ECE-399: Selected Topics in ECE: Design with FPGAs
Report 2: Sequential Logic Design, 4-bit Adder

Prof. Shlayan
Nithilam Subbaian

May 15, 2020

1 Objective of Report 2:

For this report, the goal is to familiarize myself with multiplexers and multi-bit adders in general.

2 Materials

1. Zedboard [1]
2. 4GB SD card (purchased independently of Zedboard Kit)
3. 12V Power supply
4. Micro USB cable
5. USB Adapter: Male Micro-B to Female Standard-A
6. Xilinx Vivado Design Suite Version 2018.3 [4]

3 Lecture 5 Notes

This notes in this section are based of Lecture 5 [2].

3.1 Code Samples

Multiple Instances of the Same Type of Gate in One Construct:

```
1
2 and [inst1] (out11, in11, in12), [inst2] (out21, in21, in22, in23), [inst3] (out31, in31
      , in32, in33);
```

Allowing the delays to be expressed when instantiating gates

```
1 and #5 A1(out1, in1, in2); // the rise and fall delays are 5 units
2 and #(2,5) A2(out2, in1, in2); // the rise delay is 2 units and the fall delay is 5
      units
3 notif1 #(2, 5, 4) A3(out3, in2, ctrl1); //the rise delay is 2, fall delay is 5, turn-off
       delay is 4 units
```

3.2 Dataflow Modeling

Dataflow modeling is used to describe combinational circuits using “continuous assignment” in which a value is assigned to a data type called “net”. Continuous assignment syntax: assign[delay] LHS_net= RHS_expression;
. This statement is evaluated at any time any of the course operand value changes and the result is assigned to the destination net after the delay unit.

```
1 assignout1 = in1 & in2; // perform and function on in1 and in2 and assign the result to
2 out1assignout2 = not in1;
3 assign#2 z[0] = ~(ABAR & BBAR & EN); // perform function and assign the result after 2
      units
```

3.3 Target in the Continuous Assignment Expression

```
// 1.A scalar net
assign out1 = in1 & in2;

// 2.Vector net

// 3.Constant bit-select of a vector
assign #2 z[0] = ~(ABAR & BBAR & EN);

// 4.Constant part-select of a vector

// 5.Concatenation of any of the above

wire COUT, CIN; // scalar net declaration
wire[3:0] SUM, A, B; // vector nets declaration
assign{COUT,SUM} = A + B + CIN; // A and B vectors are added with CIN and the result is
    // assigned to a concatenated vector of a scalar and vector nets
```

Note that multiple continuous assignment statements are not allowed on the same destination net.

3.4 Behavioral Modeling

Behavioral Modeling is used to describe complex circuits, primarily sequential circuits, but also pure combinatorial circuits. The statements for modeling the behavior of a design are:

```
initial[timing_control] procedural_statements; //both may contain multiple  procedural
    statements
always[timing_control] procedural_statements;
```

These statements (both “initial” and “always”) are executed concurrently at time t= 0, and then afterwards only “always” statements are executed. “initial” statemetns are non-synthesizable and used traditionally in testbenches, whereas “always” statements are synthesizable.

3.5 Numbers Representation

[size]’[signed][radix]value ([ ] indicates optional)

1. size - The number of binary bits the number is comprised of Default is 32 bits.
2. ’ - A separator
3. signed - including S Indicates if the value is signedDefault is unsigned.
4. Radix - b : binary, o : octal, h : hex, d : decimal (also default)

In Verilog HDL a signal can have the following four basic values:
0 : logic-0 or false
1 : logic-1 or true
x : unknown
z : high-impedance

There are three types of constants in Verilog HDL: Integer, Real and String. If the size specified is larger than the value for the specified constant, the number is padded to the left with 0’s but for the case where the left most bit is x or z then the padding is done with x or z. If the size specified is smaller then the extra left most bits are ignored. If the size is not specified then it will use 32-bit data.

