Brodric Young

ECEN 340

Lab #5

Verilog Combinational Logic 8-bit Multiplier

Purpose:

1. To become familiar with Verilog “assign” statements or “always” blocks to implement combinational logic.
2. To become more familiar with Verilog operators
3. To learn the techniques required to implement a simple binary Multiplier

Procedure:

Part 1: Design a unipolar binary multiplier using Verilog “assign” statements or “always” blocks. To build the multiplier, you will need to use operators like the “&”, “+” and the “{}” operators, but you will not be using use the “*” operator.

The inputs to the multiplier will be the 16 DIP switches (8 for the multiplicand and 8 for the multiplier). The bank of 16 LEDs will serve to display the result.

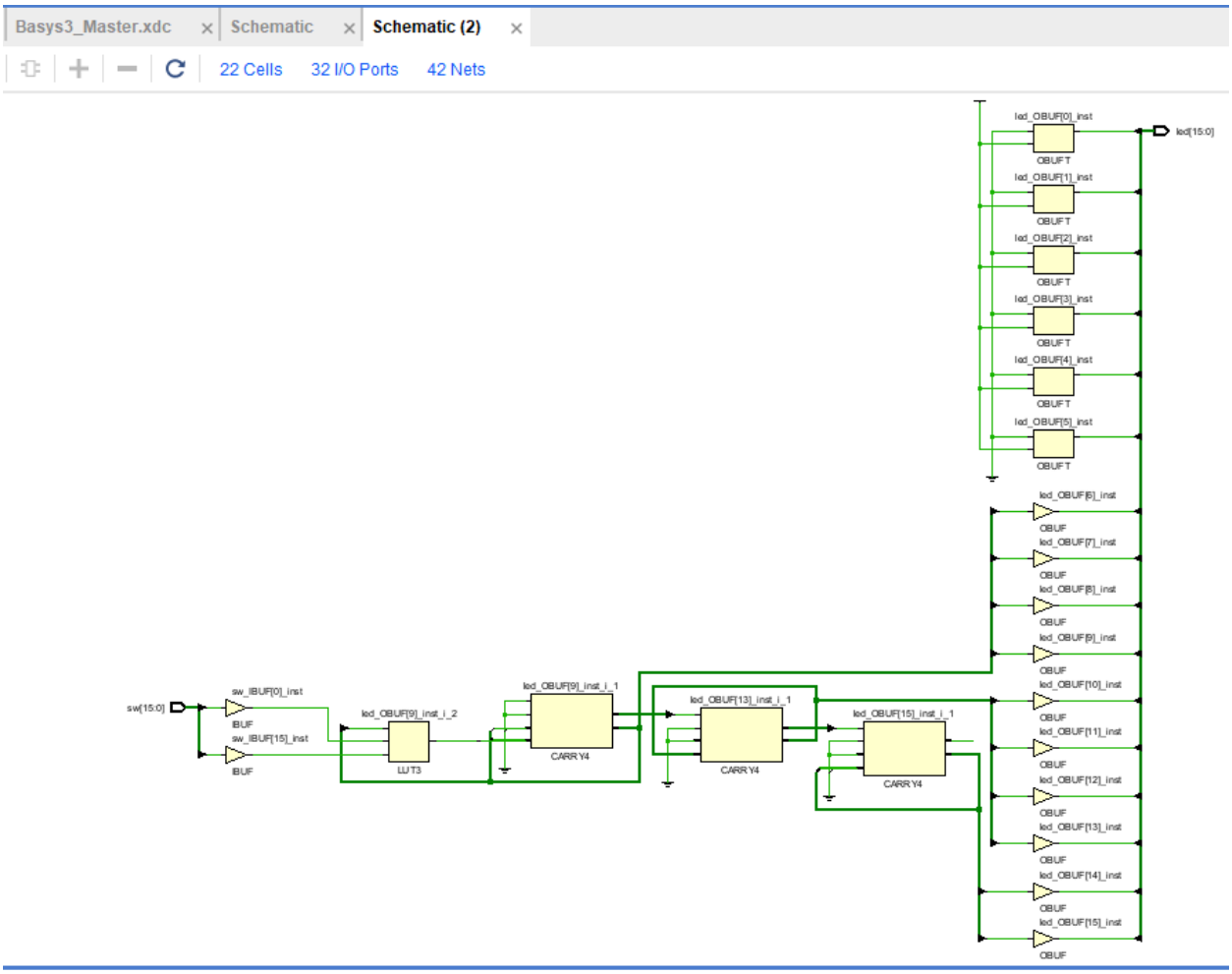
```
`timescale 1ns / 1ps
// multiplies two 8 bit numbers input for 16 switches
// displays result on the 16 leds
module multiplier(
    input [15:0] sw,
    output reg [15:0] led
    );

    reg[7:0] x;
    reg[7:0] y;

    reg[15:0] productPart[7:0]; // products to be summed together
    reg[15:0] productFinal; // final product

    integer i;
    always @(*) begin
        x = sw[7:0]; // x = first half of switches
        y = sw[15:8]; // y = second half of switches
        productFinal = 16'b0;

        for(i=0; i<8; i=i+1) begin // loop through indexes 0 to 7
(8 bits of number)
            if (y[i]) // if y[i] is a 1
```

```
                productPart[i] = (x & 8'b11111111) << i; // copy
    x
            else
                productPart[i] = 8'b0; // x is 0
        end

        for(i=0; i<8; i=i+1) begin // loop through indexes 0 to 7
    (8 product parts)
            productFinal = productFinal + productPart[i]; // sum
    all product parts
        end

        led = productFinal; //display final product
    end
endmodule
```

Part 2: Adapt part 1 to produce a two's complement signed output.

```
`timescale 1ns / 1ps
// multiplies two 8 bit numbers input for 16 switches
// displays result on the 16 leds
module multiplier(
    input [15:0] sw,
    output reg [15:0] led
    );

    reg[7:0] x;
    reg[7:0] y;

    reg[7:0] productPart[15:0]; // products to be summed together
    reg[15:0] productFinal; // final product

    integer i;
    always @(*) begin
        x = sw[7:0]; // x = first half of switches
        y = sw[15:8]; // y = second half of switches
        productFinal <= 16'b0;

        for(i=0; i<8; i=i+1) begin // loop through indexes 0 to 7 (8
bits of number)
            if (y[i]) // if y[i] is a 1
                productPart[i] = ((x & 8'b11111111) << 8) >> (8-i); //
copy x with sign extended to MSB
            else
                productPart[i] = 16'b0; // x is 0
        end

        for(i=0; i<8; i=i+1) begin // loop through indexes 0 to 7 (8
product parts)
```

```
            if(productPart[i][15]) // if product part i has a sign bit
of 1
                productPart[i] = ~productPart[i] + 1;
            productFinal = productFinal + productPart[i]; // sum all
product parts
        end

        led = productFinal; //display final product
    end
endmodule
```



Conclusion:

In this lab, I made a multiplier without using the "*" symbol in Verilog. In the first part, I used a for loop to loop through each bit in one of the numbers and depending on if it was a 1 or 0, use the "&" operator and an all 1's bitmask to copy the same bits of the other number and the "≪" operator to shift it. Then I could sum up all the parts after the for loop finished to produce the final product. The second part I added some additional functionality to handle negative numbers, using a condition to tell if the sign bit was a 1 to flip the bits and add 1 to the number which converted it to it's two's compliment. I tested the first part and verified it worked correctly before moving on the second part and testing it. To test, I programmed the FPGA and input two numbers and verified the LED's showed the correct output multiple times. In the end, both part 1 and part 2 worked 100% correctly.