Experiment 3a
Introduction to ASIC Synthesis

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 familiarize with ASIC synthesis using Synopsys Design Compiler and to analyze the area, power, and timing reports of those synthesized designs. The experiment involves synthesizing different Verilog designs — a 4-bit Fibonacci sequence generator, a 4-bit Fibonacci generator with ripple carry adder, and a 32-bit Fibonacci generator — using the 180 nm CMOS standard cell library from the Semi-Conductor Laboratory (SCL). The synthesis reports are generated and analyzed to understand the impact of clock timing constraints on area and power consumption, and to identify the critical paths that limit achievable frequency.**

I. Introduction

After the RTL design of the system is done, then synthesis of that RTL design is made to get the gate-level netlist for that design, thus making synthesis as a bridge between RTL design and the Gate-level hardware implementation. The synthesis process converts the Verilog RTL into a gate-level netlist composed of logic gates from a standard cell library.

This mapping enables estimation of hardware metrics such as:

- Area — total area taken for the logic implementation on a silicon wafer.
- Power — dynamic and leakage power consumed during operation.
- Timing — propagation delays and achievable clock frequency.

The Synopsys Design Compiler (DC) is used for logic synthesis. It reads RTL, Standard Cell Library (SCL), and constraint files to produce an optimized gate-level netlist that meets the desired timing goals. In this experiment, we explore the synthesis for three variants of the Fibonacci sequence generator and study how the synthesis results vary with different timing constraints.

II. Design Details

A. *4-bit Fibonacci generator without RCA*

This design generates the Fibonacci sequence iteratively. It does not use any modular or hierarchical style of design implementation for combinational logic.

On every clock cycle, the next Fibonacci number is generated.

```
module fibo_nonmodular #(parameter N = 4)(
    input clk, input reset,
    output [3:0] reg1 );

    wire [3:0] reg0;
    wire [3:0] next_reg0, next_reg1;
    assign next_reg0 = reg0 + reg1;
    assign next_reg1 = reg0;

    nbit_register #(4) reg0_inst (
        .clk(clk), .reset(reset), .d(next_reg0),
        .reset_value({{(N-1){1'b0}}, 1'b1}), .q(reg0)
    );

    nbit_register #(4) reg1_inst (
        .clk(clk), .reset(reset), .d(next_reg1),
        .reset_value({N{1'b0}}), .q(reg1)
    );

endmodule
```

Fig. 1. Implementation of Fibonacci Generator

B. *4-bit Fibonacci Generator with RCA*

This version explicitly instantiates a 4-bit ripple carry adder instead of using a behavioral '+' operator. The RCA introduces additional gate-level delays due to propagation through the adder chain, leading to a critical path that is longer than the previous design without any hierarchical structure. The figures 2, 3, 5 gives the implementation of the Fibonacci generator with RCA, RCA with Full adders and Full adder respectively.

```
module fibo #(parameter N=4)(input clk, input reset, output [N-1:0] fib_out);
  wire [N-1:0] a, b;
  wire [N-1:0] add_out;
  wire [N-1:0] next_a, next_b;
  wire cout_output;

  ripple_adder #(N) RA (.a(a), .b(b), .cin(1'b0), .sum(add_out), .cout(cout_output));

  assign next_a = b;
  assign next_b = add_out;

  nbit_register #(N) regA (.clk(clk), .reset(reset), .reset_value({N{1'b0}}), .d(next_a), .q(a));
  nbit_register #(N) regB (.clk(clk), .reset(reset), .reset_value({{(N-1){1'b0}}, 1'b1}), .d(next_b), .q(b));

  assign fib_out = a;
endmodule
```

Fig. 2. Implementation of Fibonacci Generator with RCA

C. *32-bit Fibonacci Generator*

Modifying the Parameter value to $N = 32$ in Fig. 1 gives the 32-bit implementation of the Fibonacci generator without

```
ripple_adder.v > ripple_adder
1  module ripple_adder #(parameter N= 32) (
2    input [N-1:0] a, input [N-1:0] b, input cin,
3    output [N-1:0] sum, output cout);
4
5    wire [N:0] c;
6    assign c[0] = cin;
7
8    genvar i;
9    generate
10     for (i = 0; i < N; i = i+1) begin: FA_LOOP
11       full_adder fa_instance1(.a(a[i]), .b(b[i]), .cin(c[i]), .sum(sum[i]), .cout(c[i+1]));
12     end
13   endgenerate
14
15   assign cout = c[N];
16 endmodule
```

