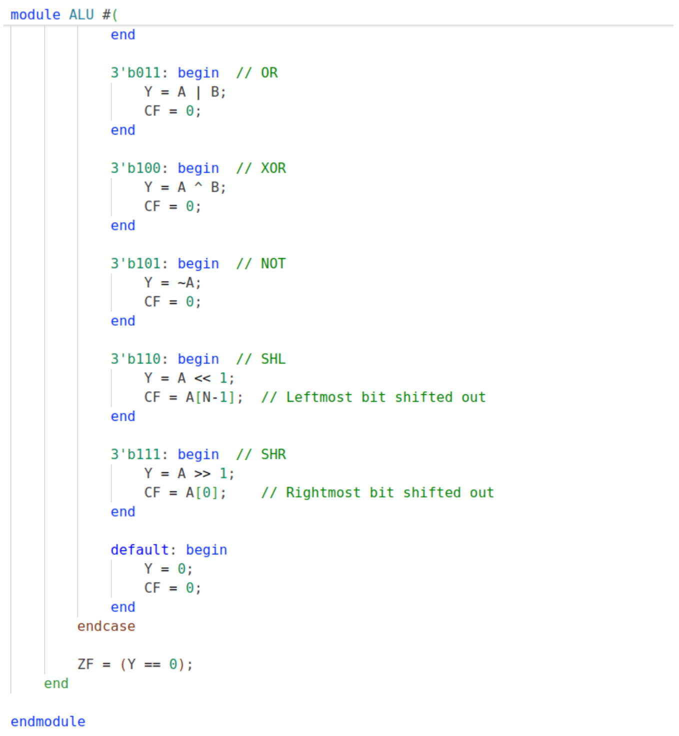
Figures 2, and 3 give the Behavioral Implementation of the ALU design. As can be seen from the figures, no leaf modules or hierarchical design was used in this design.

This version describes the ALU behavior using Verilog arithmetic and logical operators. The synthesis tool internally maps these high-level operations into corresponding logic gates and arithmetic blocks from the library.



```
module ALU #(
    parameter N = 8
)(
    input [N-1:0] A, B,
    input [2:0] opcode,
    output reg [N-1:0] Y,
    output reg CF,
    output reg ZF
);

    reg [N:0] tmp;

    always @(*) begin
        CF = 0;
        tmp = 0;

        case (opcode)
            3'b000: begin  // ADD
                tmp = A + B;
                Y = tmp[N-1:0];
                CF = tmp[N];
            end

            3'b001: begin  // SUB
                tmp = A - B;
                Y = tmp[N-1:0];
                CF = tmp[N];  // Borrow flag can be interpreted similarly
            end

            3'b010: begin  // AND
                Y = A & B;
                CF = 0;
            end

            3'b011: begin  // OR
                Y = A | B;
                CF = 0;
            end

            3'b100: begin  // XOR
                Y = A ^ B;
```

