Lab 3 Report: RCA & 4-Bit Comparator

Matt Hoertig, Hayden Rinn, and Luis Gomez

CPE 133-11

October 10, 2018

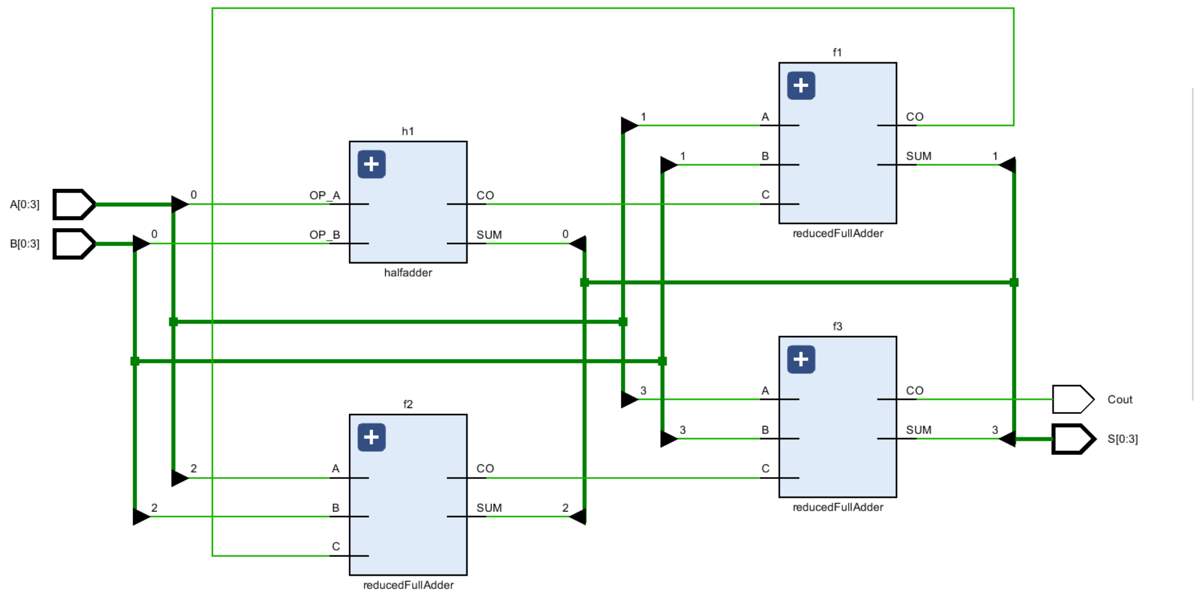
Experiment 4: 4-Bit Ripple Carry Adder (RCA)

Summary:

The goal of this experiment was to design, implement, and test a digital circuit. Our circuit utilized the half adder and full adder that we designed previously in order to design and implement the 4-bit Ripple Carry Adder. The black box diagram of the circuit can be found in Figure 1. The general diagram of the circuit is Figure 2. In addition, the K-Maps for each of the components can be found in Figure 3.

