EE5011 VLSI Design Lab Report: Experiment 1

Subham Ball
Dept. of System of Chip Design
IIT Palakkad
152202017@smail.iitpkd.ac.in

***Abstract*—Experiment 1's objective is to use Verilog to create a Fibonacci Series Generator. It assists us in becoming more skilled in executing Hierarchical design in Verilog and familiarising ourselves with Verilog Simulation.**

I. INTRODUCTION

The fundamental hierarchical unit in Verilog is referred to as a module. A module name along with an interface containing the port list together define a module. It is possible to create a hierarchical module, indicating that a module can incorporate other modules by instantiating other, lower-level modules. By name or position, ports are connected to nets.

Continuous assignments that make use of "assign" statements and always update the variable in the LHS whenever the RHS expression changes are the signatures of combinatorial logic. The assignment's left side may be a net, whereas the right side may be a register or net. A register is a variable in Verilog that can hold a value as opposed to a net, which is continuously driven and unable to do so.

Memory data types can be updated by procedural assignments that use 'initial' and 'always'statements. Every time a signal's value changes in the sensitivity list, the "always" block executes once. Any assignment made in the "always" block has to be declared as a register. Procedural assignments are of two types: blocking and non-blocking. Non-blocking statements must be employed in the "always" block to implement sequential logic.

Every time there is a positive or negative edge in the clock pulse, a 4-bit Fibonacci series generator generates a term from the Fibonacci series. A positive or negative edge reset when applied causes it to be initialized. Designing a Fibonacci Series Generator, as seen in Fig. 1, and using a testbench to verify this design is the desired target of Experiment 1.

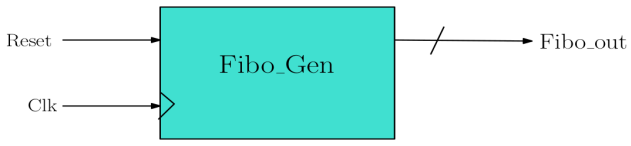

Fig. 1. Block Diagram

II. FIBONACCI GENERATOR WITH RESET, AN ADDER, AND TWO REGISTERS:

A. Implementation details:

The current f(n) and prior f(n-1) values are stored in two registers in the Fibonacci generator. Additionally, it includes an adder, a combinational logic network that creates the sum of the previous and current values. The previous value is declared as the output. A 4-bit binary one is initialized in the register storing the current value and a 4-bit binary zero is initialized in the register carrying the prior value upon reset. In this manner, the Fibonacci sequence's first term, zero, is obtained and ascites that when there is no reset condition, the series' further terms are generated. Fig. 2 depicts the layout.

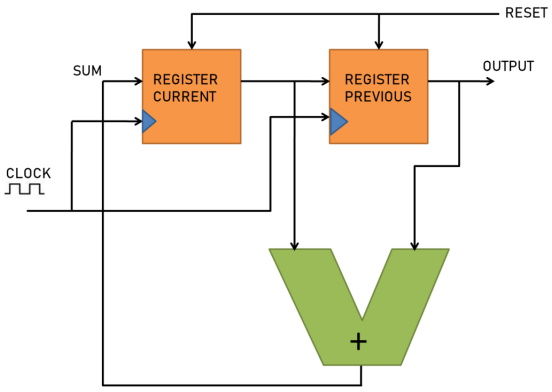

Fig. 2. Architectural Design

1) With synchronous reset: Synchronous reset denotes sampling of reset with respect to the clock. In other words, if reset is enabled, it won't take effect until the next clock edge. Fig.3 shows the code for the system design which implements a Fibonacci generator comprising a synchronous reset, an adder, and two registers.

```
// Code your design here
module fibonacci (input clk, reset,
                  output [3:0] out);
  reg [3:0] current, previous;

  always @(posedge clk)
    begin
      current <= reset? 4'd1 : current+previous ;
      previous <= reset? 4'd0 : current;
    end
  assign out = previous;
endmodule
```