3.6 Procedural Statements

A module may contain an arbitrary number of “initial” or “always” statements and may contain one or more procedural statements within them. Procedural statements are executed sequentially.
Procedural Assignment:

```
[delay] register\_name= [delay] expression; // blocking
[delay] register\_name<= [delay] expression; // non-blocking
```

Example of Procedural Assignment:

```
/ time 0: a=0; time 10: a=1; time 15 (#10+#5): a=2;
begin
a = 0;
#10 a = 1;
#5 a = 2;
end

// time 0: a=0; time 5: a=2; time 10: a=1;

begin
a <= 0;
#10 a <= 1;
#5 a <= 2;
end

// both assignments are evaluated before a or b changes
begin
a <= b;
b <= a;
end
```

3.7 Statements

1. conditional statement
   condition ? Expression1 : expression2;

   Example modeling a tri-state buffer:
   assign data_out= (enable) ? data_reg: 8'bz;

2. case statement

   Example:
   case (i)
   0 : y = a;
   1 : y = b;
   2 : y = c;
   default :$display("Error");
   endcase

3. loop statement: Forever, repeat, while, for

4. wait statement

3.8 A Multiplexer -Two Ways

```
reg[4:0] mux;
reg[1:0] addr;

//ONE
mux = (addr== 2'b00) ? i0 :
((addr== 2'b01) ? i1 :
((addr== 2'b10) ? i2 :
((addr== 2'b11) ? i3 :4'bz)));

//TWO
case(addr)
    2'b00: mux = i0;
    2'b01: mux = i1;
    2'b10: mux = i2;
    2'b11: mux = i3;
    default: mux = 4'bz;
endcase
```

To generate a combinatorial circuit, the "always" block should not be edge sensitive, every branch of the conditional statement should define all output, and every case of case statement should define all output and must have a default case.

The destination (LHS) should be of regtype; either scalar or vector

```
reg m; // scalar reg type
reg [7:0] switches; // vector reg type
```

Example 2-to-1 Mux

```
always @ (x or y or s) //The sensitive list, always block executes when changed
begin
if(s==0)
    m=y;
else
    m=x;
end
```

4 Vivado and Zedboard Work

The test benches where provided in the lectures. Here is a tutorial [3] I used to complete these labs .

4.1 Create a two-bit wide 2-to-1 multiplexer using gate-level modeling.

lab 2 Q1:

```
module dlab2_1(X, Y, S, O  );

input [0:0] S;
input  [1:0] X;
input  [1:0] Y;
output  [1:0] O;

wire W3_x0;
wire W3_x1;
wire W5_y0;
wire W5_y1;
wire W6_x0;
wire W6;


and (W3_x0, X[0], S), (W5_y0, Y[0], W6);
and (W3_x1, X[1], S), (W5_y1, Y[1], W6);
not (W6, S);
or (O[0], W3_x0, W5_y0);
or (O[1], W3_x1, W5_y1);

endmodule
```