Fig. 2. Behavioral ALU implementation Part - I



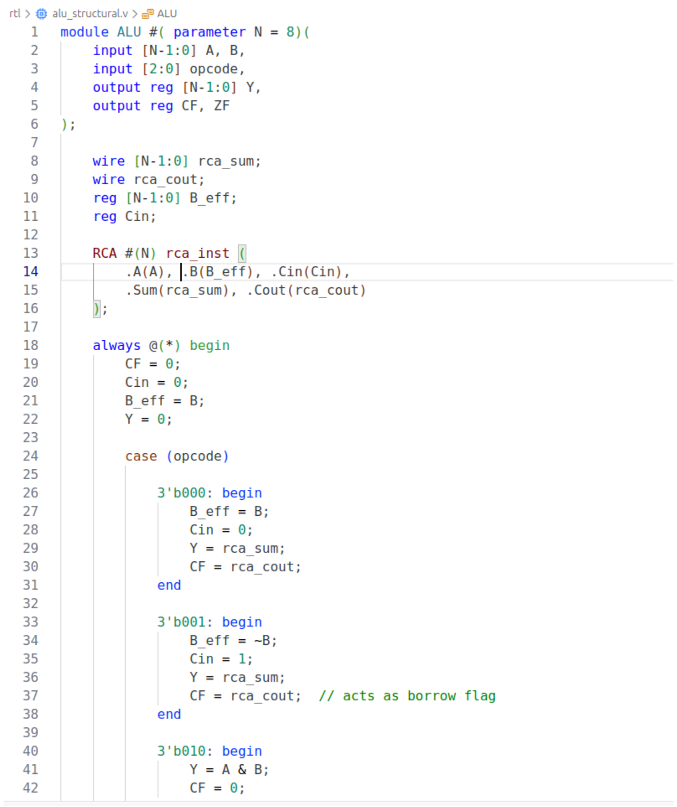
```
module ALU #(
            end

            3'b011: begin  // OR
                Y = A | B;
                CF = 0;
            end

            3'b100: begin  // XOR
                Y = A ^ B;
                CF = 0;
            end

            3'b101: begin  // NOT
                Y = ~A;
                CF = 0;
            end

            3'b110: begin  // SHL
                Y = A << 1;
                CF = A[N-1];  // Leftmost bit shifted out
            end

            3'b111: begin  // SHR
                Y = A >> 1;
                CF = A[0];    // Rightmost bit shifted out
            end

            default: begin
                Y = 0;
                CF = 0;
            end
        endcase

        ZF = (Y == 0);
    end

endmodule
```

Fig. 3. Behavioral ALU implementation Part - II

B. 8-bit Structural ALU with RCA

In this design, arithmetic operations (addition and subtraction) are implemented structurally using a Ripple Carry Adder (RCA) module.

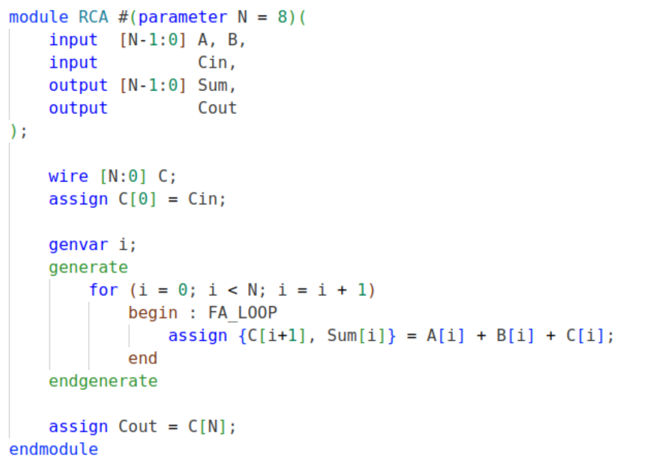
Figures 4 and 7 give the Structural Implementation of the ALU design. As can be seen from the figures, RCA module was used to implement the addition and subtraction functionality.



rtl > alu_structural.v > ALU

```
module ALU #( parameter N = 8)(
    input [N-1:0] A, B,
    input [2:0] opcode,
    output reg [N-1:0] Y,
    output reg CF, ZF
);

    wire [N-1:0] rca_sum;
    wire rca_cout;
    reg [N-1:0] B_eff;
    reg Cin;

    RCA #(N) rca_inst (
        .A(A), .B(B_eff), .Cin(Cin),
        .Sum(rca_sum), .Cout(rca_cout)
    );

    always @(*) begin
        CF = 0;
        Cin = 0;
        B_eff = B;
        Y = 0;

        case (opcode)

            3'b000: begin
                B_eff = B;
                Cin = 0;
                Y = rca_sum;
                CF = rca_cout;
            end

            3'b001: begin
                B_eff = ~B;
                Cin = 1;
                Y = rca_sum;
                CF = rca_cout;  // acts as borrow flag
            end

            3'b010: begin
                Y = A & B;
                CF = 0;
```