Fig. 3. RTL Code

2) With asynchronous reset: Asynchronous reset samples reset without regard to the clock. This suggests that when reset is enabled, it won't verify or wait for the clock edges and will take effect right away. The code implementing a Fibonacci generator with an asynchronous reset, an adder, and two registers is shown in Fig. 4.

```
// Code your design here
module fibonacci_asy (input clk, reset,
                      output [3:0] out);
  reg [3:0] current, previous;

  always @(posedge clk or negedge reset)
    begin
      if (!reset) begin
        current <= 4'd1;
        previous <= 4'd0;
      end
      else begin
        current <= current+previous;
        previous <= current;
      end
    end
  assign out = previous;
endmodule
```

Fig. 4. RTL Code

B. Experimental results:

1) With synchronous reset: Cadence software is used to simulate and synthesize the Verilog design of a Fibonacci generator with a synchronous reset, an adder, and two registers. According to the waveform in Figure 5, a Fibonacci series term is only created at the clock's positive edge after the synchronous positive edge reset has been removed. The clock period is 2 seconds.

Fig. 5. Output Waveform

Fig.6 displays the circuit schematic that was created.

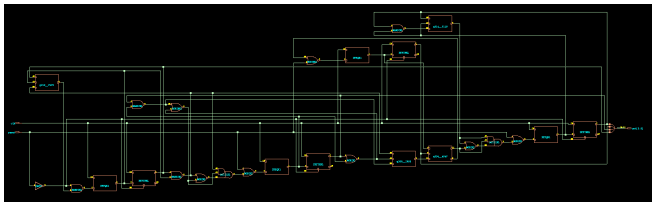

Fig. 6. Synthesized Circuit Diagram

2) With asynchronous reset: With asynchronous reset: Cadence software is used to simulate and synthesize the Verilog design of a Fibonacci generator with an asynchronous reset, an adder, and two registers. According to the waveform in Figure 7, a Fibonacci series term is only created at the clock's positive edge after the asynchronous negative edge reset has been removed. The clock period is 2 seconds.

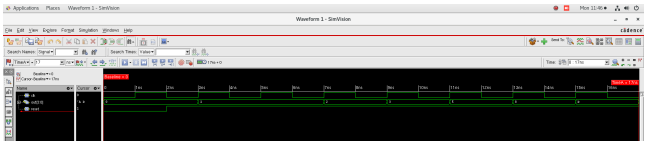

Fig. 7. Output Waveform

Fig.8 displays the circuit schematic that was created.

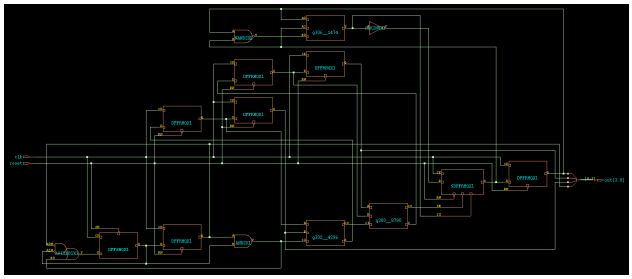

Fig. 8. Synthesized Circuit Diagram

III. Fibonacci Generator with 4-bit Adder Module using Ripple Carry Structure with Four Full Adders, Reset, and Two Registers

