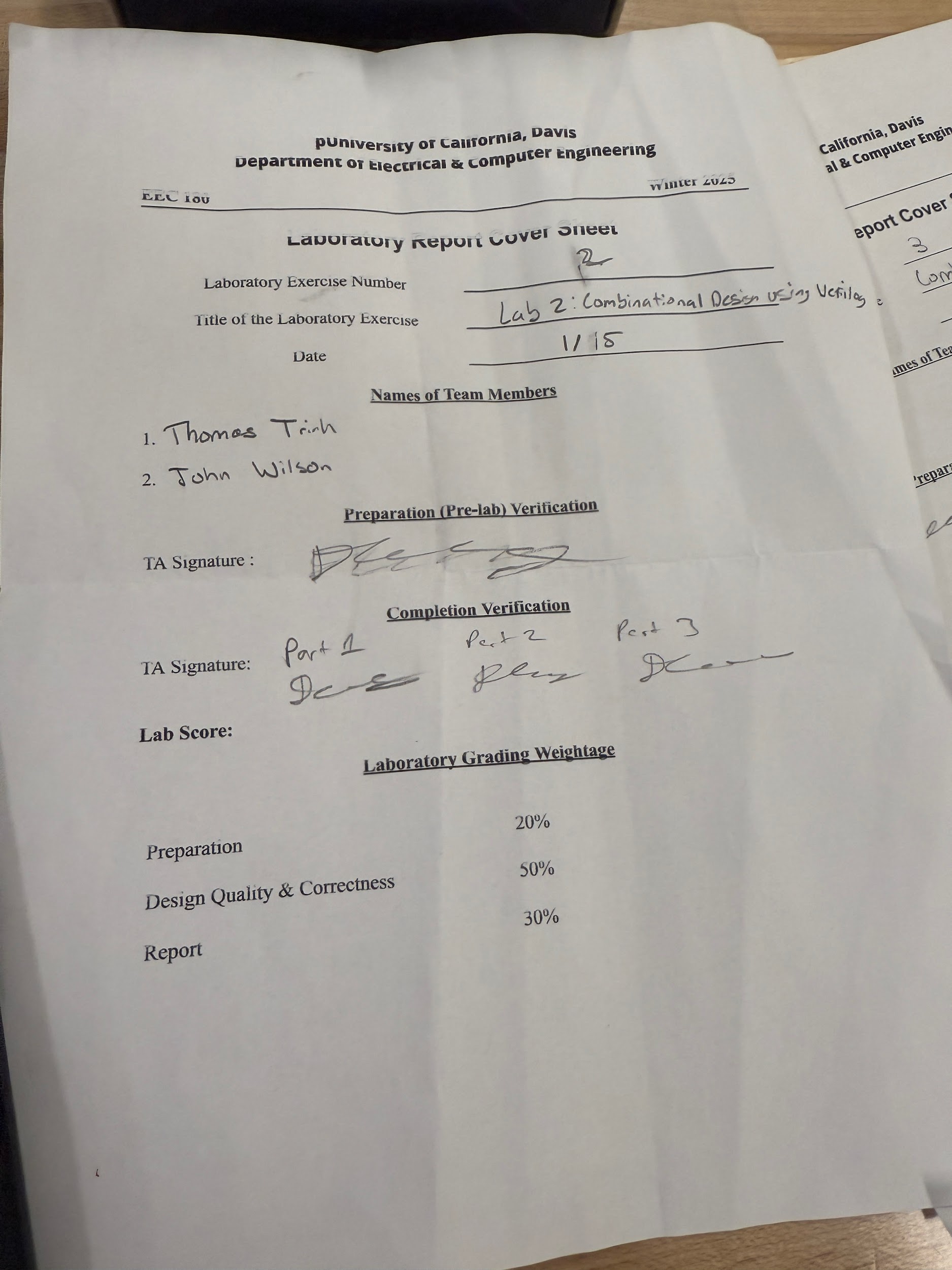
****

**LAB 2: Combinational Logic Design Using Verilog**

**EEC180**

**Prof. E. Akella**

**WQ2025**

**Thomas Trinh**

**John Wilson**

**Objective:**

The objective of this lab is to design combinational arithmetic circuits using Verilog. In part 1 of 3, an 8 bit ripple carry adder is designed. In part 2, a 4x4 unsigned array multiplier will be designed. Lastly, in part 3, Verilog parameters will be used to design a k-bit ripple carry adder. Each part involves designing self-checking testbenches.

