EE108 Lab 1 Extension

Multiplication, The Hard Way

Lab Due: **11:59pm 10/1/20**

1. Introduction

So, what are we doing here?

This lab is a brief addition to the usual Lab 1, designed to give you some more experience writing combinational verilog modules. In this Lab, you will be designing a N-Bit Multiplication module using only basic logic gates. This module will be parameterized to accept two unsigned inputs of the same length and output the product as an unsigned wire of twice the length of the input.

The Rules

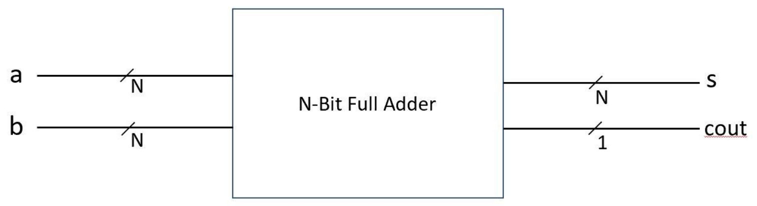
Let's start with the obvious, you cannot use the built-in multiplication (*) operator in verilog or the addition (+) operator. Using either would defeat the purpose of this lab. Instead, we are building a circuit which multiplies its inputs from the ground up. That is, using only the basic logic operators, AND (&), NAND (~&), OR(|), NOR (~|), XOR (^), XNOR (~^), and NOT (~). This should give you a much better working understanding of what's happening in your computer literally millions of times each second.

Where to start? Addition.

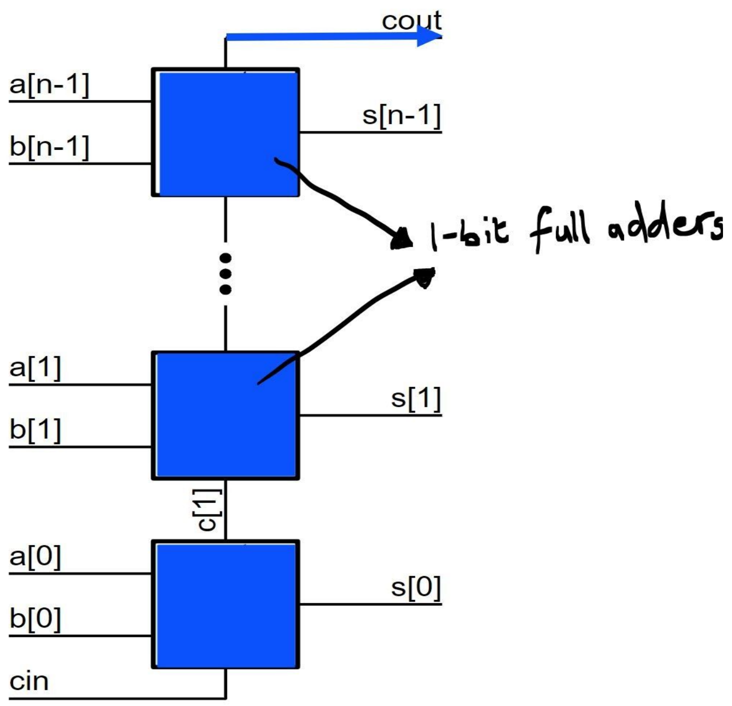
Remember when you learned in elementary school that multiplication is just repeated addition? That will come in handy now. You're going to need to implement a full adder for the partial sums needed for the multiplication module.

N-Bit Full Adder

There are different ways to implement an n-bit full adder but in the context of what we treated in class, our n bit full adder will comprise a series of one bit full adders performing bitwise addition on its operands. The N-Bit Full Adder on a high level is represented by this block:



It takes in 2 n-bit wide inputs and produces a result (also n-bits wide) and a carry bit to indicate whether there was an overflow in the addition. A detailed block level view of the N-bit full adder, showing the series of 1-bit full adders is shown below:



Addition, repeated

Now that we know how to add two numbers, how do we do multiplication? We'll break it down into a series of repeated addition steps, just like in elementary. The good news is that binary makes this actually very easy. Imagine we're multiplying two 4-bit inputs, 4 and 3.