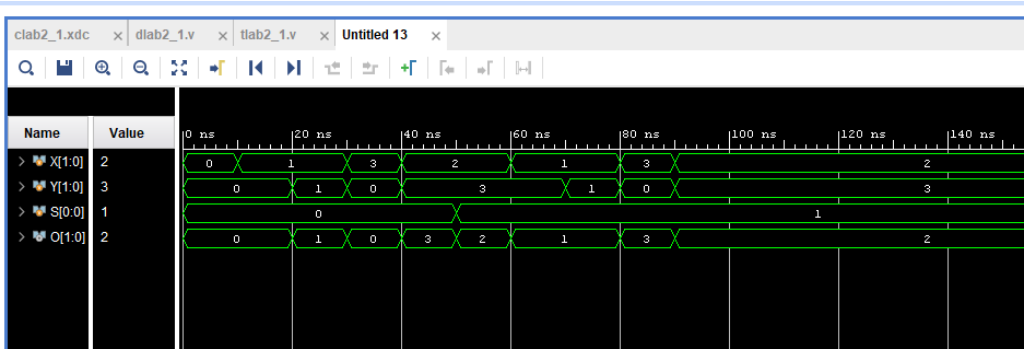

Figure 1: lab 2 Q1

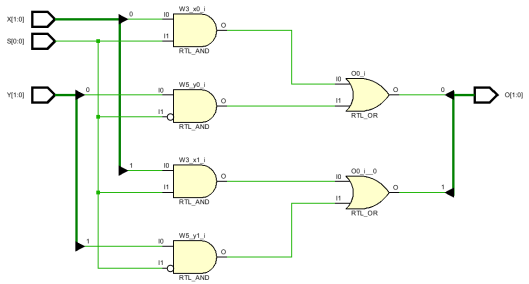

Figure 2: lab 2 Q1

4.2 Model a two-bit wide 2-to-1 multiplexer using dataflow modeling with net delays of 3 ns.

lab 2 Q2:

```
1
2  module dlab2_1(X, Y, S, O  );
3
4  input [0:0] S;
5  input  [1:0] X;
6  input  [1:0] Y;
7  output  [1:0] O;
8
9  assign #3 O[0] =(S)?X[0]:Y[0];
10 assign #3 O[1] =(S)?X[1]:Y[1];
11
12 endmodule
```

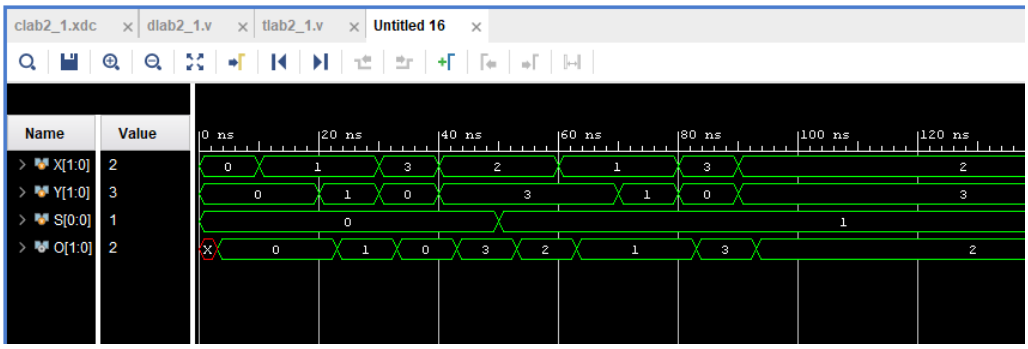

Figure 3: lab 2 Q2

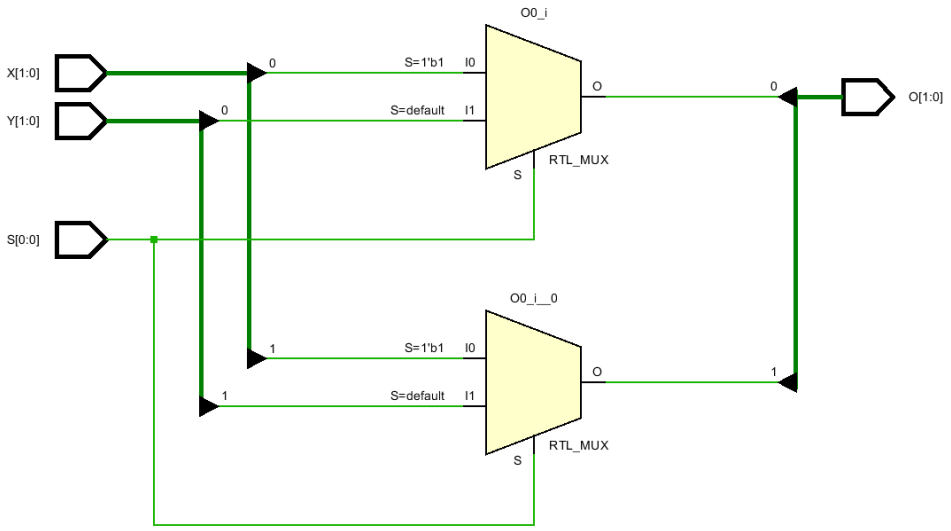

Figure 4: lab 2 Q2

4.3 Create a two-bit wide 2-to-1 multiplexer using behavioral modeling.

lab 2 Q3:

```
module dlab2_1(X, Y, S, O  );

input [0:0] S;
input  [1:0] X;
input  [1:0] Y;
output reg [1:0] O;

always @(X or Y or S)
begin
    if(S)
        O = Y;
    else
        O = X;
end

endmodule
```