**Prelab - Behavioral vs Structural RTL**

1. ***What is the difference between behavioral and structural HDL code?***

The difference between behavioral and structural HDL code is that structural code is just the primitive gates and wires while behavioral code is the procedure/code that you want to happen (not connecting gates).

1. ***Provide one small example (10 lines of code maximum) of a behavioral RTL module, and one small example of a structural RTL module.***

*Structural:*

module example1 (x1, x2, x3, f) (

input x1, x2, x3;

output f;

);

and and1 (g, x1, x2);

not not 1 (k, x2);

and and2 (h, k, x3);

or or1 (f, g, h);

endmodule

*Behavioral:*

module example3 (x1, x2, x3, f) (

input x1, x2, x3;

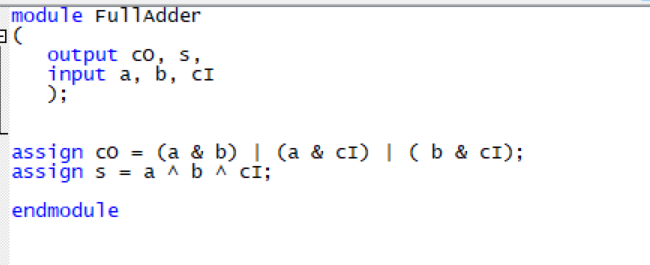
output f ;

);

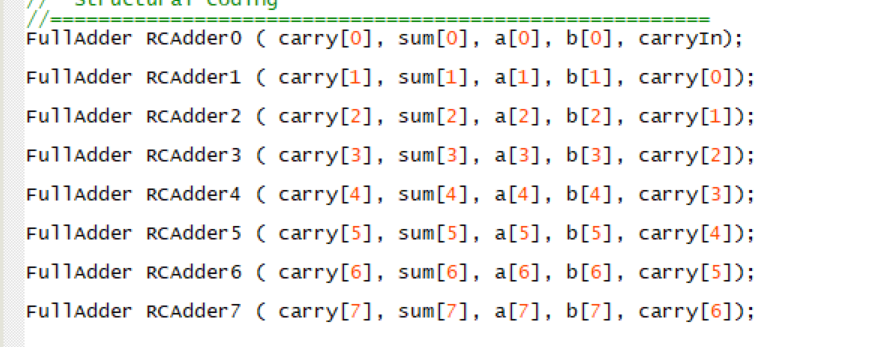
assign f = (x1 & x2) | (~x2 & x3);

endmodule

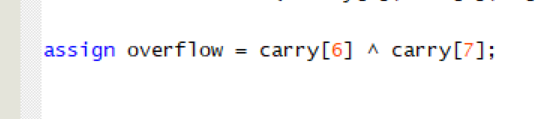
**Part 1 - Ripple-carry Adder**

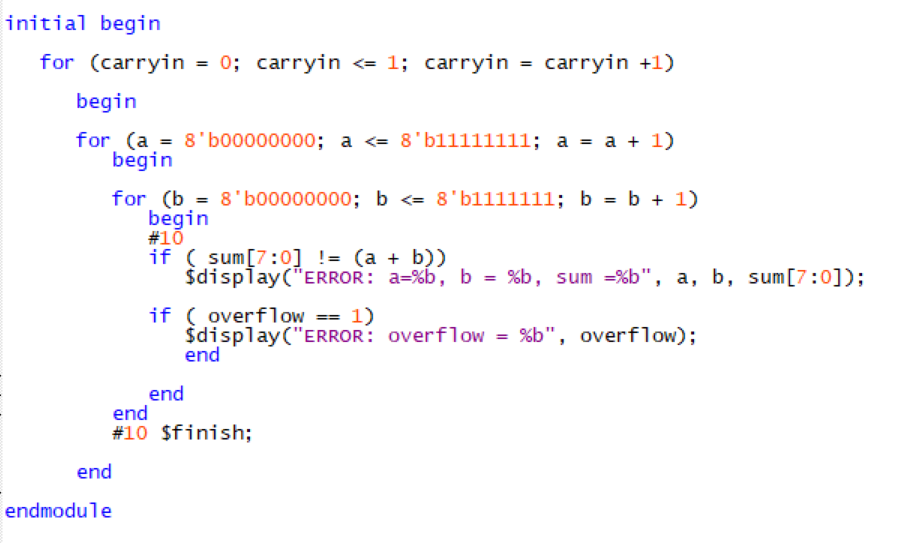
The first subsection of part 1 requires a behavioral model of a full adder to be designed in Verilog. In order to achieve this, the equations for carry out ( a & b | a & c\_i | b & c\_i) and sum (a ⊕ b ⊕ c\_i) were used. 