$$0100 \ * \ 0011 = \ 1100$$

Did you see what happened there? Multiplying by 4 effectively shifted the binary representation of 3 to the left by 2. Shifting left is the same as multiplying by 2^n, where n is the number of bits shifted. So, what does each bit of one of the inputs represent? It represents whether or not the other input shifted to the same position should be added to the final product. Take 5 times 7, for example,

$$0101 \ * \ 0111 \ = \ 0111 * 0001 \ + \ 0111 \ * \ 0100 \ = 0111 + \ 011100 \ = \ 00100011$$

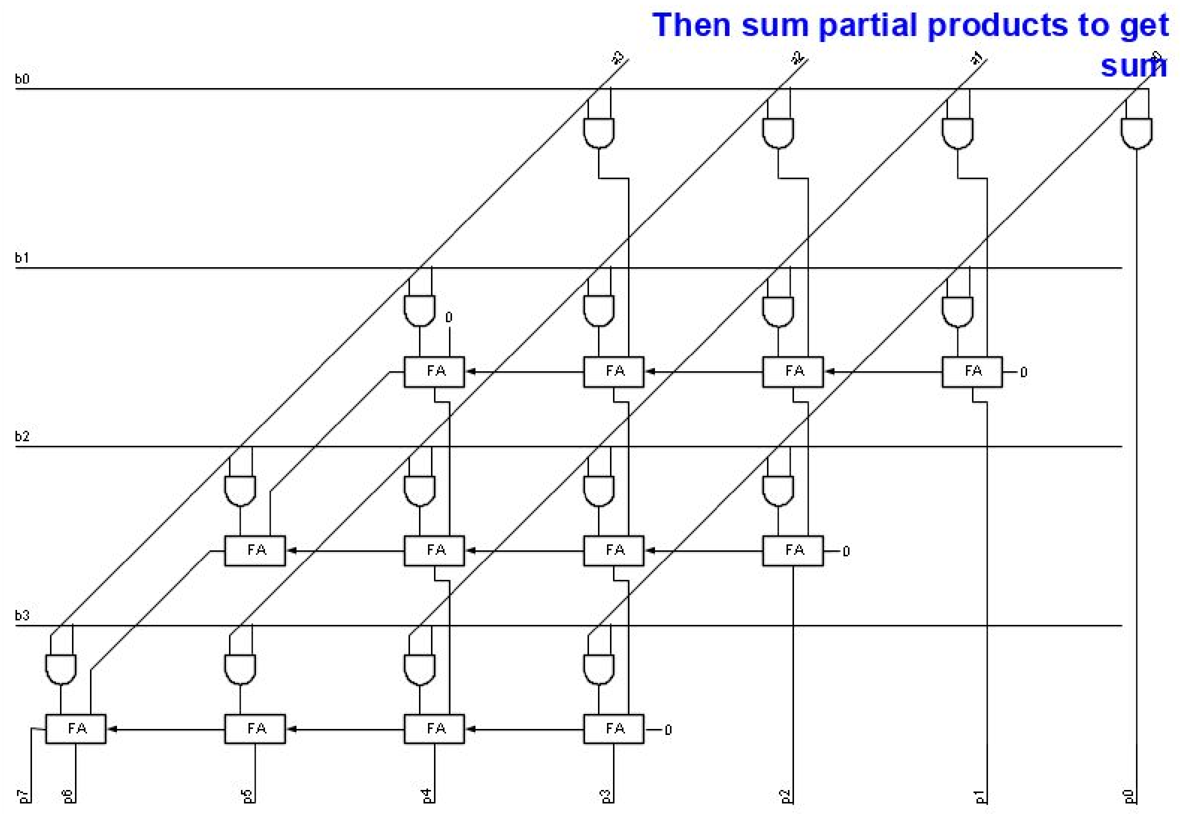
Or, another way of looking at it:

$$7 * 5 \ = \ (7 << 3)\&5_4 \ + \ (7 << 2)\&5_3 \ + \ (7 << 1)\&5_2 \ + \ (7 << 0)\&5_1$$

Here, $(7 \ll n-1)\&5_n$ is the n^{th} bit in the binary representation of 5 ANDed with each bit in the left shifted 7. We call these ***partial products,*** because they represent 7 multiplied with each bit in 5. To build a multiplication block, all we need to do is find these partial products and sum them all together. Note that, since we don't know ahead of time how many bits are 1 and not 0, we need an adder block for each potential partial sum.