A. Implementation details:

A 4-bit ripple carry adder executes the combinational logic in the Fibonacci generator. Four full adder modules are instantiated in this ripple carry adder module. In the top module, that is fibo_gen, the ripple carry adder module is instantiated. Figure 9 provides the code that implements this system design.

```
// Code your design here
module fibo_gen (input clk, reset,
                 output [3:0] out);
  reg [3:0] current, previous;
  wire [3:0] sum;
  wire cout;
  always @(posedge clk)
    begin
      current <= reset? 4'b0001 : sum ;
      previous <= reset? 4'b0000 : current;
    end
  ripple_adder rca(current,previous,sum,cout);
  assign out = previous;
endmodule

module ripple_adder(X, Y, S, Co);
 input [3:0] X, Y;
 output [3:0] S;
 output Co;
 wire w1, w2, w3;
 fulladder u1(X[0], Y[0], 1'b0, S[0], w1);
 fulladder u2(X[1], Y[1], w1, S[1], w2);
 fulladder u3(X[2], Y[2], w2, S[2], w3);
 fulladder u4(X[3], Y[3], w3, S[3], Co);
endmodule

module fulladder(in0, in1, cin, out, cout);
    input in0, in1, cin;
    output out, cout;
    assign out = in0 ^ in1 ^ cin;
    assign cout = ((in0 ^ in1) & cin) | (in0 & in1);
endmodule
```

Fig. 9. RTL Code

B. *Experimental results:*

Using Cadence software, a Fibonacci generator with a 4-bit adder module using described ripple carry structure is simulated and synthesized. In Figures 10 and 11, respectively, the observed waveform and the synthesized circuit are depicted.

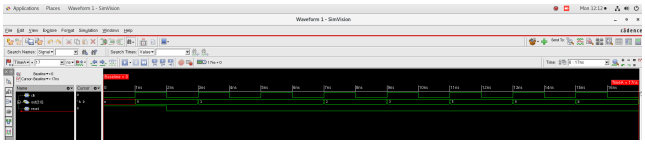

Fig. 10. Output Waveform

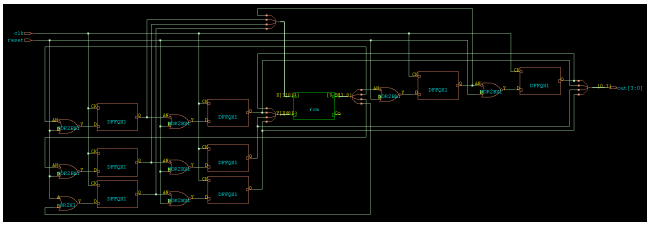

Fig. 11. Synthesized Circuit Diagram

IV. Fibonacci generator with 4-bit ripple carry adder module, reset and 4-bit register module

A. *Implementation details:*

A 4-bit ripple carry adder executes the combinational logic in the Fibonacci generator. Four full adder modules are instantiated in this ripple carry adder module. On the other hand, a 4-bit register executes the sequential logic in the Fibonacci generator. In the top module, that is fibo_gen, the ripple carry adder module as well as the register module is instantiated. Figure 12 provides the code that implements this system design.

```
// Code your design here
module fibo_gen (input clk, reset,
                 output [3:0] out);
  wire [3:0] current, previous;
  wire [3:0] sum;
  wire cout;
  flipflop f1(.clk(clk),.reset(reset),.d(current),.reset_value(4'b0),.q(previous));
  flipflop f2(.clk(clk),.reset(reset),.d(sum),.reset_value(4'b1),.q(current));
  ripple_adder rca(.X(current),.Y(previous),.S(sum),.Co(cout));
  assign out = previous;
endmodule

module ripple_adder(X, Y, S, Co);
 input [3:0] X, Y;
 output [3:0] S;
 output Co;
 wire w1, w2, w3;
 fulladder u1(.in0(X[0]), .in1(Y[0]), .cin(1'b0), .out(S[0]), .cout(w1));
 fulladder u2(.in0(X[1]), .in1(Y[1]), .cin(w1), .out(S[1]), .cout(w2));
 fulladder u3(.in0(X[2]), .in1(Y[2]), .cin(w2), .out(S[2]), .cout(w3));
 fulladder u4(.in0(X[3]), .in1(Y[3]), .cin(w3), .out(S[3]), .cout(Co));
endmodule

module fulladder(in0, in1, cin, out, cout);
    input in0, in1, cin;
    output out, cout;
    assign out = in0 ^ in1 ^ cin;
    assign cout = ((in0 ^ in1) & cin) | (in0 & in1);
endmodule

module flipflop(input clk, reset, input [3:0] d, input [3:0] reset_value , output reg [3:0] q);
always @(posedge clk)
  if(reset)
    q<=reset_value;
  else
    q<=d;
endmodule
```

Fig. 12. RTL Code

B. *Experimental results:*

Using Cadence software, a Fibonacci generator with a 4-bit ripple carry adder module and 4-bit register module is simulated and synthesized. In Figures 13 and 14, respectively, the observed waveform and the synthesized circuit are depicted.