As seen in the behavioral model to the right, instead of individual gates, assign statements are used in order to achieve full adder logic.

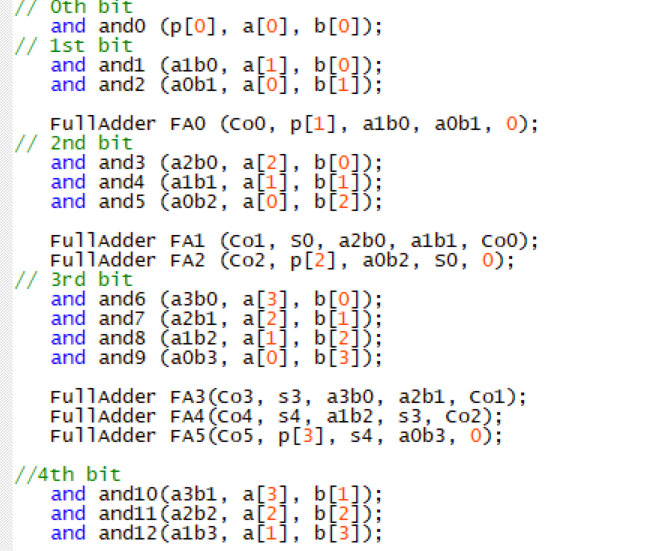


In the second subsection, an 8 bit ripple carry adder is realized in the top level file by instantiating eight different full adders and connecting them in the ripple carry adder order as seen to the right. Each full adder has a carry in, carry out, sum, A, and B input. For the 8 bit ripple carry design, adders 1 through 7 have carry in inputs connected to the previous adder’s carry out.. The 0th adder has a carry in that is set to 0.

Finally, logic is implemented to handle cases of overflow which can be detected if the last two carry out values are not the same, as seen to the right. This logic compares the carry outs of the last two adders and uses a NOR gate to check if the two values are the same.