Also, notice that our output is now 8 bits long. This is because, in the general case, multiplication will yield a number that is lengths of the input added together. For this lab, we will always have our inputs be the same length, so the output will be twice the length of the inputs.

Below is a diagram from Lecture 4 of a multiplication circuit. Each row represents shifting, then ANDing the 4-bit number "**a**" with one bit in "**b**". Thus, each row represents a partial sum! All that is left to do is sum the partials together using adders, and you have a multiplier circuit!



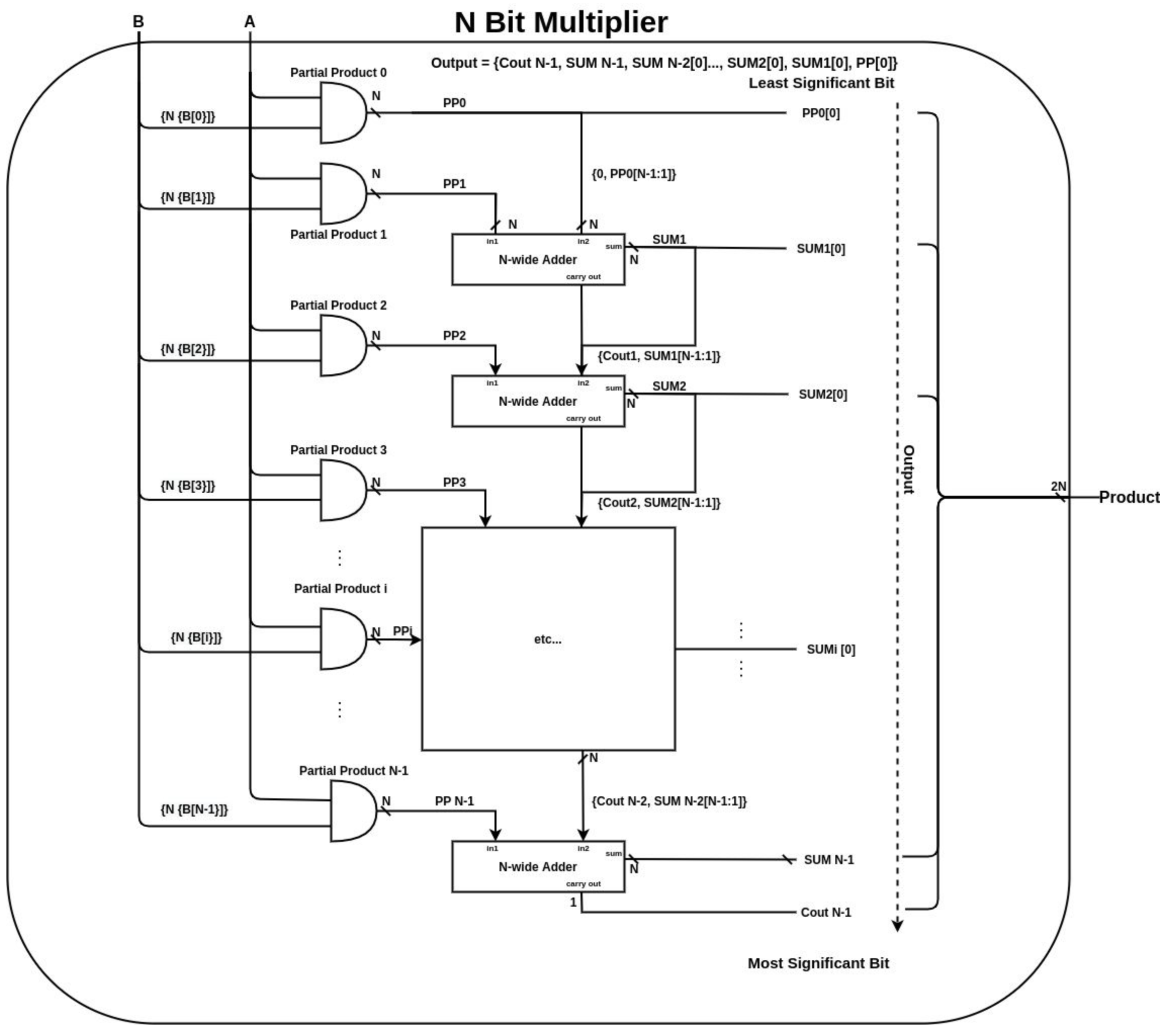
One more note: To use as small width adders as possible, we note that each stage of addition is only 4 bits wide (the length of the inputs). To use these smaller adders, we have to "shift" the adders with the partial products. To do this, we have to note that the full adders essentially output a n+1 bit output if you include the overflow/carry output. At each stage of addition, all we have to do is take the top four bits of this output, and output the least significant bit as the n^{th} bit of the product. That is why you see the Full Adder (FA) block above shifting to the left with each row.