Figure 1: The Black Box Diagram of the 4-Bit RCA

Figure 2: The Circuit Diagram of the 4-Bit RCA



FA SUM=	A,B,C	00	01	11	10
	0	0	1	0	1
	1	1	0	1	0

FA Cout=	A,B,C	00	01	11	10
	0	0	0	1	0
	1	0	1	1	1

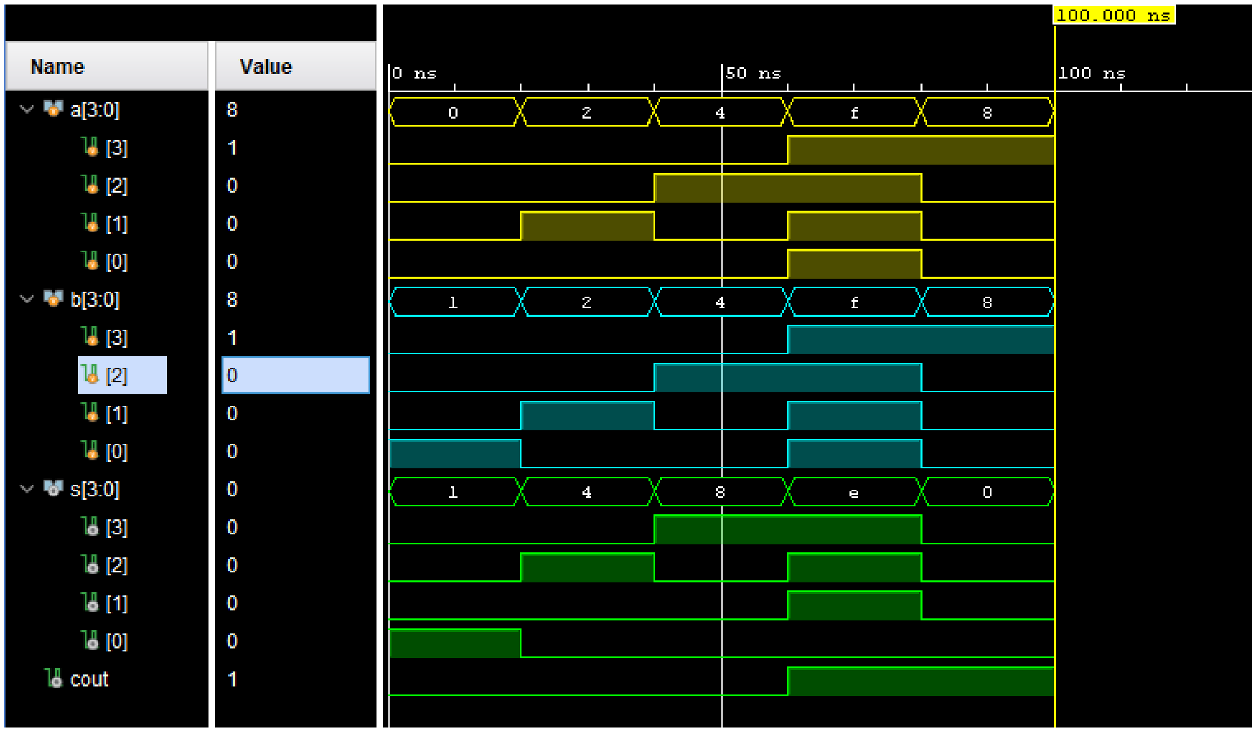
HA SUM=	AB	0	1
	0	0	1
	1	1	0

HA
Cout=

AB	0	1
0	0	0
1	0	1
1	0	1

Figure 3: The K-Maps for the Components of the 4-Bit FCA.
Note: Lighter colors are Essential PI, Bolder colors are Implicants

<u>Verification:</u>
To verify that the efficacy of our behavioral models for the RCA and the 4-bit Comparator, we exhaustively ran behavioral simulations and physical board tests. We tested for several instances that could possibly bring up conflict. The results of our simulation can be found below in the Figure 4 diagram. Once we found that our simulation had matched that of the truth table we proceeded to run the FPGA. The results of our run on the FPGA are shown below in the video.



Name	Value
a[3:0]	8
[3]	1
[2]	0
[1]	0
[0]	0
b[3:0]	8
[3]	1
[2]	0
[1]	0
[0]	0
s[3:0]	0
[3]	0
[2]	0
[1]	0
[0]	0
cout	1

Figure 4: These were the results of the simulation for the 4-Bit RCA

<u>Questions:</u>
1. In your own words, briefly but completely explain why the circuit in this lab activity is referred to as a ripple carry adder.

The reason why it is called a ripple carry adder is due to the amount of adders that are present in the program. The carry out of the half adder becomes the carry in for the next adder, a full adder. Then, the carry out of the full adder becomes the carry in for the next full adder and it just ripples down the line. So each output is carried or “rippled’ to the next adder giving it the name of the Ripple Carry Adder.