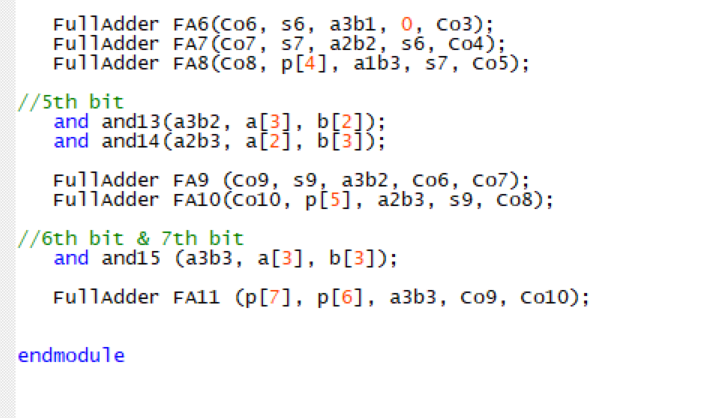


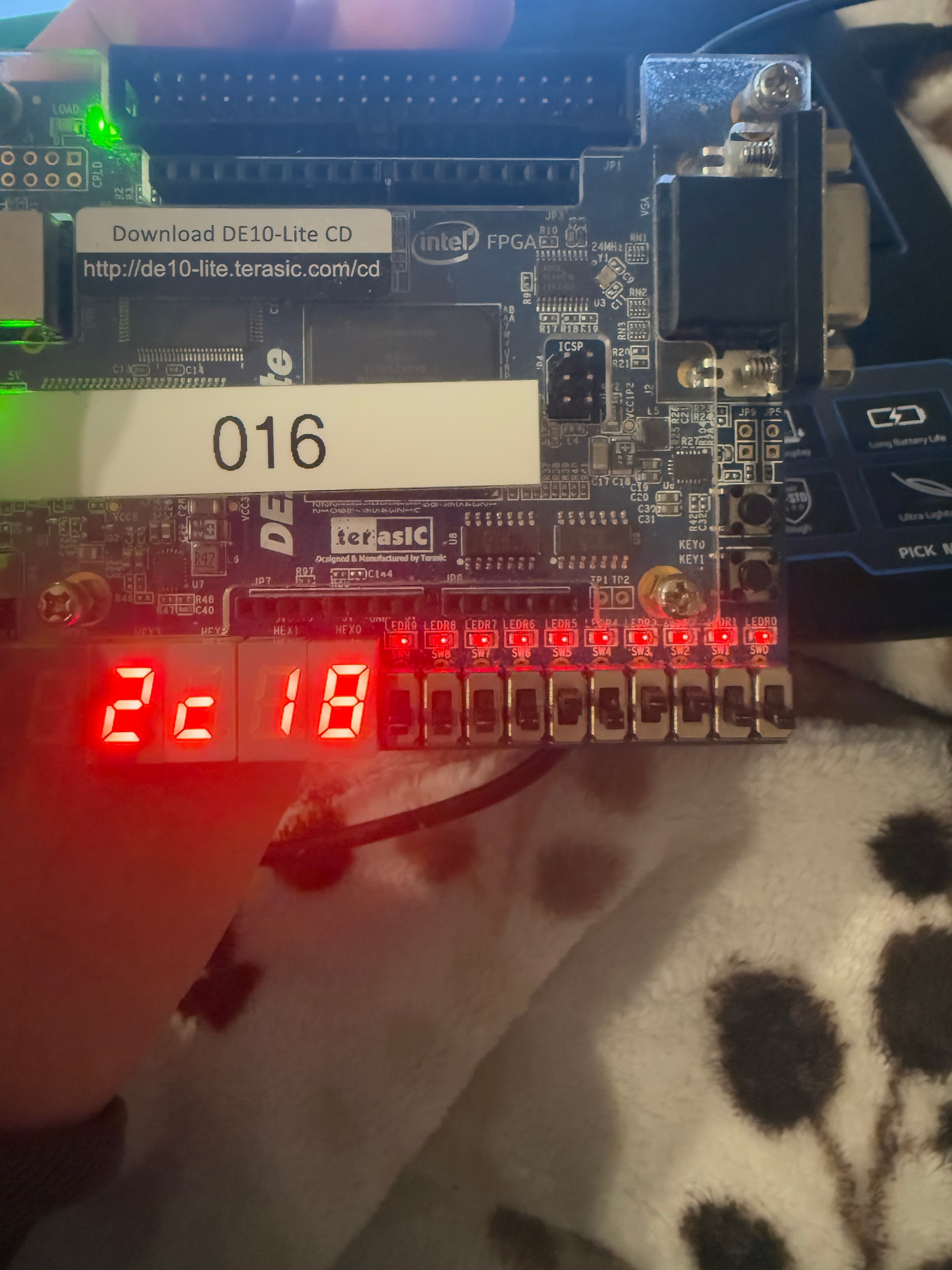
In the third and last subsection which covers the self checking testbench, two for loops are used to iterate through every possible value of A and B. As A and B iterate, the testbench checks the output of the design with the equation A + B. If the output and the sum differ, an error message displays which two inputs of A and B result in the error. The testbench also outputs when an overflow occurs by checking if the overflow output is equal to 1.