Fig. 3. Implementation of RCA using Full Adders

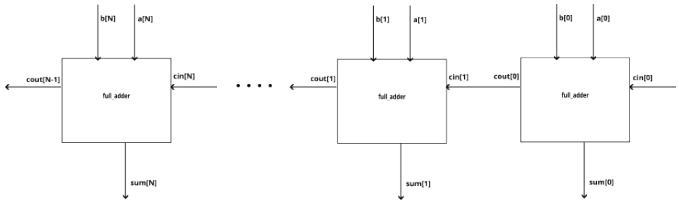

Fig. 4. Block diagram of RCA using N Full Adders

```
full_adder.v > ...
1  module full_adder(input a, input b, input cin, output sum, output cout);
2    //assign {cout, sum} = a + b + cin;
3    assign sum  = a ^ b ^ cin;
4    assign cout = (a & b) | (b & cin) | (a & cin);
5  endmodule
6
```

Fig. 5. Implementation of Full Adder

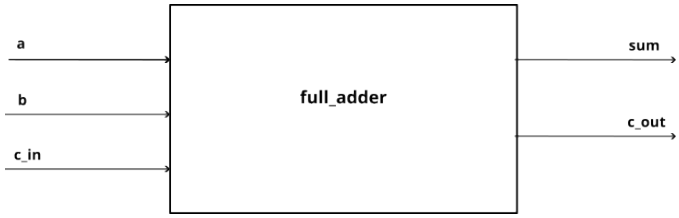

Fig. 6. Block diagram of a Full Adder

any modules. Similarly, modifying it in the code of Fig. **??** gives the 32-bit Fibonacci implementation using RCA.

III. Experimental Procedure

The synthesis of the above RTL designs is done with the RTL design, Standard cell library, and the Constraints file as the input to the Synopsys DC compiler.

A. Synthesis

Synthesis was done for 4-bit and 32-bit Fibonacci generators (with and without RCA) and their reports on time, area, and power were taken for analysis.

1) ***4-bit Fibonacci without RCA****:* Reports were generated and the necessary data from those reports were taken for further analysis.

Table 1 shows the values of Slack, Power, and Total Area, gathered by modifying the Clock period during the synthesis of the design.

From this table, it can be seen that the value of slack is negative for $Clock = 1ns$. The design did not meet the timing

TABLE I
Timing, Power, and Area Analysis for 4-bit Fibonacci Behavioral Design

Clock (ns)	Slack (ns)	Power (mW)	Area (μm^2)
1	-0.76	0.8406	1016.243643
2	0.02	0.4203	1016.243643
3	0.63	0.2802	1016.243643
4	1.31	0.2102	1016.243643
5	2.22	0.1681	1016.243643
6	3.31	0.1402	1016.243643
7	4.54	0.1201	1016.243643
8	5.98	0.1051	1016.243643
9	6.68	0.0934	1016.243643
10	7.31	0.0841	1016.243643

constraints in this clock period. Then when the Clock period was relaxed little by little, the slack increases, thus becoming positive after $Clock = 2ns$, thus making this the minimum possible attainable clock period, so the maximum attainable clock frequency for 4-bit Fibonacci without RCA becomes *1 GHz*

The total power consumption decreases sharply as the clock period increases — from 0.8406 mW at 1 ns to about 0.0841 mW at 10 ns. This strong inverse relationship indicates that the dynamic power is dominated by switching activity, which scales directly with operating frequency. Thus,

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

The total area remains constant at approximately 1016.24 µm² across all clock periods. This shows that the synthesis tool did not need to upsize or restructure logic to meet timing — the existing cell configuration was sufficient even for the tightest constraint. Hence, the design is area-efficient and structurally stable across all timing scenarios.

2) ***4-bit Fibonacci with RCA****:* Table 2 gives the values of slack, power and Total area obtained by varying the Clock period from 1 to 15 ns.