Fig. 4. Structural ALU implementation Part - I



```
module RCA #(parameter N = 8)(
    input  [N-1:0] A, B,
    input          Cin,
    output [N-1:0] Sum,
    output         Cout
);

    wire [N:0] C;
    assign C[0] = Cin;

    genvar i;
    generate
        for (i = 0; i < N; i = i + 1)
            begin : FA_LOOP
                assign {C[i+1], Sum[i]} = A[i] + B[i] + C[i];
            end
    endgenerate

    assign Cout = C[N];
endmodule
```

Fig. 5. Ripple Carry Adder (RCA) implementation

For addition, the inputs are given as $A = A, B = B, C_{\text{in}} = 0$ for addition. And in case of subtraction, they are modified as $A = A, B = -B, C_{\text{in}} = 1$. Inverting the bits of B, and having Carry-in bit as 1, means that we are actually doing addition of A and 2's complement of B, which is same as A-B

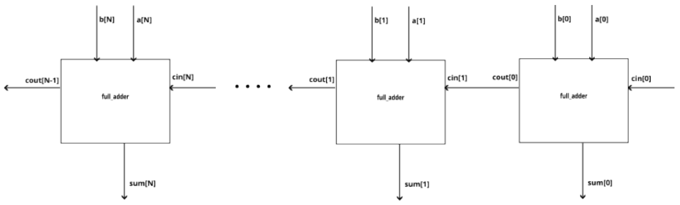
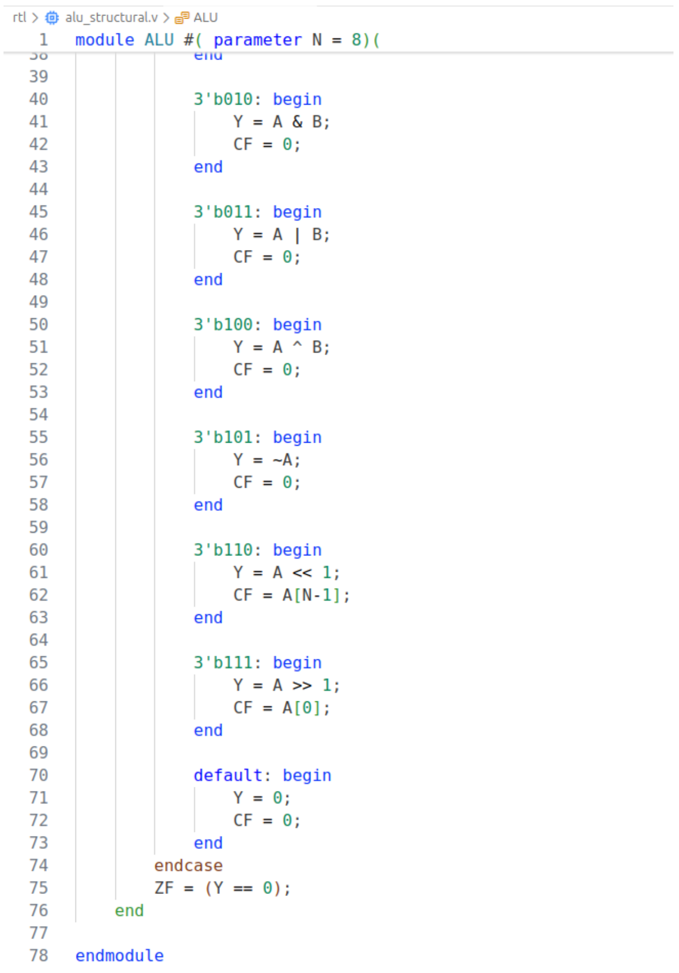


Fig. 6. Ripple Carry Adder (RCA) Block diagram



```
rtl > alu_structural.v > ALU
module ALU #( parameter N = 8)(
            end

            3'b010: begin
                Y = A & B;
                CF = 0;
            end

            3'b011: begin
                Y = A | B;
                CF = 0;
            end

            3'b100: begin
                Y = A ^ B;
                CF = 0;
            end

            3'b101: begin
                Y = ~A;
                CF = 0;
            end

            3'b110: begin
                Y = A << 1;
                CF = A[N-1];
            end

            3'b111: begin
                Y = A >> 1;
                CF = A[0];
            end

            default: begin
                Y = 0;
                CF = 0;
            end
        endcase
        ZF = (Y == 0);
    end

endmodule
```

Fig. 7. Structural ALU implementation Part - II

III. Experimental Procedure

A. Simulation

The simulation of the above 2 RTL designs are done with the testbench from Figure 10.