2. Implementation Overview

Just like in Lab 1, we're going to walk through the design of each module together. We're providing you with working adders, so all you need to do is implement the N-Bit Multiplier module using partial products and adders chained together. We've already translated the Multiplier circuit above into a block diagram below.

Each row of AND gates from the previous section is translated into an N wide AND gate to form the partial products. Pay attention to how each adder is connected. Their inputs are the next partial sum and the last adder's carry bit concatenated with its top 3 sum bits. The output is the concatenation of the least significant bits of each adder stage's output and the final sum and carry bit. That's all there is, but making it modular will be tricky.

The goal with this lab is to giver you experience translating parameterized block diagrams into Verilog code, but pay attention to how we're organizing things at the block diagram level since in lab 2 you'll have to build the whole system from scratch.



Make a Vivado project

Create a new, empty Vivado project named 'lab1_extension'.

The starter code is located on Canvas. Add the starter code to your project the same way you did in lab 0, using the “Add Sources” tab on the far left.

1) First, add all the design sources using “Add Sources -> Add or create **design sources** -> Add files.” Select all the files that DON’T end in “_tb.v” and add them.
2) Repeat step 1, but select “Add or create **simulation sources**” instead. Select all the files that DO end in “_tb.v” and add them.
3) Repeat step 1, but for the **constraint source** (lab1_extension.xdc).

Next, set up your github repository. As you may have seen in lab 0, Vivado creates a lot of folders and log files when simulating and synthesizing. To keep it simple, just track the starter files (the .v and .xdc files). You may also find it helpful to track your .bit file (**if you happen to have a lab kit and are planning on running your bit file on the FPGA**), which will be located in lab1_extension -> lab1_extension.runs -> impl_1 -> lab1_extension_top.bit once it is generated. Now you can proceed to modify all the incomplete modules and their corresponding testbenches.

Testing as we go along

Similar to the process you went through with lab1, we strongly recommend that you test the full adder to make sure that it is functional and produces expected results for all possible ranges of test cases. This will save you time as you work upwards to define your multiplication module.

To demonstrate that you understand the complete flow of test benching, we didn’t provide any starter code for the multiplication module testbench so you’d have to do that from scratch. Test benches have been provided for the 1-bit and n-bit full adders but feel free to modify and play around with that if you want.