TABLE II
Timing, Power, and Area Analysis for Hierarchical design

Time (ns)	Slack (ns)	Total Power (µW)	Total Area (µm²)
1	0.13	1976.6	3465.42
2	1.05	1020.4	3501.12
3	1.98	550.2	3531.03
4	2.95	412.8	3530.85
5	3.92	340.7	3531.77
6	4.95	290.9	3531.21
7	5.91	246.4	3531.16
8	6.98	206.33	3531.03
9	7.99	184.7	3530.72
10	8.98	163.9	3531.05
11	9.97	148.4	3531.15
12	10.98	137.55	3531.03
13	11.96	124.8	3531.26
14	12.97	116.3	3531.04
15	13.98	110.04	3531.03

From this table, it can be seen that the Slack is already positive at Clock = 1 ns, indicating that the design is able

to meet the timing constraints even at the highest operating frequency tested. As the clock period is relaxed (i.e., frequency decreases), the slack value gradually increases, showing that the design easily satisfies setup requirements under slower clocks.

Since the timing constraint is met even at Clock = 1 ns, this can be taken as the minimum possible achievable clock period, corresponding to a maximum operating frequency of approximately 1 GHz for the given design (4-bit Fibonacci without RCA).

The Total power consumption decreases sharply as the clock period increases — from about 1976.6 µW at 1 ns to only 110 µW at 15 ns. This happens because at lower frequencies, the switching activity per unit time is reduced, and dynamic power (which depends on frequency) decreases accordingly. Hence,

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

The total area remains nearly constant (3531 µm²) across all clock periods, with only minor variations. This shows that the synthesis tool did not need to perform gate resizing or add buffers for timing optimization — since the design already met timing comfortably even at 1 ns. Thus, the design is area-stable and well optimized.

B. Critical path analysis

For both designs, the relevant synchronous timing path is:

Register Q (on clock edge at cycle n) → combinational logic → D input of one or more registers (setup before next clock edge at cycle n+1).

So the critical path = register-output drive + combinational logic delay (adder, routing, buffers, any combinational assigns) + register input network + setup time of destination flop. This is the theoretical critical path, that can be seen from the RTL design code.

From the timing report, the worst case path will show the startpoint as reg1 Q pin, then a chain of gate level implementation of the '+' sign that was used in case of the Behavioural design and the Ripple-carry adder in case of the Hierarchical design, then endpoint = flop D pin for reg0. Path delay will be dominated by the sum of those adder cell delays.

As no specific adder module was specified in the behavioural design, the best adder module that can satisfy the timing constraints, and has optimized performance and area, will be chosen during the synthesis process.

C. Graphical representation

As can be seen graph in Figure 7 , the Slack varies with Clock period almost linearly. And the Slack becomes nearly 0, at clock period 2ns.

The Fig. 8, shows how Power varies inversely with the Clock period.

Area remains the same for various values of Clock period.