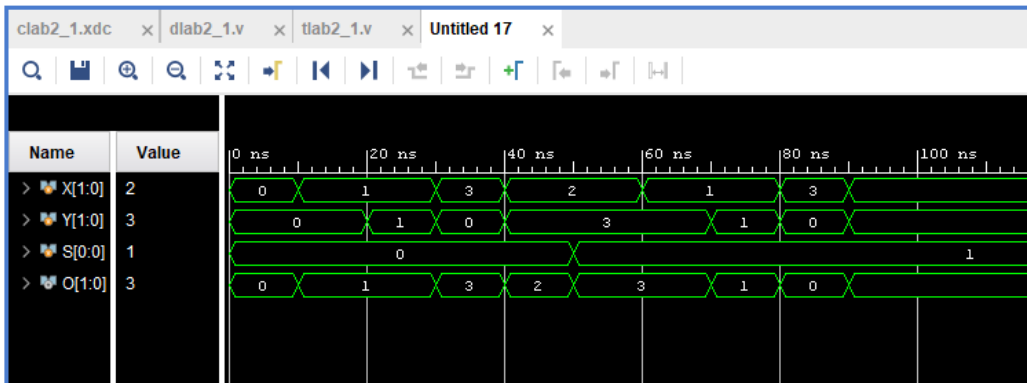

Figure 5: lab 2 Q3

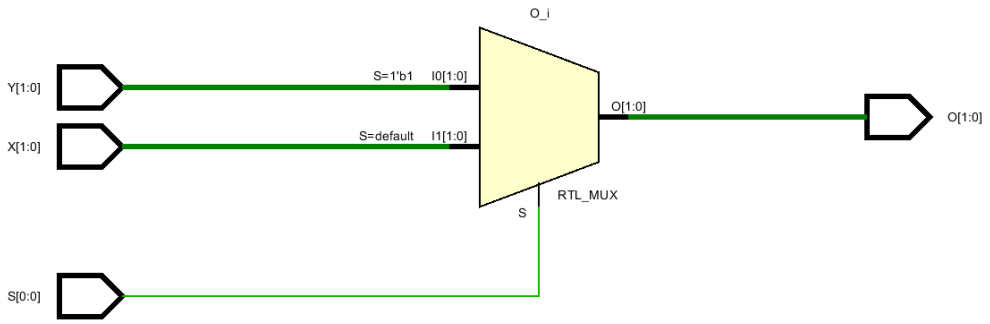

Figure 6: lab 2 Q3

4.4 Create a two-bit wide 3-to-2 multiplexer using two 2-1 multiplexers from parts 1, 2, 3, or 4

lab 2 Q4:

```
module dlab2_1(X, Y, Z, S, S1, O, O1  );

input [0:0] S;
input [0:0] S1;
input  [1:0] X;
input  [1:0] Y;
input  [1:0] Z;
output  [1:0] O;
output  [1:0] O1;

assign #3 O[0] =(S)?X[0]:Y[0];
assign #3 O[1] =(S)?X[1]:Y[1];

assign #3 O1[0] =(S1)?O[0]:Z[0];
assign #3 O1[1] =(S1)?O[1]:Z[1];

endmodule
```