**Part 2 - Multiplication**

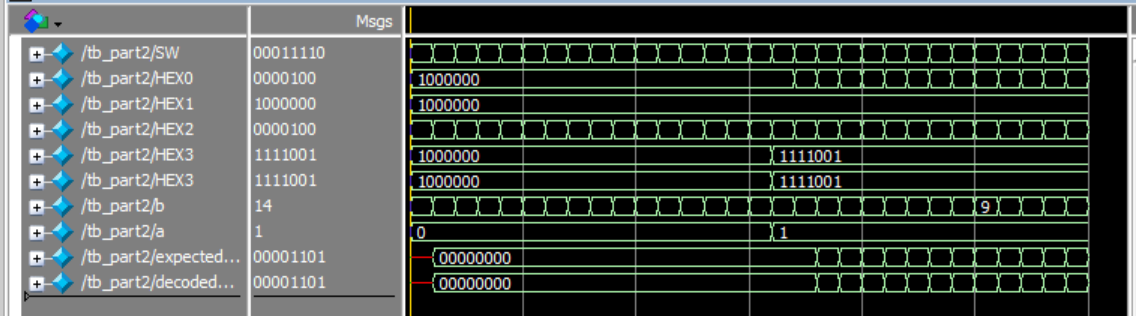
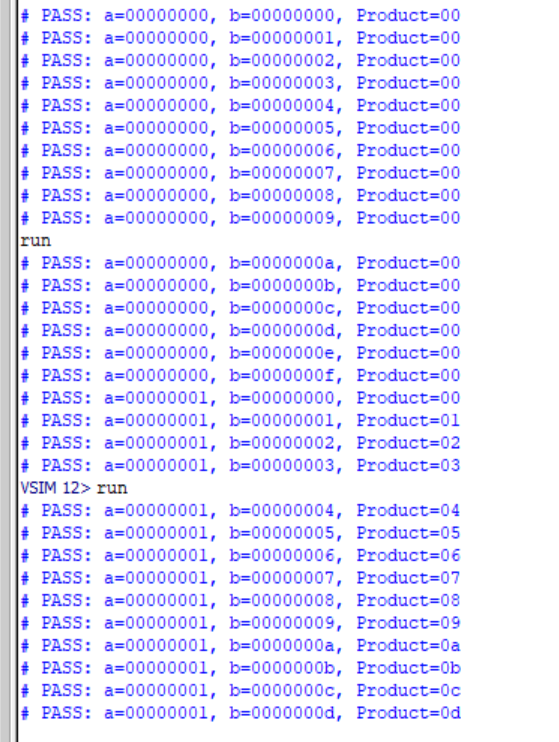
The first subsection of part 2 involved writing a structural design of a 4x4 unsigned array multiplier and compiling it for the DE-10 Board.

For this design, a structural design is required instead of a behavioral one. This involves using keywords like and(), not(), and or(). Using these keywords and specific inputs and outputs allow for specific logic to be designed as seen to the right. The design of the multiplier is followed from the lab and requires several AND gates and full adders. A structural model approach for implementing the multiplier improves readability as to where each input and output are going. As seen to the right, many outputs of several gates and full adders serve as inputs to other gates and adders.