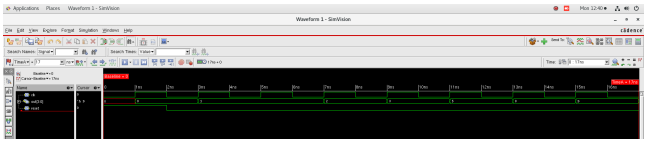

Fig. 13. Output Waveform

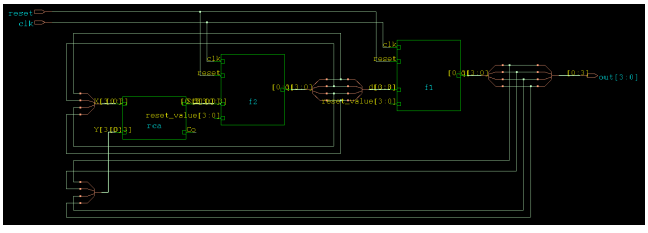

Fig. 14. Synthesized Circuit Diagram

V. Fibonacci generator with 4-bit ripple carry adder module, reset and register module consisting of four D flip-flops

A. Implementation details:

A 4-bit ripple carry adder executes the combinational logic in the Fibonacci generator. Four full adder modules are instantiated in this ripple carry adder module. On the other hand, a register module containing four instances of D flip-flop modules executes the sequential logic in the Fibonacci generator. In the top module, that is fibo_gen, the ripple carry adder module as well as the register module is instantiated. Figure 15 provides the code that implements this system design.

```
// Code your design here
module fibo_gen (input clk, reset,
                 output [3:0] out);
  wire [3:0] current, previous;
  wire [3:0] sum;
  wire cout;
  reg4bit r1(.clk(clk),.reset(reset),.D(current),.r_v(4'b0),.Q(previous));
  reg4bit r2(.clk(clk),.reset(reset),.D(sum),.r_v(4'b1),.Q(current));
  ripple_adder rca(.X(current),.Y(previous),.S(sum),.Co(cout));
  assign out = previous;
endmodule
module ripple_adder(X, Y, S, Co);
 input [3:0] X, Y;
 output [3:0] S;
 output Co;
 wire w1, w2, w3;
  fulladder u1(.in0(X[0]), .in1(Y[0]), .cin(1'b0), .out(S[0]), .cout(w1));
  fulladder u2(.in0(X[1]), .in1(Y[1]), .cin(w1), .out(S[1]), .cout(w2));
  fulladder u3(.in0(X[2]), .in1(Y[2]), .cin(w2), .out(S[2]), .cout(w3));
  fulladder u4(.in0(X[3]), .in1(Y[3]), .cin(w3), .out(S[3]), .cout(Co));
endmodule
module fulladder(in0, in1, cin, out, cout);
    input in0, in1, cin;
    output out, cout;
    assign out = in0 ^ in1 ^ cin;
    assign cout = ((in0 ^ in1) & cin) | (in0 & in1);
endmodule
module flipflop(input clk, reset, reset_value , d, output reg q);
always @(posedge clk)
  if(reset)
    q<=reset_value;
  else
    q<=d;
endmodule
module reg4bit (input clk,input reset, input [3:0] r_v, input [3:0] D, output [3:0] Q);

  flipflop fl1 (.d(D[0]),.reset(reset),.clk(clk),
                .reset_value(r_v[0]),. q(Q[0]));
  flipflop fl2 (.d(D[1]),.reset(reset),.clk(clk),
                .reset_value(r_v[1]),. q(Q[1]));
  flipflop fl3 (.d(D[2]),.reset(reset),.clk(clk),
                .reset_value(r_v[2]),. q(Q[2]));
  flipflop fl4 (.d(D[3]),.reset(reset),.clk(clk),
                .reset_value(r_v[3]),. q(Q[3]));
endmodule
```