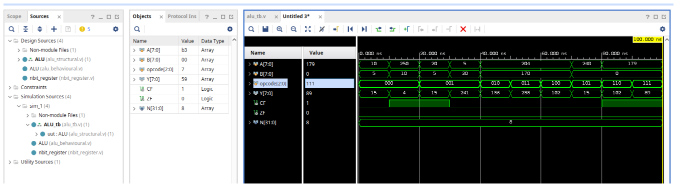


Fig. 8. Waveform of Behavioral ALU

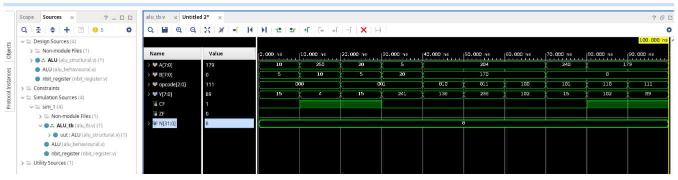
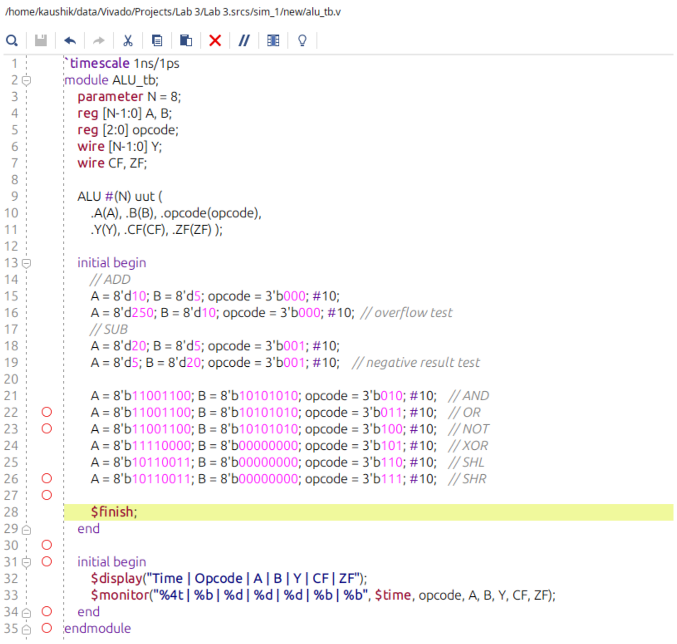


Fig. 9. Waveform of Structural ALU



```
/home/kaushik/data/Vivado/Projects/Lab 3/Lab 3.srcs/sim_1/new/alu_tb.v

`timescale 1ns/1ps
module ALU_tb;
  parameter N = 8;
  reg [N-1:0] A, B;
  reg [2:0] opcode;
  wire [N-1:0] Y;
  wire CF, ZF;

  ALU #(N) uut (
    .A(A), .B(B), .opcode(opcode),
    .Y(Y), .CF(CF), .ZF(ZF) );

  initial begin
    // ADD
    A = 8'd10; B = 8'd5; opcode = 3'b000; #10;
    A = 8'd250; B = 8'd10; opcode = 3'b000; #10; // overflow test
    // SUB
    A = 8'd20; B = 8'd5; opcode = 3'b001; #10;
    A = 8'd5; B = 8'd20; opcode = 3'b001; #10;   // negative result test

    A = 8'b11001100; B = 8'b10101010; opcode = 3'b010; #10;   // AND
    A = 8'b11001100; B = 8'b10101010; opcode = 3'b011; #10;   // OR
    A = 8'b11001100; B = 8'b10101010; opcode = 3'b100; #10;   // NOT
    A = 8'b11110000; B = 8'b00000000; opcode = 3'b101; #10;   // XOR
    A = 8'b10110011; B = 8'b00000000; opcode = 3'b110; #10;   // SHL
    A = 8'b10110011; B = 8'b00000000; opcode = 3'b111; #10;   // SHR

    $finish;
  end

  initial begin
    $display("Time | Opcode | A | B | Y | CF | ZF");
    $monitor("%4t | %b | %d | %d | %d | %b | %b", $time, opcode, A, B, Y, CF, ZF);
  end
endmodule
```