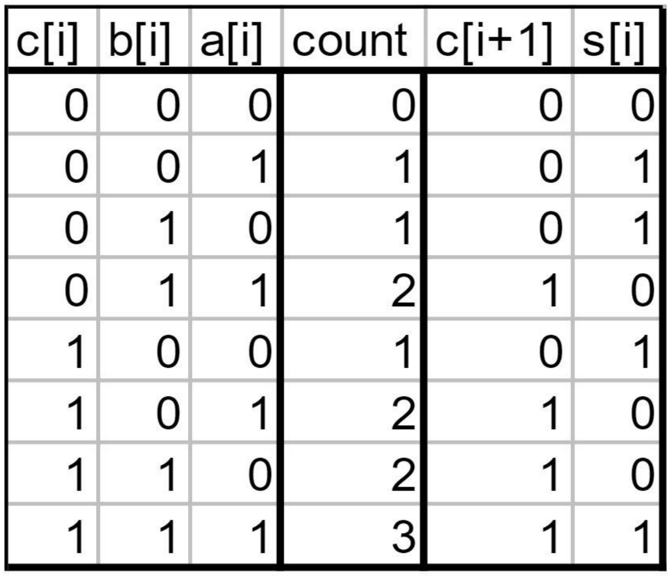
Also, recall from lab 0 that you can run a testbench by going to ‘Simulation Sources’ in the ‘Sources’ pane, then setting the desired testbench to run as “top.” Then, click “Run Simulation” on the far left pane. **After you finish each module, you should run its testbench (adding test cases if necessary) and ensure it behaves correctly.**

3. Implementation

1-Bit Full Adder

We provide you with a functional 1-bit full adder as a starter code for this lab so there is no need to worry about the implementation or the testbenching for this module. The 1-bit full adder takes in 3 1-bit inputs, 2 of which denote the nth bit of the addition operands and the last indicates the carry bit (overflow bit). The functionality and implementation details for this module was covered in lectures so you can refer to the lecture slides as a resource for understanding this module.
In essence, it helps to draw a truth table to capture the different combinations of inputs into this module and what the corresponding outputs should be. From the truth table, it becomes evident that the carry bit is computed using a majority circuit composing the input wires, as shown below. The sum bit, “s”, is finally computed by XORing all the input wires. And again we know that we need to XOR the input wires because of the patterns observed in the truth table for this module.

Truth table for a 1-bit full adder:



c[i]	b[i]	a[i]	count	c[i+1]	s[i]
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Below is the module definition (`one_bit_full_adder.v`):

```
module one_bit_full_adder
               (
                 input wire a, b, cin,
                 output wire s, cout
               );
     //module implementation
     assign s = a ^ b ^ cin;
     assign cout = (a & b) | (a & cin) | (b & cin); //majority
endmodule
```

N-Bit Full Adder

We also provide you with a functional n-bit full adder module for this lab so there is no need to worry about implementation or testbenching for this module. This module takes as input 2 n-bit wide operands (referred to as “a” and “b” in the module definition below) and a carry-in bit (“cin”, which indicates whether there was an overflow from a previous calculation that needs to be taken into consideration; also it could be used as a mechanism for supporting sum of signed numbers as we saw in class). For the purposes of this lab, this extra “cin” bit going into the n-bit full adder module will always be 0. The n-bit full adder outputs an n-bit wide output (labelled “out”) which holds the sum of “a” and “b” and an extra bit named “of” which is used to indicate whether or not the addition of “a” and “b” overflowed.

The module definition uses syntax that we may not have covered in sections or lectures. These constructs are the “generate” and “genvar” keywords and the idea of synthesizable looping constructs. The “generate” keyword is used to denote a block of Verilog code that is used to perform conditional instantiation and/or multiple instantiations of modules. We did mention in sections that we should avoid looping constructs in general because they are not synthesizable. Well there is a small caveat for the “for” loop. A for loop construct can be used inside of a generate block when it is going to run for a definite number of iterations. Verilog only accepts the use of the constants or a special type “genvar” for the iteration variable.

For our n-bit full adder circuit, we use the generate construct to create N different instances of our 1-bit full adder and use that to compute the sum of our n-bit wide addition operands. Note that the for loop syntax is essentially the same as we covered in lectures/sections, the only difference here is that the looping variable must be of type “genvar”.