2. If you needed to extend the RCA from this lab activity to an 8-bit RCA by using a structural model with two 4-bit RCAs, what changes would you need to apply to the 4-bit RCA?

If you needed to extend the RCA from this lab activity to an 8-bit RCA by using a structural model with two 4-bit RCAs you would have to make the second 4-bit RCA start with a full adder and make the carry out of the first 4-bit RCA become the carry-in of the first full adder of the second 4-bit RCA. That would the way to extend the RCA from this lab activity to an 8-bit RCA.

3. How many rows would there be in the truth table for a 4-bit RCA? How many input and output variables are there? Would it be feasible to design a 32-bit RCA using this technique?

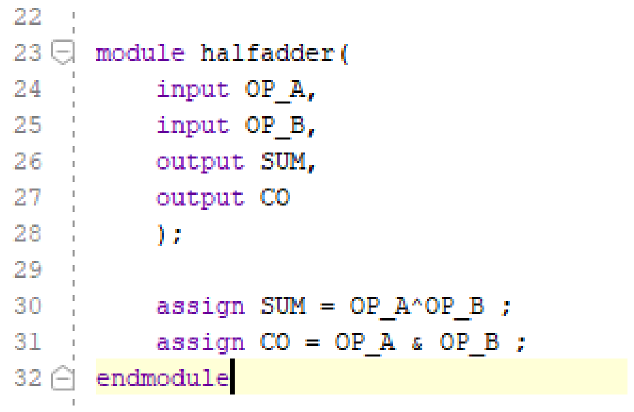
There would be 256 rows in the truth table for a 4-bit RCA due to their being two 4-bit inputs. That would result in there being 2^8 rows in the truth table. For the 4-bit RCA there are 8 input variables and 5 output variables. It would not be feasible to design a 32-bit RCA using this technique because there would be 2^64 rows in the truth table. In addition, there are 64 possible input combinations and 33 output variables that are possible.

4. If you were to implement the 4-bit RCA using discrete logic, how many logic gates would be required?

If you were to implement the 4-bit RCA using discrete logic, 23 gates would be required in order to the produce this program. A Full Adder has 7 gates. A Half Adder has 2 gates. A 4-Bit RCA is composed of 3 Full Adders and 1 Half Adder, so in terms of discrete logic a 4-bit RCA would require 23 gates.

<u>Code:</u>

Below: Half-Adder Code:

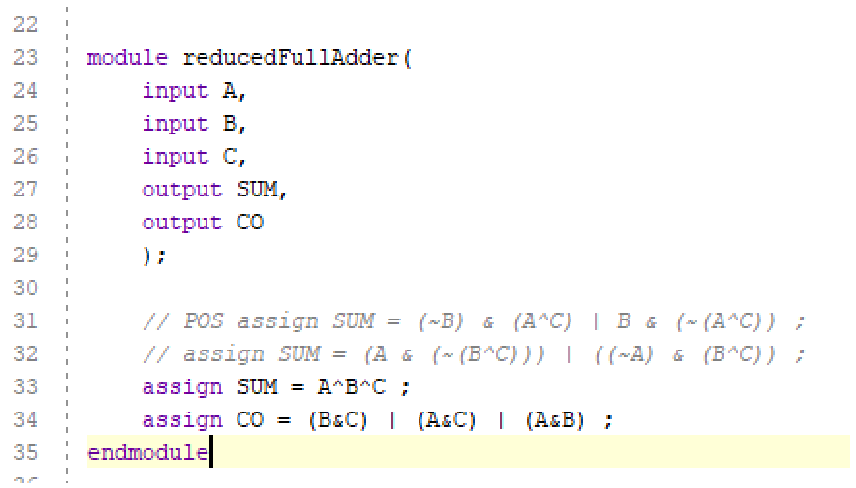


```

module halfadder(
    input OP_A,
    input OP_B,
    output SUM,
    output CO
    );

    assign SUM = OP_A^OP_B ;
    assign CO = OP_A & OP_B ;
endmodule
```