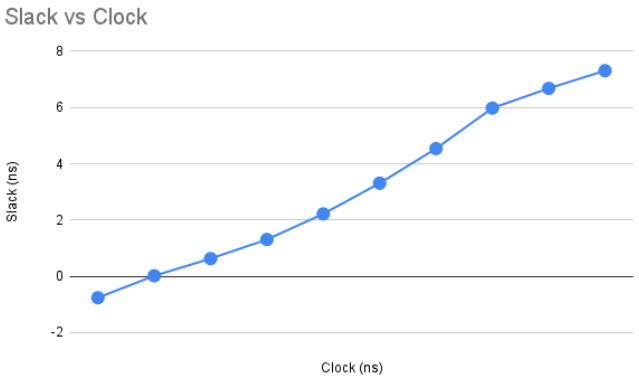

Fig. 7. Slack variation with change in Clock period for 4-bit Behavioral design

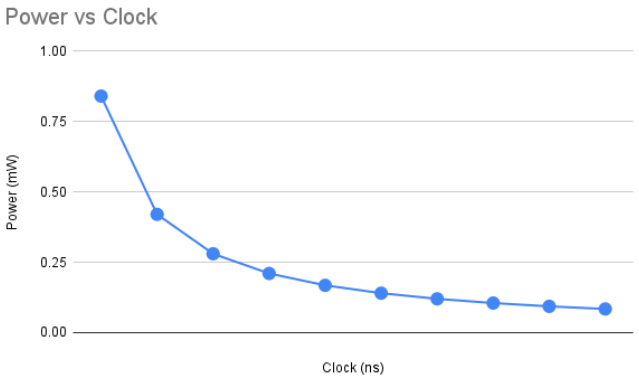

Fig. 8. Power variation with change in Clock period for 4-bit Behavioral design

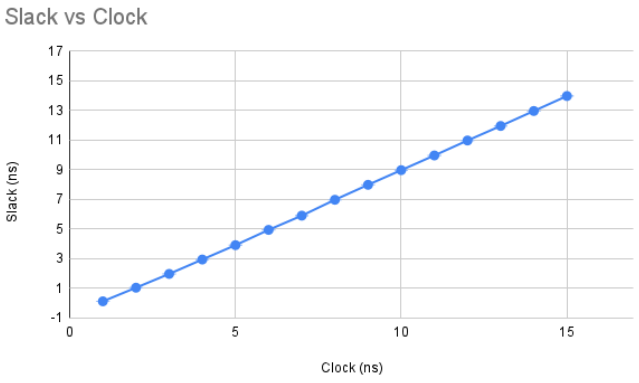

Fig. 9. Slack variation with change in Clock period for 4-bit Hierarchical design

Figures 9, 10, 11 shows the variations of Slack, Power, and Area with clock period. As can be seen, the Slack, power, and area vary in the same way as it did for the Behavioral design.

The same graphical interpretation can be made for the 32-bit implementation of Fibonacci generator. The area increases due

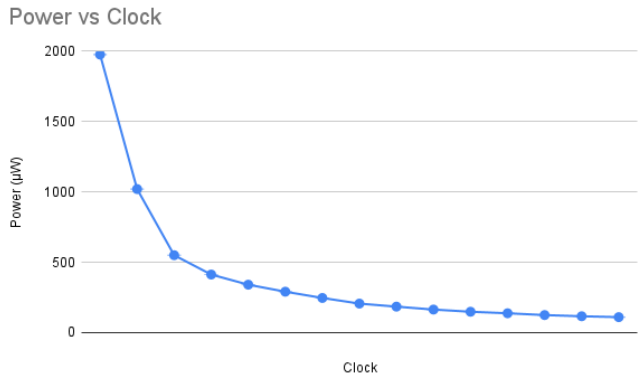

Fig. 10. Power variation with change in Clock period for 4-bit Hierarchical design

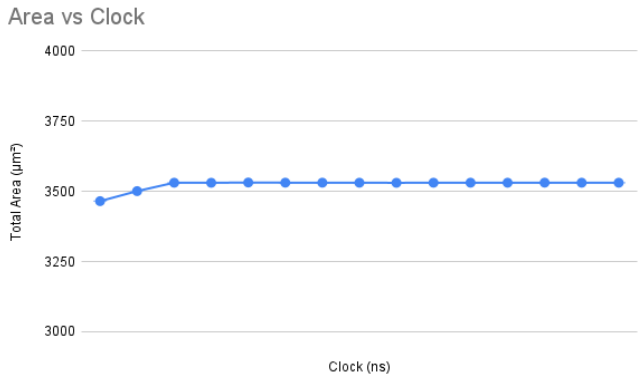

Fig. 11. Area variation with change in Clock period for 4-bit Hierarchical design

to the increase in number of flip flops that will be implemented in the register and the size of the adder also increases due to increase in number of bits. As more bits are used, the power needed to drive these loads will also increase accordingly. So Area and power vary in the same way it does for 4-bit implementation, but has higher value.