Fig. 10. Testbench for the simulation of RTL designs

As can be observed from the Figures 8, and 9, the waveforms are similar for both the implementations for the same testbench. The panel on the left shows which module is chosen as the 'Top' module while running the simulation.

B. Synthesis

Both the ALU designs were synthesized and their respective Gate-level netlist mapped code, reports on Timing, area and power were obtained.

*1) **Behavioral ALU:*** Table 2 shows the variation of Slack, Power, and Total Area obtained by modifying the Clock period during the synthesis of the Behavioral ALU design using Synopsys Design Compiler. The results highlight how the design behaves under different timing constraints.

From the table, it is observed that the slack value is negative for small clock periods (for example, at Clock=1ns, Slack = –2.44 ns). This indicates that the design fails to meet timing requirements under such aggressive constraints. As the clock period is gradually increased, the slack also increases and eventually becomes positive at around 4 ns, indicating that the design meets timing from this point onward. Thus, the minimum achievable clock period (or the critical path delay) for this ALU design is approximately 4 ns, corresponding to a maximum clock frequency of 250 MHz.

TABLE II
BEHAVIORAL ALU SYNTHESIS RESULTS FOR VARYING CLOCK PERIODS

Clock (ns)	Slack (ns)	Power (mW)	Total Area (μm^2)
1	-2.44	5.0077	13205.1277
2	-1.50	2.5146	13542.8089
3	-0.56	1.7084	13303.1603
4	0.00	1.1199	10973.0805
5	-0.15	0.8290	10074.5103
6	0.01	0.6492	9494.5512
7	0.25	0.5072	8694.3898
8	1.25	0.4438	8694.3898
9	2.25	0.3945	8694.3898
10	3.25	0.3551	8694.3898
11	4.25	0.3228	8694.3898
12	5.25	0.2959	8694.3898
13	6.25	0.2732	8694.3898
14	7.25	0.2537	8694.3898
15	8.25	0.2368	8694.3898

The power consumption shows an inverse dependence on the clock period. As the clock period increases, the total power decreases sharply — from 5.00 mW at 1ns to about 0.23 mW at 15ns. This trend confirms that the dynamic power is primarily dependent on switching activity, which inturn is dependent on the operating clock's frequency. Thus,

$$P_{\text{dynamic}} \propto \frac{1}{T_{\text{clk}}} \quad \text{or equivalently} \quad P_{\text{dynamic}} \propto f_{\text{clk}}$$

The total area initially fluctuates slightly between 13,000 µm² and 10,000 µm² for smaller clock periods, but stabilizes at around 8694 µm² for clock periods beyond 7 ns. This indicates that the synthesis tool performs some degree of cell resizing and restructuring under tighter constraints to improve timing, but once timing is comfortably met, the area becomes constant, suggesting no further optimization is needed.

*2) **Structural ALU:*** Table 3 gives the values of slack, power and Total area obtained by varying the Clock period from 1 to 20 ns.

From the table, it is evident that for Clock = 1 ns, the Slack = –3.12 ns, meaning the design fails to meet timing under tight timing constraints. As the clock period is relaxed gradually, the slack value improves and becomes positive at around 4 ns, which implies that the design starts meeting timing constraints after this point. Hence, the minimum achievable clock period (or the critical path delay) for the RCA-based ALU is approximately 4 ns, corresponding to a maximum operating frequency of 250 MHz (same as that of Behavioral ALU design).