Below is the module definition for the n-bit full adder (`n_bit_full_adder.v`):

```
module n_bit_full_adder #(parameter n = 4)
               ( input wire [n-1:0] a, b,
                 input wire cin,
                 output wire [n-1:0] out,
                 output wire of );
      //module implementation
      generate
            wire [n:0] carry;
            assign carry[0] = cin;
            assign of = carry[n];
            genvar i;
            for (i = 0; i < n; i = i + 1) begin:
n_bit_full_adder_array
                  onebit_full_adder adder_i (
                        .a(a[i]),
                        .b(b[i]),
                        .cin(carry[i]),
                        .cout(carry[i+1]),
                        .s(out[i])
                  );
            end
      endgenerate
```

```
endmodule
```

N-Bit Multiplier

This module is where the rubber hits the road. You have your n-bit adder, now you need to find a way to modularly create an array of them. You also need to be able to make **N** partial products of **N** length and assign all of their values. Even at the output, we have **N-1** sum least significant bits that need to be assigned bits in the output. In all of these cases, there is special verilog syntax called "**generate**". In our case, **generate** allows you run a for loop to programmatically assign variables and create modules in synthesizable verilog. It's syntax is as follows:

`Generate module`

```
generate
     // Generates n n-bit adders
     genvar i;
     for(i=0; i < n; i=i+1) begin: name_of_module_array
          N_bit_full_adder #() name_of_adder (
          .a(),
          .b(),
          .cin(0),
          .out(),
          .of()
     )
     end

endgenerate
```

`Generate assign`

```
generate
     // assigns array of partial_products to 0
     genvar i;
     for(i=0; i < n; i=i+1) begin: name_of_module_array
          assign partial_product[i] = 4'd0;
     end


endgenerate
```

The other piece of syntax you will need to know before diving into implementation is arrays. You can create an array of wires by placing square brackets after the declaration. To index a wire, you need to use square brackets again. Here's a sample of array syntax:

`Array syntax`

```
wire [3:0] a [0:7]; // create an array of 8 wires of width 4
assign a[1] = 4'b0110; // Set the wire at index 1 in the array to
0110
assign a[3][1:0] = 2'b10; // Set bits 1 and 0 of wire 3 in array to
10
```

The **generate** and **array** syntax fit together perfectly, since for this lab you will be creating an N long array of N bit numbers to represent your partial products. After that you will need to assign each wire in the array of partial products to the first input AND'd with each bit in the second input. (TIP: The bitwise **&** operator needs both inputs to be the same length. Use the notation {N {B[i]}} to make a binary number of N copies of the i^{th} bit in B.)

After you've assigned all of your partial products, you need two arrays to hold your carry bits and the sum produced by each adder. We will have N-1 adders, so we need arrays of at least size N-1 to hold and pass along the outputs of each adder. The carry bits are 1 wide, so we can represent them by a wire of length N-1. The sums are all N width, so we need an array of length N-1 of N wide wires.

Next, you need to do is generate your adders using a generate block. You will need N-1 of them. One input will be the next partial product, and the other will be the carry bit concatenated with the top 3 bits of the previous sum (this represents the top 4 bits of the last adder block). The outputs, sum and overflow, will each be connected to their respective array entry.

The edge case with the adders is that the first adder is only connected to the first two partial products, where the first input is sliced to the top 3 bits and prepended with an extra 0. You can manually instantiate this first module, or you can expand the length of your carry and sum arrays by 1 and set the first value in your sum array to the first partial product, and the first value in the carry array to 0. For an example of the latter, look at our implementation of n_bit_adder.v

After generating your adders, all that is left is to assign the output. The first bit of the output is the least significant bit of the first partial product. The next N-2 bits are the least significant bits from the sum produced by each adder in the sequence. After that, the next N bits are the sum produced by the final adder. At this point, your output array should be length 2N-1. All that is left, is to add the final overflow bit as the most significant bit of the output! You now have a working multiplication unit!

