University of California, Davis
Department of Electrical and Computer Engineering

EEC 180B

Spring Quarter 2015

LAB 2: Combinational Logic Design Using Verilog

Objective: The purpose of this lab is to use Verilog to design combinational arithmetic circuits. You will also learn how to write self-checking testbenches.

I. Ripple-carry Adder

A *full adder* (FA) has inputs a, b and c_i (carry in) and produces outputs s (sum) and c_0 (carry out).

$$c_0 = a.b + a.\ c_i + b.c_i$$
$$s = a \oplus b \oplus c_i$$

An n-bit ripple-carry adder can be designed by connecting full-adders in a chain with the carry in for a given stage being the carry out of the previous stage and the carry in of the least significant bit being a 0.

Perform the following steps:

1. Write a **behavioral** model for a full adder in Verilog. (Hint: an assign statement can describe each output equation.)

2. Instantiate your full adder subcircuit in order to build an *8-bit ripple-carry adder*. Add logic to produce an *Overflow* output, which should be set to 1 whenever the sum produced by the adder does not provide the correct signed (twos complement) value.

3. Write a testbench to simulate your design for all possible input combinations. Note that for an 8-bit adder there are 2^{16} (256 x 256) possible inputs. See Appendix at the end of this document for an example of how to construct a testbench using a **for loop** in Verilog.

II. Multiplication

Figure 2 shows the traditional procedure for performing the multiplication P=A x B, where A and B are 4-bit unsigned binary numbers. Since each bit in B is either 1 or 0, the summands are either shifted versions of A or 0000. The Boolean AND operation can be

used to multiply any two binary bits. Figure 3 shows an array multiplier circuit that implements P = A x B, where A and B are 4-bit unsigned binary numbers.

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

Figure 2. Multiplication of Unsigned Binary Numbers

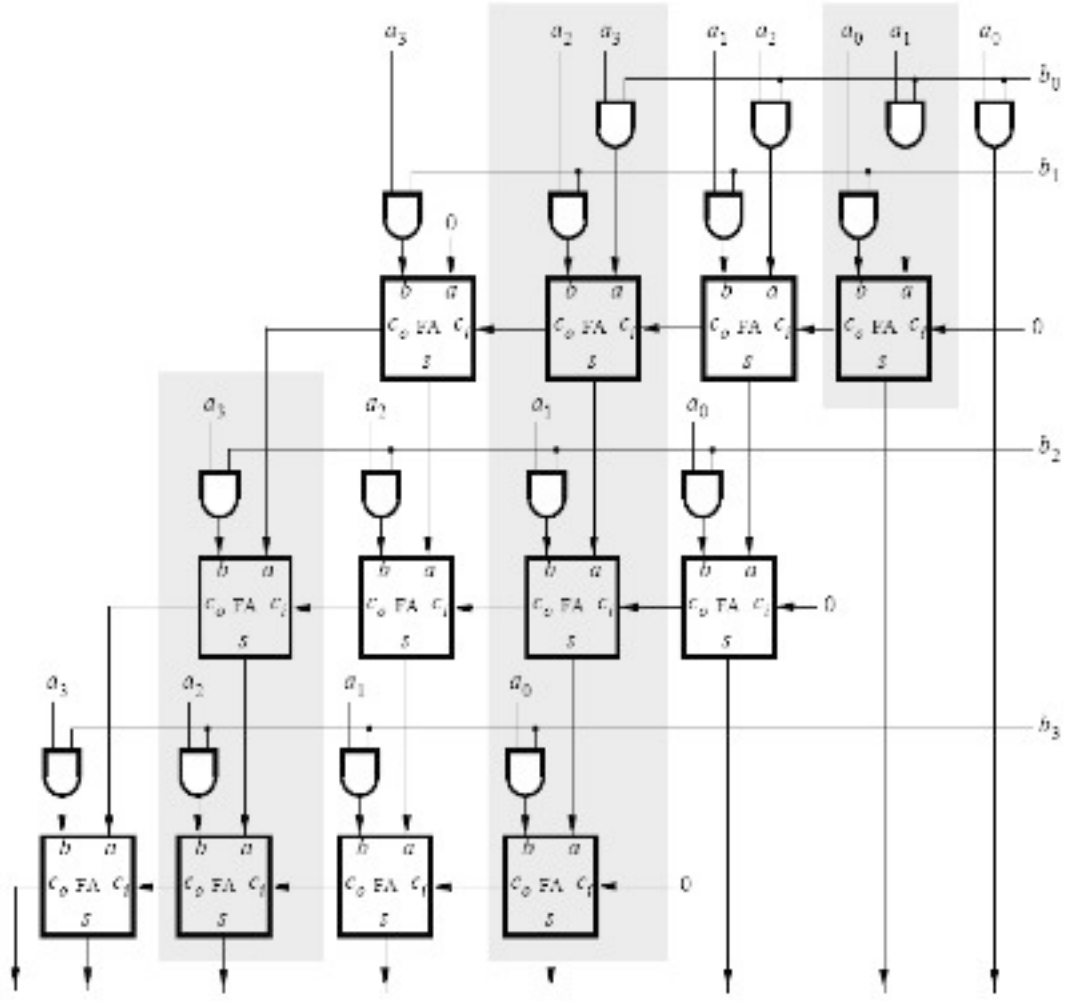

Figure 3. Array Multiplier Circuit Block Diagram

Perform the following steps:

1. Create a **structural** model in Verilog that describes an 8x8 unsigned array multiplier that can be implemented on the Altera DE2 board. Use switches SW_{15-8} to represent the number A and switches SW_{7-0} to represent the number B. Display the hex value of A on HEX7-6 and the hex value of B on HEX5-4. Display the result P = A x B on HEX3-0.

2. Perform a functional simulation of your design and print the simulation results.

3. Compile your design in Quartus. Download your design to the Altera DE2 board and test your circuit. **Demonstrate your circuit to your TA.**

4. Estimate the performance of your circuit in terms of the critical path using the timing analysis too

III. **Lab Report**

For your lab report, include the following:

- Lab Cover Sheet with signed TA verification for successful simulations and implementation in the Altera board.
- Design and implementation of the overflow circuit in part I of the lab.
- Complete Verilog test bench for each part. The testbench should test your design exhaustively, i.e., for all possible input combinations and should be self-checking as shown in the appendix.

Grading Guidelines

Part 1 – 50 points
Part 2 – 50 points

You are supposed to work alone in this lab. Any unauthorized collaboration will be reported to SJA.

IV. Appendix: Writing a Self Checking Testbench

Here we are writing a testbench to test a half adder. Note that half adder has 2 inputs, so for exhaustive verification we have to check the output for all the four input combinations, which is being done by the for loop.

The testbench is self-checking – it examines the outputs and prints out an error message if the output is wrong.

Cut and paste this in http://www.iverilog.com

*Obviously you should not see an errors because the design for the half adder is correct. Now, change the **xor** operation to an **or** and re-run the simulation. You should see an error.*

```
module halfAdd (sum, cOut, a, b);
        output  sum, cOut;
        input   a, b;

        xor             (sum, a, b);
        and             (cOut, a, b);
endmodule


module tBench;
        wire    sum, co;
        reg [2:0] test;

        // design under test;
 halfAdd        HA (sum, co, test[1], test[0]);

        // stimulus and verification that the output is correct
initial begin
        for (test = 0; test < 4; test = test + 1)
        begin
          #10;
            if ({co, sum} !=   (test[1] + test[0]) )
              $display("ERROR:  a=%b b=%b sum=%b cout=%b", test[1], test[0], sum, co);
            end
          #10 $finish;
         end

endmodule
```