The power consumption exhibits a strong inverse relationship with the clock period. It decreases significantly from 4.51 mW at 1 ns to about 0.19 mW at 20 ns. This clearly demonstrates that dynamic power dominates total power dissipation, as it depends directly on the switching frequency:

$$P_{\text{dynamic}} \propto \frac{1}{T_{\text{clk}}} \quad \text{or equivalently} \quad P_{\text{dynamic}} \propto f_{\text{clk}}$$

The area initially varies slightly in the range of 11,000–12,000 µm² for lower clock periods (tight constraints),

TABLE III
STRUCTURAL ALU SYNTHESIS RESULTS FOR VARYING CLOCK PERIODS

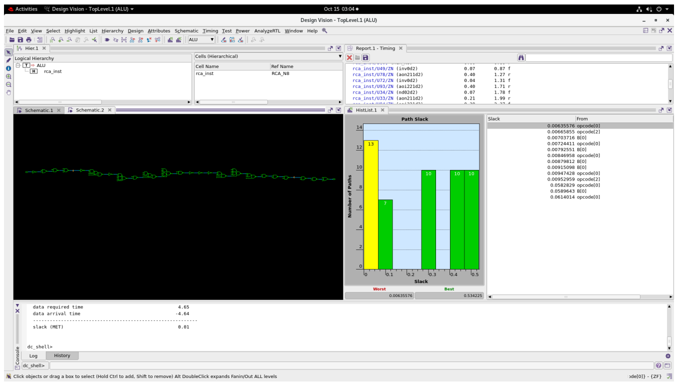
Clock (ns)	Slack (ns)	Power (mW)	Area (μm^2)
1	-3.12	4.5059	11431.71532
2	-2.18	2.3200	11916.14881
3	-0.82	1.5656	11785.08358
4	0.00	1.1115	11305.96095
5	0.01	0.8555	10557.36371
6	0.03	0.6339	9584.435581
7	0.82	0.5356	9499.621078
8	1.82	0.4687	9499.621078
9	2.82	0.4166	9499.621078
10	3.82	0.3750	9499.621078
11	4.82	0.3409	9499.621078
12	5.82	0.3125	9499.621078
13	6.82	0.2885	9499.621078
14	7.82	0.2679	9499.621078
15	8.82	0.2501	9499.621078
16	9.82	0.2344	9499.621078
17	10.82	0.2207	9499.621078
18	11.82	0.2084	9499.621078
19	12.82	0.1974	9499.621078
20	13.82	0.1876	9499.621078

then stabilizes at 9499 µm² once the timing constraint is sufficiently relaxed. This stabilization indicates that the synthesis tool initially performs cell upsizing and gate restructuring to meet stringent timing goals, but once timing is satisfied, it converges to a minimal-area configuration.

C. Critical path analysis

For the Behavioral ALU, Critical path is through the adder/subtractor datapath synthesized into a carry-select style adder internally.

In case of Structural ALU, Critical path spans the carry propagation chain across all 8 full-adders. The delay increases linearly with bit-width, leading to timing violations under tight constraints.



```
Cell Name   Ref Name
rca_inst    RCA_N8
```

```
rca_inst/U49/ZN (inv0d2)         0.07    0.87 f
rca_inst/U78/ZN (aon211d2)       0.40    1.27 r
rca_inst/U72/ZN (inv0d2)         0.04    1.31 f
rca_inst/U93/ZN (aoi221d2)       0.40    1.71 r
rca_inst/U34/ZN (nd02d2)         0.07    1.78 f
rca_inst/U33/ZN (aon211d2)       0.21    1.99 r
```

Slack	From
0.00635576	opcode[0]
0.00665855	opcode[2]
0.00703716	B[0]
0.00724411	opcode[0]
0.00792551	B[0]
0.00846958	opcode[0]
0.00879812	B[0]
0.00915098	B[0]
0.00947428	opcode[0]
0.00952959	opcode[2]
0.0582829	opcode[0]
0.0589643	B[0]
0.0614014	opcode[0]

Worst 0.00635576 Best 0.534225

```
  data required time                         4.65
  data arrival time                         -4.64
  -----------------------------------------------------
  slack (MET)                                0.01


dc_shell>
```