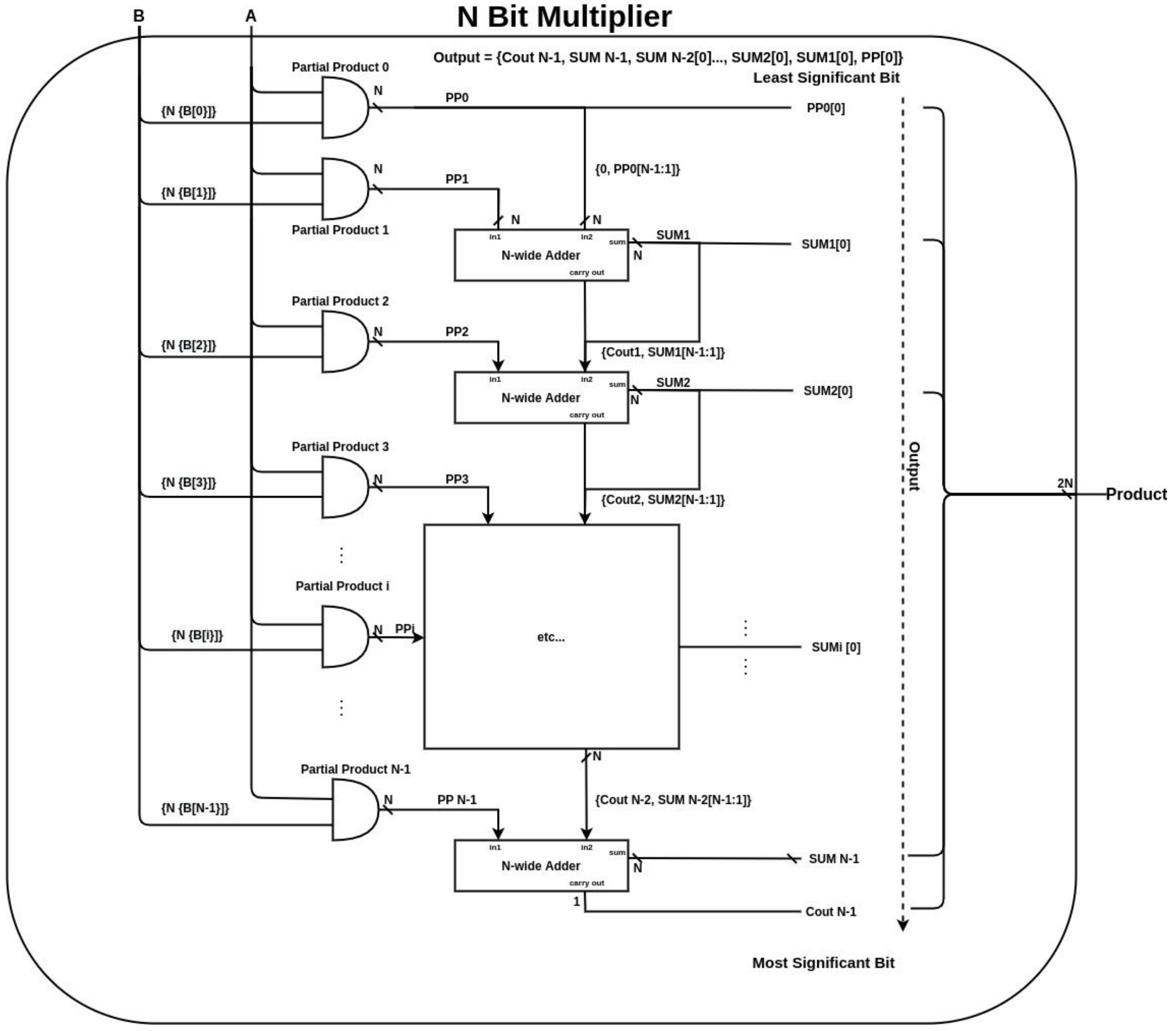
Here is the starter code:

n_bit_multiplier.v
```
module n_bit_multiplier #(parameter N = 4)
(
   ________ _______ [___:___] a, // Input 1
   ________ _______ [___:___] b, // Input 2
   ________ _______ [___:___] p  // Output Product: a*b
);

     // You got this! :)

endmodule
```

Block Diagram for N-Bit Multiplier:



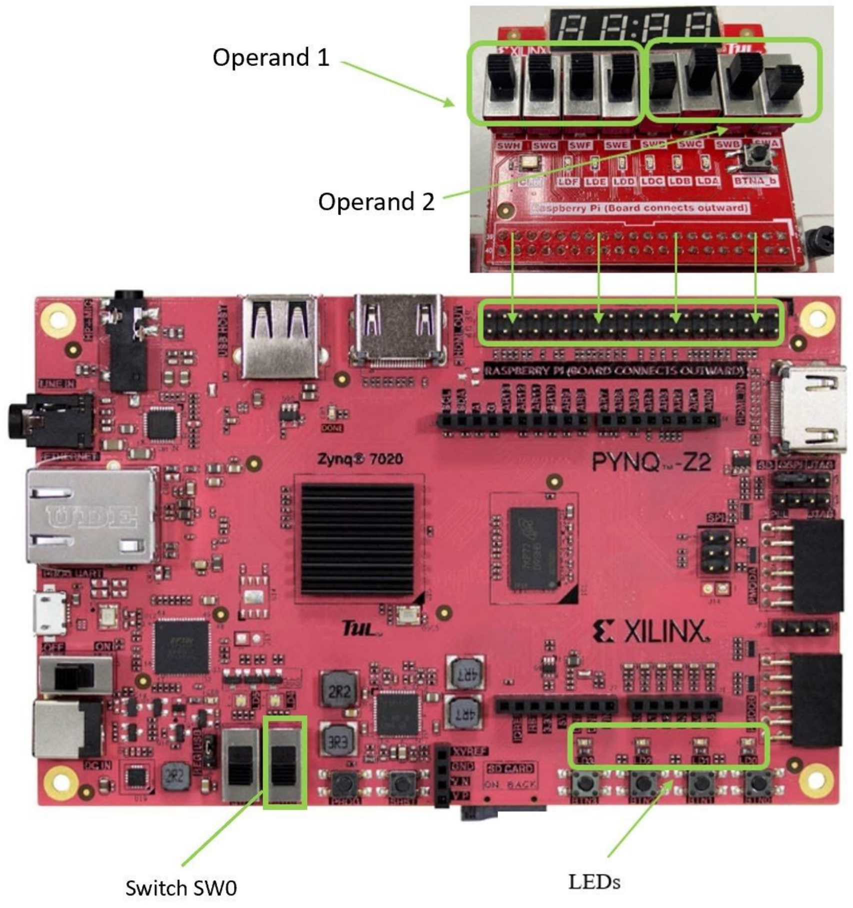
4. Testing on the FPGA

Once you're **positive** your design is working according to your testbenches, you're ready to synthesize. Synthesize your design according to the process outlined in Lab 0. Make sure 'lab1_extension' is selected as the top-level module in the hierarchy before you generate the bit file. Once everything is complete, download your design to the FPGA, as described in lab 0 using the Vivado hardware manager.

Due to hardware constraints we only support 4-bit multiplication operands. Using your switch pad, you can enter the two operands (use the first 4 switches to set the bits for the first operand and the next 4 switches to set the bits for the second operand). The operands can be in any order because of commutativity of the multiplication operation.

You should see the LEDs right above the buttons (btn*) on the FPGA light up to show the 4 LSBs of the product of your operands. You can toggle SW0 to show either the 4 LSBs or the product or the 4 MSBs of the product. When the switch SW0 on the board goes high, it displays the 4MSBs, otherwise (switch is off) the LEDs will display the 4LSBs of your product.

An annotated picture of the FPGA is shown below, highlighting components you'd need:



5. Lab Submission (due on 1/22/20 at 10:30 AM)

Make sure to back up your files before following the submission instructions below. Also remember that this component of lab1 requires individual lab submission. The deliverable is:

Create a **.zip** file with the following contents:

1. All of your Verilog files (*.v files), for both your modules and testbenches. Make sure you include the correct versions of your files! Sometimes Vivado duplicates your files and caches old versions. This includes:
   n_bit_multiplier.v, n_bit_multiplier_tb.v
2. Your bit file, located at lab1_extension/lab1_extension.runs/impl_1/:
   lab1_extension.bit
3. A file containing annotated text output of **any** testbenches you modified! Please refer back to lab 0 on how to annotate simulations.

Finally, do not forget to upload the zip file to gradescope. And also don't worry about checking for invalid inputs, the TAs will only test your design using randomly generated valid inputs.