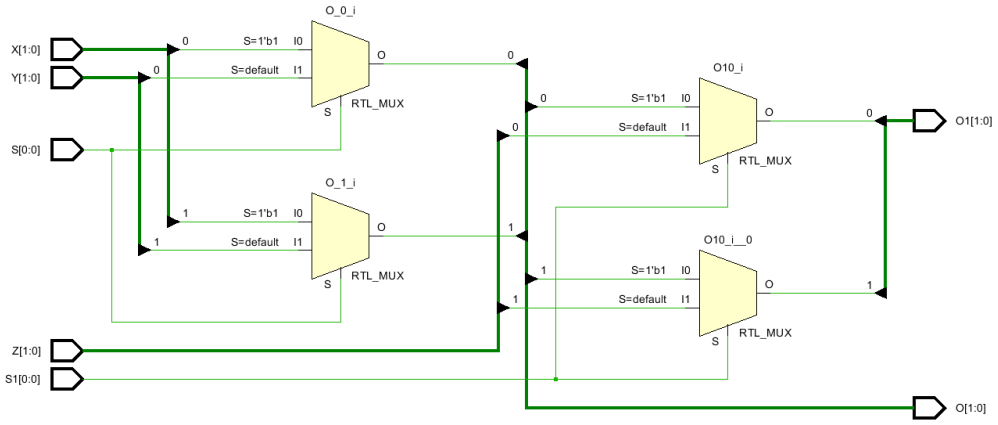

Figure 7: lab 2 Q4

4.5 Create a 4-bit ripple carry adder using dataflow modeling

full adder simulation:

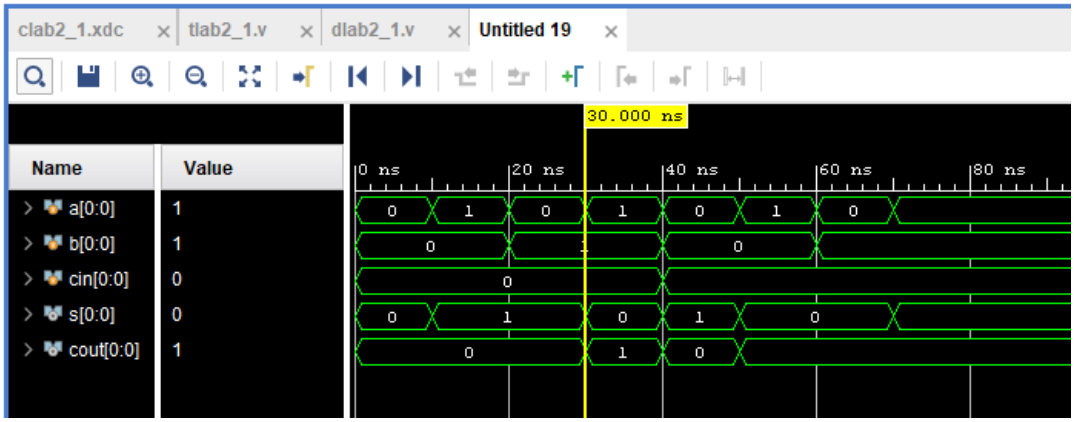

Figure 8: lab 2 Q4

4-bit ripple carry adder :

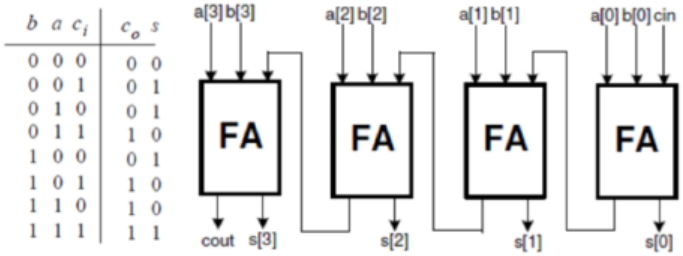

Figure 9: lab 2 Q4

```
1
2  module dlab2_1(a, b, cin, s, cout );
3
4  input wire [3:0] a;
5  input wire [3:0] b;
6  input wire [0:0] cin;
7  output wire [3:0] s;
8  output wire  [0:0] cout;
9
10 wire w1;
11 wire w2;
12 wire w3;
13
14 fulladder block1(.a(a[0]), .b(b[0]), .cin(cin),
15  .s(s[0]), .cout(w1));
16 fulladder block2(.a(a[1]), .b(b[1]), .cin(w1),
17  .s(s[1]), .cout(w2));
18 fulladder block3(.a(a[2]), .b(b[2]), .cin(w2),
19  .s(s[2]), .cout(w3));
20 fulladder block4(.a(a[3]), .b(b[3]), .cin(w3),
21  .s(s[3]), .cout(cout));
22
23 endmodule
```