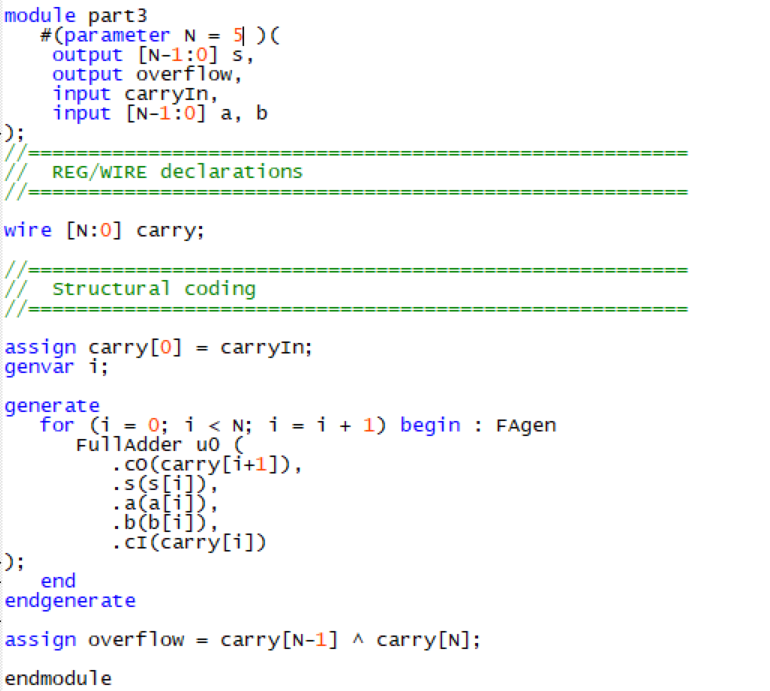




The images above and bottom right, show the compiled design on the DE-10 board. The image above shows the multiplication of 6x4 and the corresponding hexadecimal value of 18. Converting this to decimal (16 + 8) results in 24. The other example in the image to the right multiplies 2 with C (or 12) which results in a hexadecimal value of 18 as well. Using hexadecimal is important because multiplying 2 four bit values can result in 3 digit decimal numbers, but with hexadecimal, they can be represented in only 2 bits.

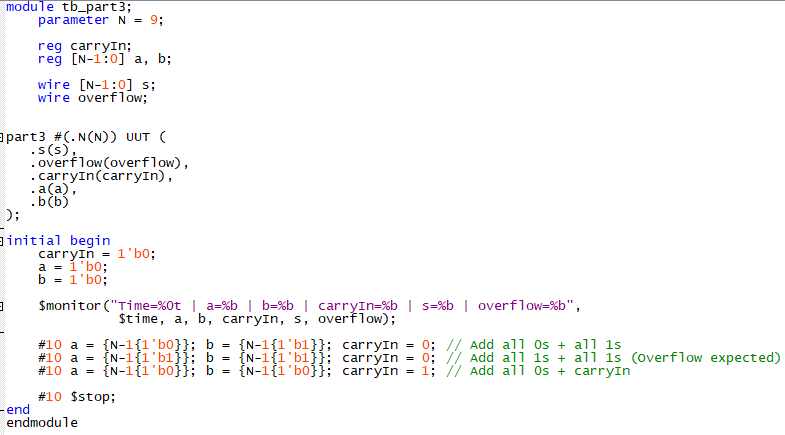


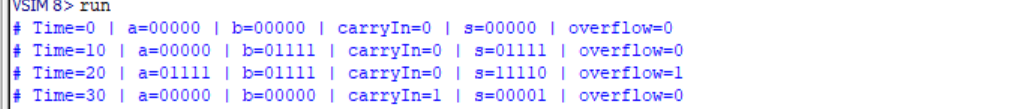
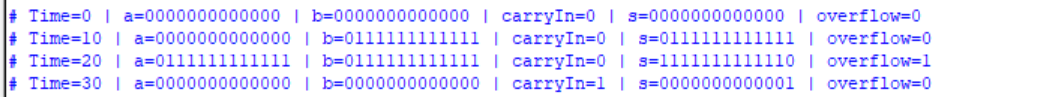
The testbench for part 2 is self checking as well. The approach is similar to the previous test bench, in that it uses two for loops to iterate through the A and B inputs. However, this test bench uses a helper function to decode the hex display values back into usable binary values. The testbench compares these values with the expected product of A and B. If the values are not equal, the testbench displays an error message that displays the expected product and the product from the hex displays. If the values are the same it displays a “Pass” statement.

**Part 3 - Generic Adder**

Part 3 of this lab involved designing a generic adder. In order to do so, the use of parameters and generate statements were required. The parameter value k is arbitrarily set and through the generate statement, connects k full adders to yield a k-bit ripple carry adder, as seen to the right.

The testbench as seen below allows the parameter controlling the number of bits to be changed with the variable N. The fixed test cases adjust for the appropriate bit width with the multiplied concatenate statements, ex. {N-1{1’b0}}.



Sample testbench output shows that changing the parameter to 13 results in a 13 bit ripple carry adder. Changing the parameter to 5 results in a 5 bit ripple carry adder.

**Conclusion**

This lab highlighted the usefulness of self checking testbenches and parameterized modules. Self checking testbenches show promise of being very useful in the future as projects that require a larger number of inputs would require an impractical amount of labor to test manually and self checking testbenches allows for quicker debugging and testing. Parameterized modules would be helpful in future as modules that have already been created could be reused for varying requirements as the parameters allow for flexibility.

**Statement of contribution**

Both Thomas and John worked on designing part 1 and the testbench for part1. Thomas worked on the design for part 2 and the testbench for part2. John worked part 3 and the testbench for part3.