Below: Reduced Full-Adder Code:

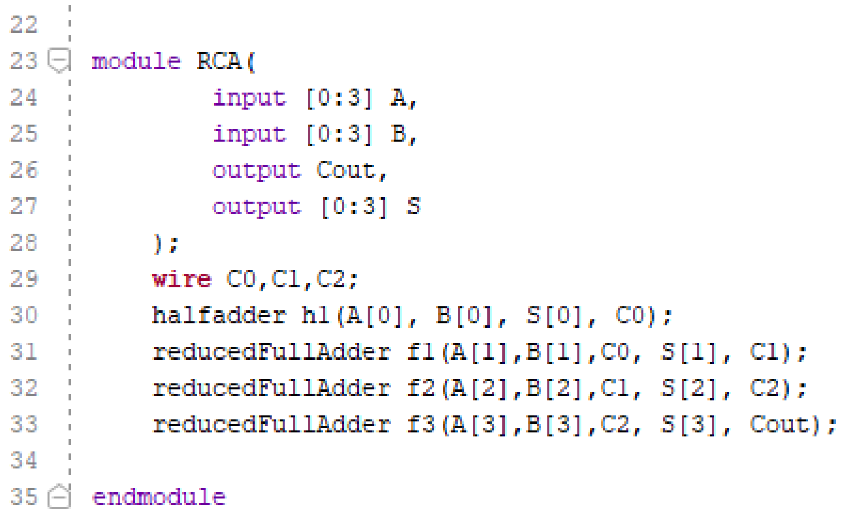


```verilog

module reducedFullAdder(
    input A,
    input B,
    input C,
    output SUM,
    output CO
    );

    // POS assign SUM = (~B) & (A^C) | B & (~(A^C)) ;
    // assign SUM = (A & (~(B^C))) | ((~A) & (B^C)) ;
    assign SUM = A^B^C ;
    assign CO = (B&C) | (A&C) | (A&B) ;
endmodule
```

Below: 4-Bit RCA Code



```verilog

module RCA(
        input [0:3] A,
        input [0:3] B,
        output Cout,
        output [0:3] S
    );
    wire C0,C1,C2;
    halfadder h1(A[0], B[0], S[0], C0);
    reducedFullAdder f1(A[1],B[1],C0, S[1], C1);
    reducedFullAdder f2(A[2],B[2],C1, S[2], C2);
    reducedFullAdder f3(A[3],B[3],C2, S[3], Cout);

endmodule
```

<u>**Experiment 5: 4-Bit Comparator**</u>

<u>**Summary:**</u>

The goal of this experiment was to design, implement, and test a digital circuit. Our circuit took in two 4-bit values and then compared them to one another and told of if the A input was less than, equal to, or greater than the B input. The black box diagram of the circuit can be found in Figure 5. The general diagram of the circuit is Figure 6.