Click objects or drag a box to select (Hold Ctrl to add, Shift to remove) Alt DoubleClick expands Fanin/Out ALL levels

Fig. 11. Critical path (as shown by DC compiler) of the Structural ALU

D. Graphical representation

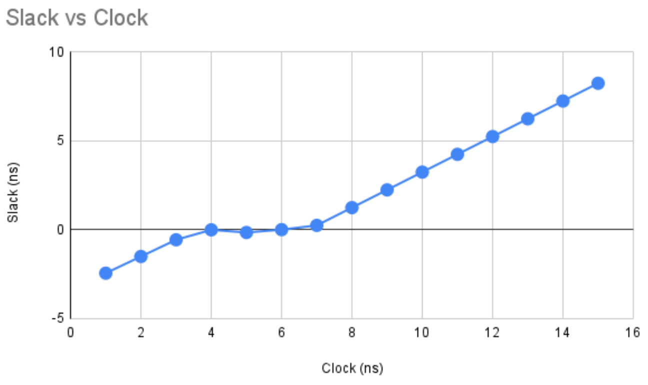


Fig. 12. Slack variation with change in Clock period for Behavioral ALU

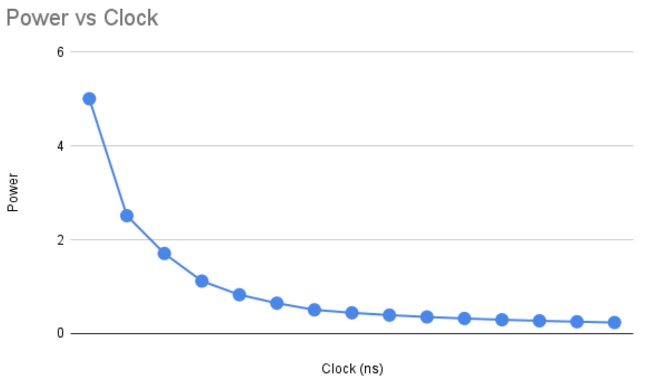


Fig. 13. Power variation with change in Clock period for Behavioral ALU

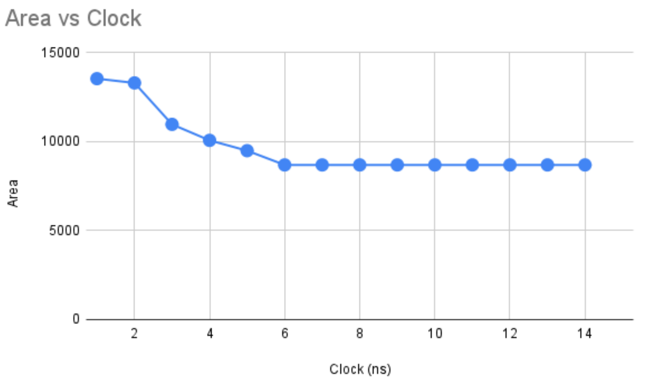


Fig. 14. Area variation with change in Clock period for Behavioral ALU

*1) **Behavioral ALU:*** As can be seen from Figures 12, 13, 14, the graphs correlate with the inferences made in the above section III-B.

*2) **Structural ALU:*** As can be seen from Figures 15, 16, 17, the graphs correlate with the inferences made in the above section III-B.