D. 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.

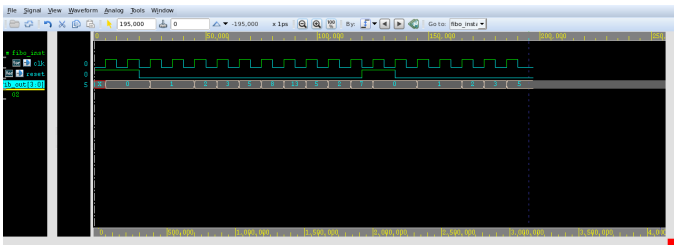

Fig. 12. Gate-level simulation of 4-bits Hierarchical Fibonacci design

```
////////////////////////////////////////////////////////////
// Created by: Synopsys DC Expert(TM) in wire load mode
// Version   : U-2022.12-SP5
// Date      : Wed Oct  8 03:20:24 2025
////////////////////////////////////////////////////////////
`timescale 1ns/1ps

module fibo ( clk, reset, fib_out );
  output [3:0] fib_out;
  input clk, reset;
  wire   n5, n13, n15, n16, n17, n18, n19, n20, n21, n22, n23, n24, n25, n26,
         n27, n28, n29, n30, n31;
  wire   [3:0] a;
  wire   [3:0] add_out;

  dfcrq1 \a_reg[3]  ( .D(add_out[3]), .CP(clk), .CDN(n31), .Q(a[3]) );
  dfcrq1 \fib_out_reg[3]  ( .D(a[3]), .CP(clk), .CDN(n31), .Q(fib_out[3]) );
  dfcrq4 \a_reg[1]  ( .D(n5), .CP(clk), .CDN(n31), .Q(a[1]) );
  dfcrq4 \a_reg[2]  ( .D(add_out[2]), .CP(clk), .CDN(n31), .Q(a[2]) );
  dfcrq2 \fib_out_reg[0]  ( .D(a[0]), .CP(clk), .CDN(n31), .Q(fib_out[0]) );
  dfprb2 \a_reg[0]  ( .D(add_out[0]), .CP(clk), .SDN(n31), .Q(a[0]), .QN(n17)
       );
  dfcrb2 \fib_out_reg[1]  ( .D(a[1]), .CP(clk), .CDN(n31), .Q(fib_out[1]),
    .QN(n19) );
  dfcrb4 \fib_out_reg[2]  ( .D(a[2]), .CP(clk), .CDN(n31), .Q(fib_out[2]),
    .QN(n15) );
  invbd2 U18 ( .I(n27), .ZN(n24) );
  xr02d1 U19 ( .A1(fib_out[0]), .A2(a[0]), .Z(add_out[0]) );
  xr03d1 U20 ( .A1(n30), .A2(fib_out[3]), .A3(n29), .Z(add_out[3]) );
  an02d4 U21 ( .A1(n18), .A2(fib_out[0]), .Z(n13) );
  aoi222d2 U22 ( .A1(n28), .A2(n27), .B1(n26), .B2(n25), .C1(a[2]), .C2(
    fib_out[2]), .ZN(n30) );
  inv0d2 U23 ( .I(n13), .ZN(n16) );
  nr02d2 U24 ( .A1(n24), .A2(n16), .ZN(n26) );
  inv0d1 U25 ( .I(n17), .ZN(n18) );
  aoi21d2 U26 ( .B1(n13), .B2(n25), .A(n20), .ZN(n21) );
  xr03d1 U27 ( .A1(n15), .A2(a[2]), .A3(n21), .Z(add_out[2]) );
  invbd2 U28 ( .I(a[2]), .ZN(n23) );
  nd02d2 U29 ( .A1(fib_out[1]), .A2(a[1]), .ZN(n22) );
  nd02d4 U30 ( .A1(n15), .A2(n23), .ZN(n27) );
  nd12d2 U31 ( .A1(a[1]), .A2(n19), .ZN(n25) );
  invbd7 U32 ( .I(reset), .ZN(n31) );
  inv0d1 U33 ( .I(n22), .ZN(n28) );
  inv0d1 U34 ( .I(n22), .ZN(n20) );
  inv0d1 U35 ( .I(a[3]), .ZN(n29) );
  xr03d1 U36 ( .A1(fib_out[1]), .A2(a[1]), .A3(n13), .Z(n5) );
endmodule
```

Fig. 13. Gate-level netlist for 4-bit Hierarchical Fibonacci design

Figure 13 is the Gate-level netlist generated by Synopsys during the synthesis of 4-bit Hierarchical Fibonacci design.

As can be seen, all the RTL level codes written in Verilog in Fig. **??** were converted into their gate level implementation by the DC compiler.

Fig. 12 gives the Simulation of the Gate-level netlist generated during synthesis. The simulation was performed using the same testbench that was used for the RTL simulation. The gate-level simulation gave the expected output as the RTL simulation.

CONCLUSION

The experiment successfully demonstrated the ASIC synthesis process using Synopsys DC and SCL 180 nm libraries. The critical path in both Fibonacci designs was taken from the obtained schema, and the timing, power, and area reports for different clock periods were analyzed. The simulation of the generated netlist was also simulated, and the waveform was verified.