**Lab 5: Verilog Combinational Logic 8-bit Multiplier**

Author: Seth Ricks

Date: 10/15/2025

Class: ECEN 340

**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

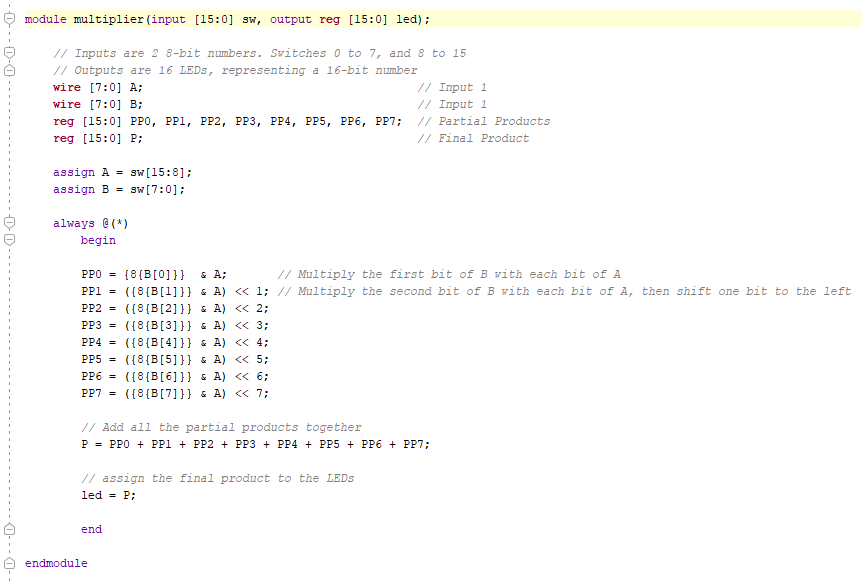
**Procedure:**

1. Design a unipolar binary multiplier using Verilog.
2. Adapt part 1 to produce a two’s complement signed output.

**Results/Report:**

Part 1:

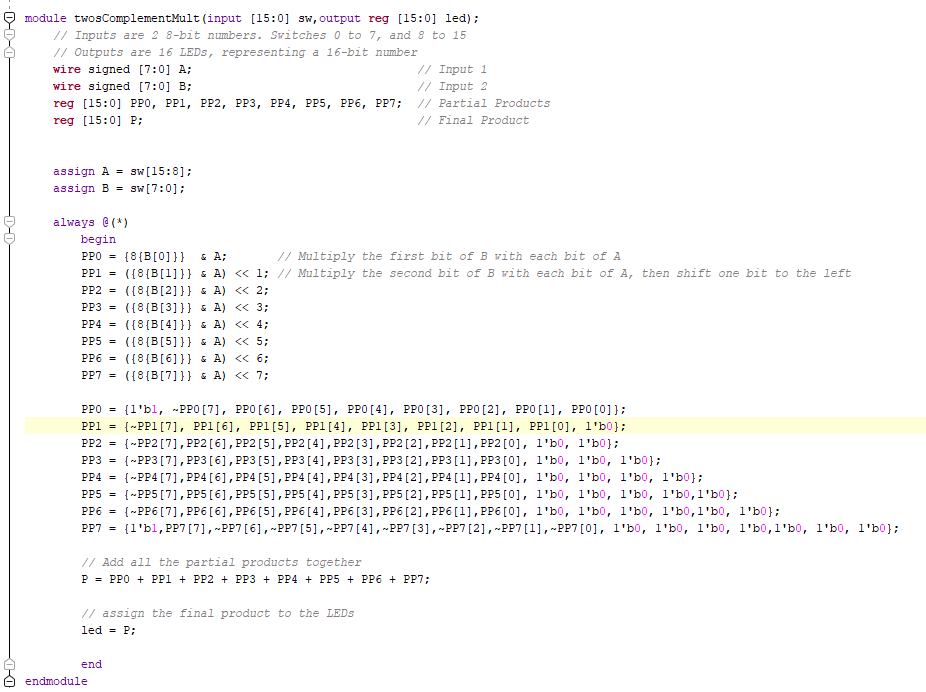
In this portion of the lab, we were tasked with creating a binary multiplier using Verilog. We weren’t allowed to use the \* operator. Instead, we were asked to make it out of partial products logic. My partner and I were able to do this successfully, and an image of our code for this portion of the lab can be seen in **Figure 1** below.



Figure

The trickiest part about writing this code was figuring out how to AND each bit of the B value with the entire value of A, like is done in regular decimal multiplication. We initially were going to try to do a for loop, which would’ve iterated through each bit of A and multiplied it by each bit of B. But this proved to be too complicated, especially with 7 partial products. What we ended up doing instead was using the replication operator {{}}, which allowed us to duplicated a bit of B into an 8 bit binary number. We then multiplied the entire number by A, shifted it accordingly, and stored them into partial products. We added all the partial products together to get the final product, and stored this final product in the LEDs to display.

Part 2:

This part of the lab was optional extra credit, in which we were asked to edit the above code in part one to allow for signed integers. This proved to be harder than expected, but we were able to do so effectively. An image of our code can be seen in **Figure 2** below. 

Figure

Unlike the previous code, we couldn’t add the partial products right away. First, we had to change the partial products bit by bit to comply with two’s complement. This was given to us, and is outside of the scope of this report. The hardest part about writing this code was small errors we made, such as keeping a PP2 in the PP3 line. But after reviewing our statements multiple times, the code worked as desired. We tested it directly on the FPGA board, using simple values such as 1 x -3 = -3. An image of this test can be seen in **Figure 3** below.

A close-up of a circuit board

AI-generated content may be incorrect.

Figure

**Conclusion:**

The most difficult part about this lab was understanding the task. We are so used to plugging in the multiplication operator that we forget what is happening behind the scenes. This was useful to remember and understand how binary multiplication works, along with how to use it with negative numbers with two’s complement. My partner and I now have a better understanding of blocking vs. non-blocking assignments, and how they can change our hardware.