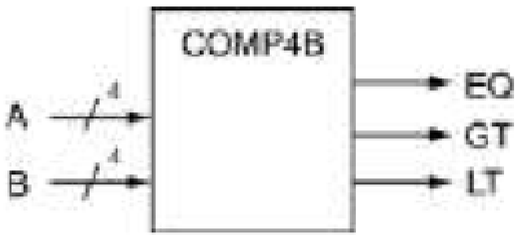


Figure 5: The Black Box Diagram of the 4-Bit Comparator

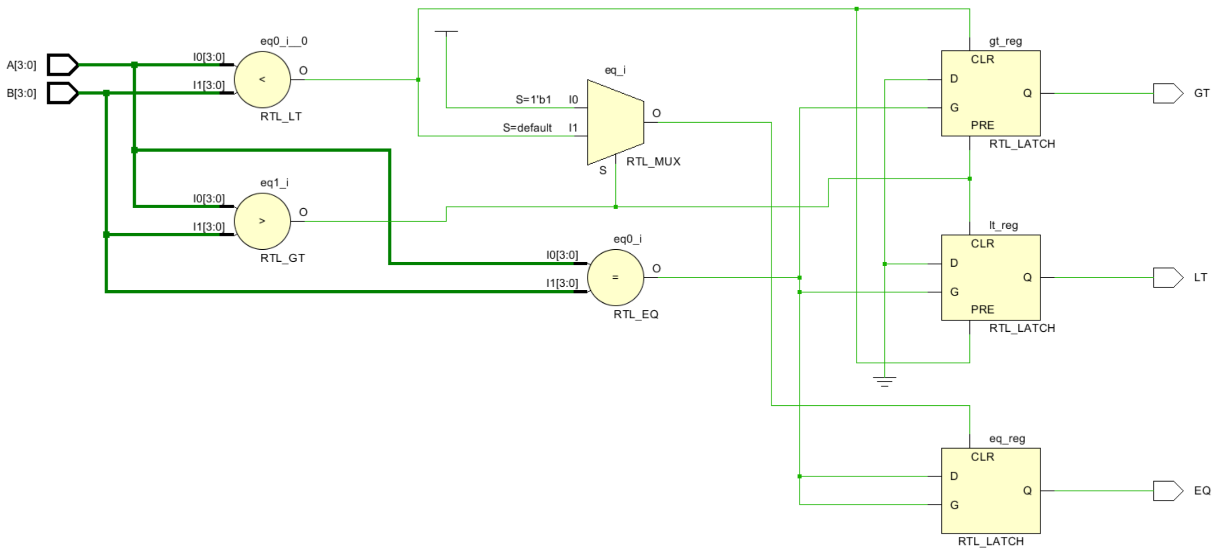


Figure 6: The Circuit Diagram of the 4-Bit Comparator

Verification:

In order to verify that the code for our circuit was effective and correct, we ran a simulation of the code and compared the results of the code to that of our truth table. We tested for several instances that could possibly bring up conflict. The results of our simulation can be found below in the Figure y diagram. Once we found that our simulation had matched that of the truth table we proceeded to run the FPGA. The results of our run on the FPGA are shown below in the video.

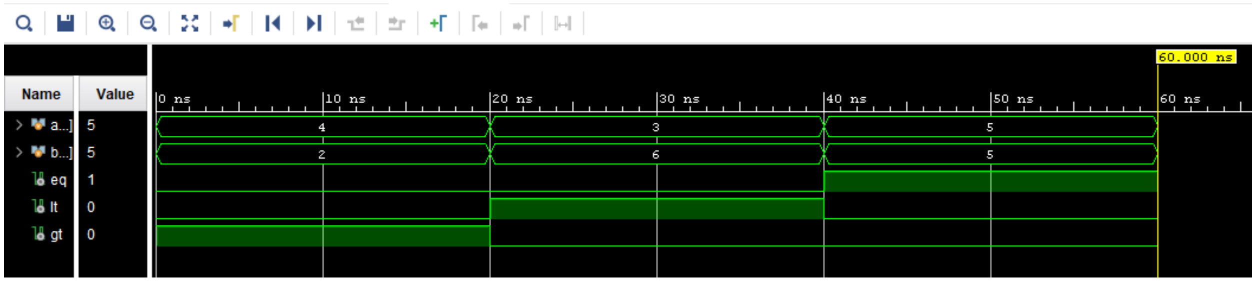


Figure 7: These were the results of the simulation for the 4-Bit Comparator

Questions:

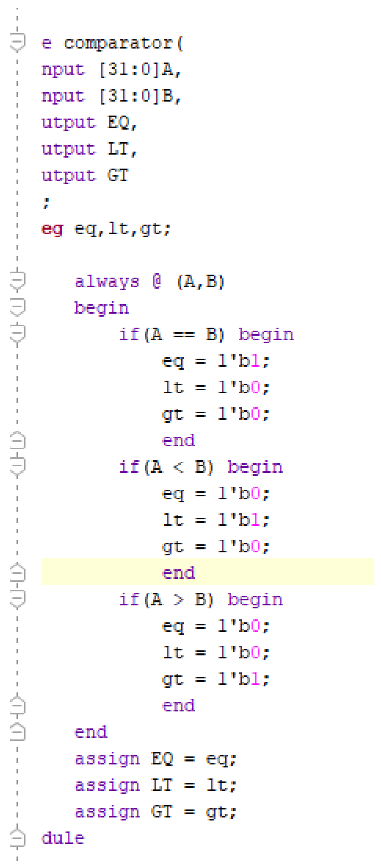
1. In your own words, briefly describe the advantage of using a behavioral model as opposed to a gate-level implementation of the comparator.

There are many advantages to using behavioral models as opposed to a gate-level implementation of the comparator. In our opinion, the main advantage of using a behavioral model is that we are defining the behavior at a high level instead of writing out each individual logic gate. This is a much more efficient way to code something and it doesn't take nearly as much logic as having each input go into its own gate.

2. Modify your 4-bit comparator model in this activity to be a 32-bit comparator; include this code with your lab activity report.

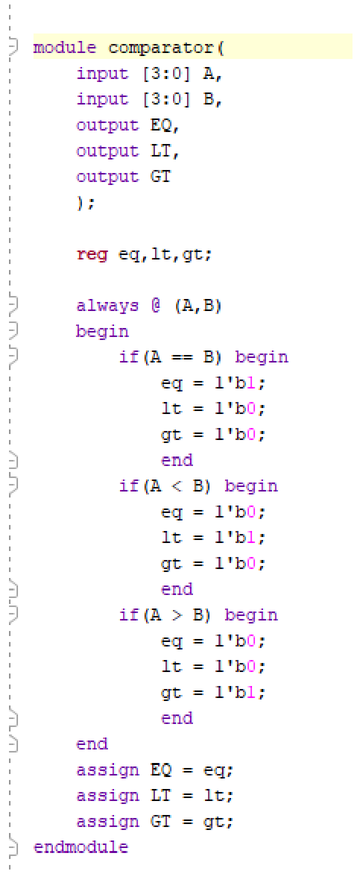
The only difference between the 4-bit comparator model and a 32-bit comparator is the number of inputs that are put into the comparator. As you can see, the code is all the same but the number of the inputs have been increased from 4 to 32.

Code:



```
e comparator(
nput [31:0]A,
nput [31:0]B,
utput EQ,
utput LT,
utput GT
;
eg eq,lt,gt;

   always @ (A,B)
   begin
       if(A == B) begin
           eq = 1'b1;
           lt = 1'b0;
           gt = 1'b0;
           end
       if(A < B) begin
           eq = 1'b0;
           lt = 1'b1;
           gt = 1'b0;
           end
       if(A > B) begin
           eq = 1'b0;
           lt = 1'b0;
           gt = 1'b1;
           end
   end
   assign EQ = eq;
   assign LT = lt;
   assign GT = gt;
dule
```



```
module comparator(
    input [3:0] A,
    input [3:0] B,
    output EQ,
    output LT,
    output GT
    );

    reg eq,lt,gt;

    always @ (A,B)
    begin
        if(A == B) begin
            eq = 1'b1;
            lt = 1'b0;
            gt = 1'b0;
            end
        if(A < B) begin
            eq = 1'b0;
            lt = 1'b1;
            gt = 1'b0;
            end
        if(A > B) begin
            eq = 1'b0;
            lt = 1'b0;
            gt = 1'b1;
            end
    end
    assign EQ = eq;
    assign LT = lt;
    assign GT = gt;
endmodule
```

Figures above: 32-Bit Comparator (left) 4-Bit Comparator (right)