Fig. 15. RTL Code

B. Experimental results:

Using Cadence software, a Fibonacci generator with a 4-bit ripple carry adder module and a register module including D flip-flop module instances is simulated and synthesized. In Figures 16 and 17, respectively, the observed waveform and the synthesized circuit are depicted.

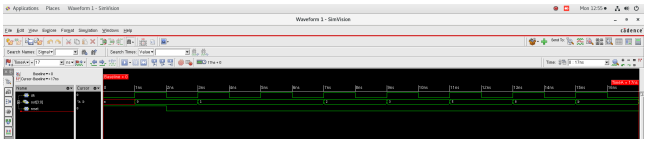

Fig. 16. Output Waveform

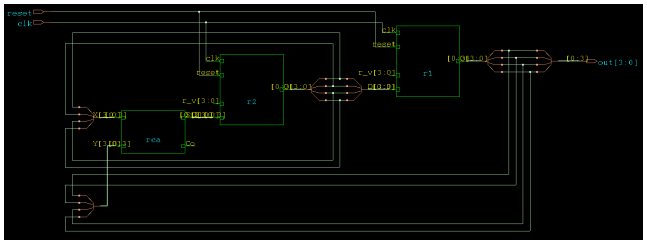

Fig. 17. Synthesized Circuit Diagram

VI. Testbench

In Figure 18 and Figure 19, the testbench codes created for verifying the DUT with synchronous and asynchronous resets are depicted.

```
// Code your testbench here
// or browse Examples
module fb_tb();
  reg clk, reset;
  wire [3:0] out;
  fibonacci fb (clk, reset, out);
  initial begin
  forever #1 clk = ~clk;
  end

  initial begin
    $monitor ("At time = %t, Out = %d", $time, out);
    clk = 0; reset = 1;

    #2 reset = 0;

    #15 $finish;

  end
endmodule
```

Fig. 18. Testbench of DUT with synchronous reset

```
// Code your testbench here
// or browse Examples
module fb_tb_asy();
  reg clk, reset;
  wire [3:0] out;
  fibonacci_asy fb (clk, reset, out);
  initial begin
  forever #1 clk = ~clk;
  end
  initial begin
    $monitor ("At time = %t, Out = %d", $time, out);
    clk = 0; reset = 0;
    #2 reset = 1;
    #15 $finish;
  end
endmodule
```

Fig. 19. Testbench of DUT with asynchronous reset

VII. Analysis

The following table in Fig.20 shows the comparison of all Fibonacci generators with various hierarchical designs in regard to power, time, and area

	Fibonacci generator with reset, an adder, and two registers		Fibonacci generator with 4-bit ripple carry adder module consisting of four full adders, reset, and two registers	Fibonacci generator with 4-bit ripple carry adder module, reset and 4-bit register module	Fibonacci generator with 4-bit ripple carry adder module, reset and register module consisting of four D flip-flops
	With synchronous reset	With asynchronous reset			
Power (nW)	25063.170	34244.841	27483.987	26835.319	26835.319
Timing (ps)	8705	8710	8618	8617	8617
Area (nm^2)	226.313	228.584	230.854	230.854	230.854

Fig. 20. Comparative analysis of all the Fibonacci generators with different hierarchical designs

VIII. CONCLUSION

In Cadence software, the Verilog design of the Fibonacci generator is successfully simulated and synthesized. The hierarchical design encourages breaking down a design into different layers of abstraction, leading to the creation of a top-level schematic that only contains block symbols and their relationships. It promotes the reuse of low-level modules and facilitates comprehension of the design by allowing us to concentrate on the operation and interconnection of individual modules.