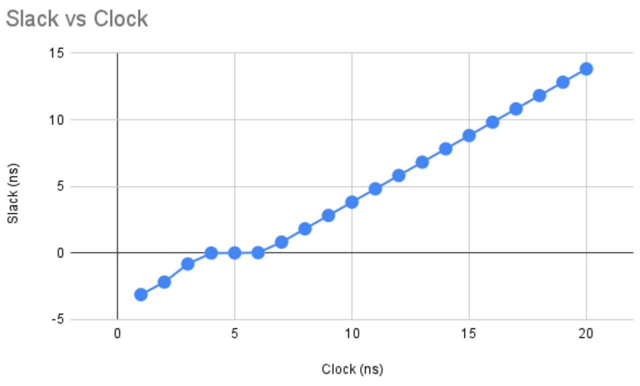


Fig. 15. Slack variation with change in Clock period for Structural ALU

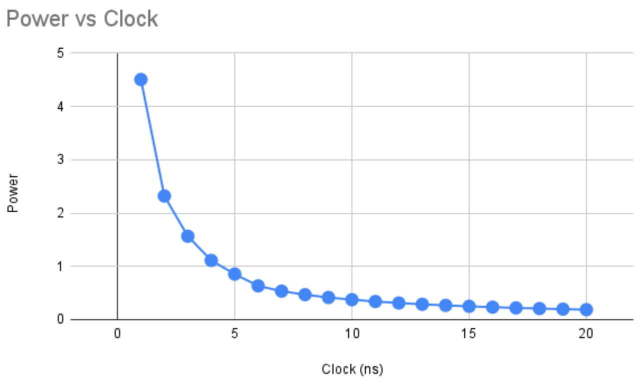


Fig. 16. Power variation with change in Clock period for Structural ALU

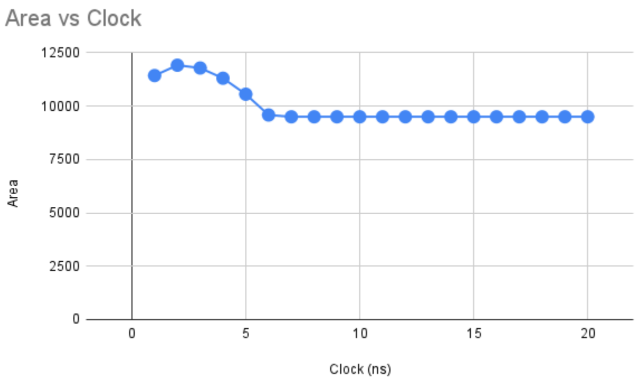


Fig. 17. Area variation with change in Clock period for Structural ALU

E. Gate level simulation

One of the outputs of the synthesis of the RTL design was the veriolg file of the RTL design converted to their gate level implementation. This is called the Gate-level netlist.

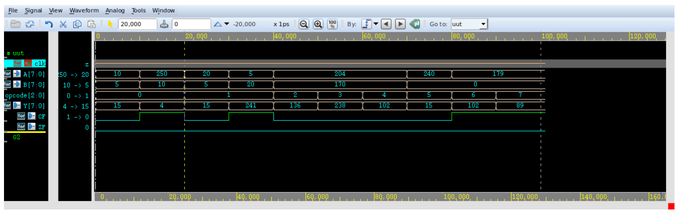


Fig. 18. Gate-level simulation of Behavioral ALU netlist

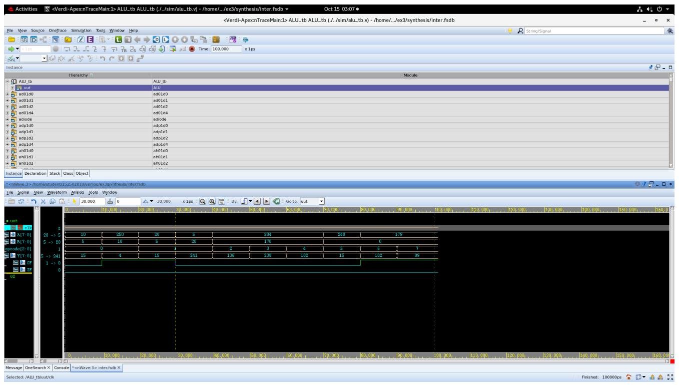


Fig. 19. Gate-level netlist for Structural ALU netlist

Figure 18, 19 gives the gate level netlist simulation of both the Behavioral and Structural ALU designs using the testbench mentioned in Figure 10. The gate-level simulation gave the expected output as the RTL simulation.

Conclusion

The experiment successfully demonstrated the simulation and synthesis of 8-bit ALU using Behavioral and Hierarchical/Structural design methods. The synthesis of these designs were done, their reports analyzed and the gate-level netlist was also simulated to verify the correctness of the synthesis.