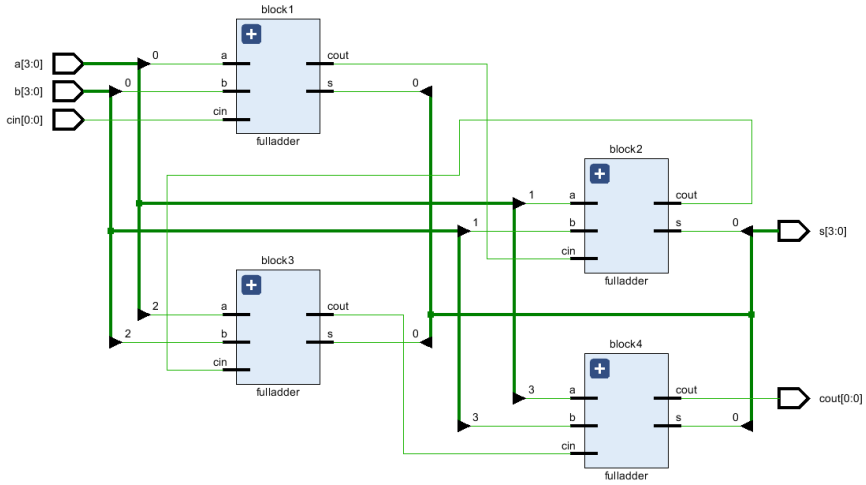

Figure 10: 4-bit ripple carry adder

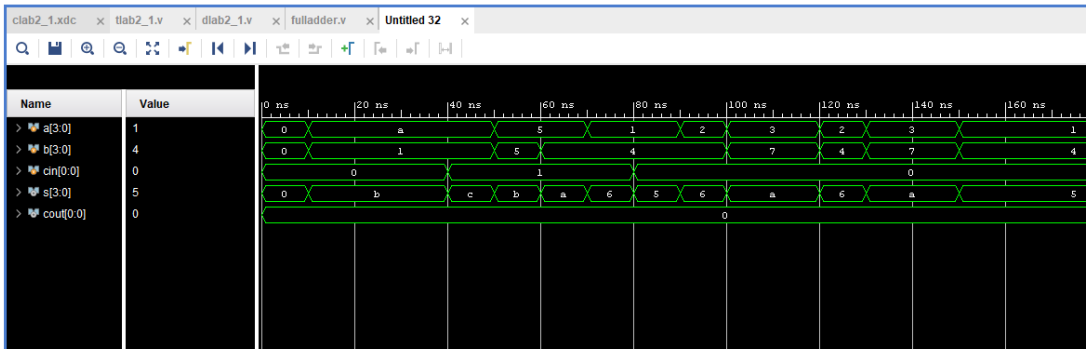

Figure 11: 4-bit ripple carry adder

4.6 Carry Lookahead Adder

Carry Lookahead Adder:

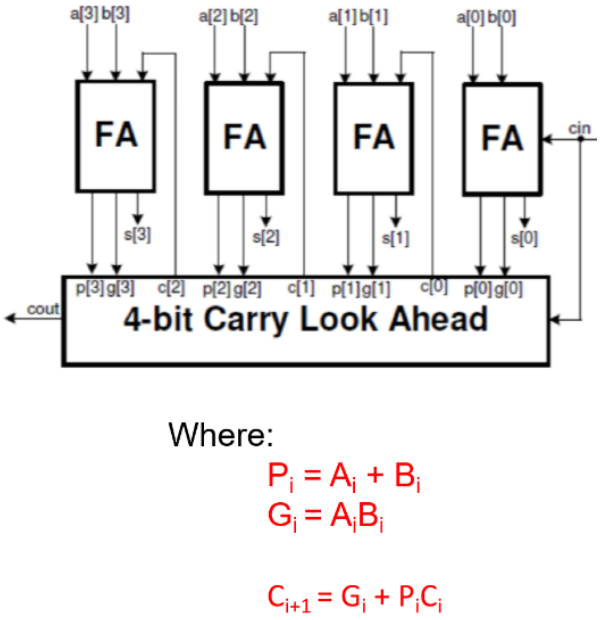

Figure 12: Carry Lookahead Adder

```
module dlab2_1(a, b, cin, s, cout );

input wire [3:0] a;
input wire [3:0] b;
input wire [0:0] cin;
output wire [3:0] s;
output wire  [0:0] cout;
wire  [2:0] c;

fulladder block1(.a(a[0]), .b(b[0]), .cin(cin),
 .s(s[0]));
assign c[1] = (a[1] & b[1]) | ((b[1] | b[1]) & c[0]);

fulladder block2(.a(a[1]), .b(b[1]), .cin(c[0]),
 .s(s[1]));
assign c[0] = (a[0] & b[0]) | ((b[0] | b[0]) & cin);

fulladder block3(.a(a[2]), .b(b[2]), .cin(c[1]),
 .s(s[2]));
assign c[2] = (a[2] & b[2]) | ((b[2] | b[2]) & c[1]);

fulladder block4(.a(a[3]), .b(b[3]), .cin(c[2]),
 .s(s[3]));
 assign cout = (a[3] & b[3]) | ((b[3] | b[3]) & c[2]);


endmodule
```

p and g combined to make c for simplifcation

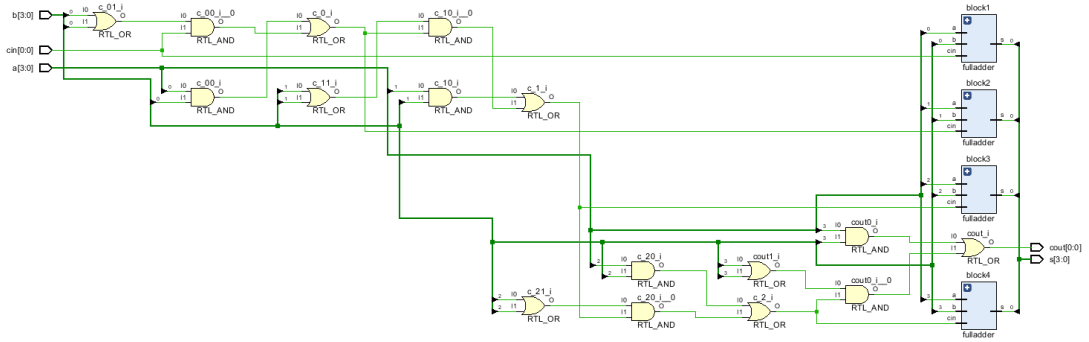

Figure 13: Carry Lookahead Adder

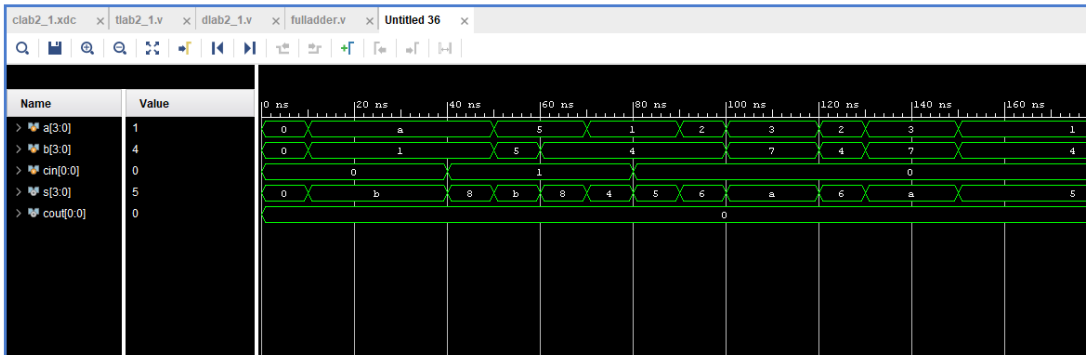

Figure 14: Carry Lookahead Adder

5 Discussion of Difficulties

References

[1] Avnet Electronics Marketing. *Zynq^{TM}Evaluation and Development Hardware User's Guide*. URL: https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf.

[2] Naveen Shlayan. *Sequential Logic Design*. URL: https://moodle.cooper.edu/moodle/pluginfile.php/82181/mod_resource/content/1/Lecture5.pdf.

[3] technobyte. URL: https://www.technobyte.org/verilog-multiplexer-2x1/.

[4] xlinx. *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html.