Lab 3: Multipliers

Introduction

In this lab you will build two multiplier circuits and then simulate them using *ModelSim* software with your own testbench. You will also learn about using arrays in System Verilog.

Part I: Carry Save Multiplier

A simple multiplier performs multiplication in hardware in a manner similar to performing it by hand. This type of multiplier is called an *Array Multiplier*. However, array multipliers are quite slow. Faster multipliers can compute a product of two numbers more quickly. For Part I you will design an unsigned 8×8 *Carry Save Multiplier* (CSM).

NOTE: The suggested method of completing Part I using `generate` loops may be conceptually difficult at first. However, it is a good way to learn how to use loops to create more complex hardware. You are **strongly** encouraged to draw out the entire 8×8 multiplier schematic, label all the wires and use that as your guide for designing the hardware.

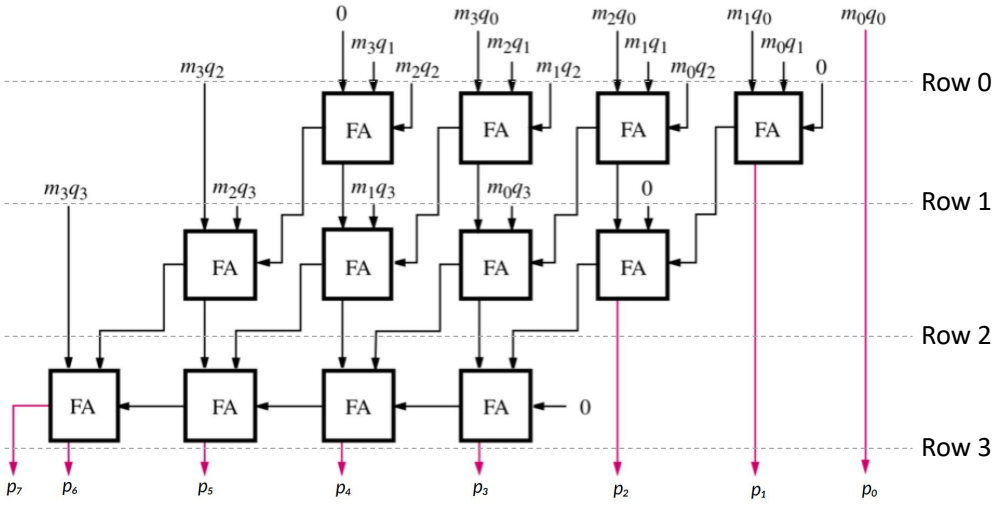


Figure 1: 4-bit Multiplier Carry Save Array [1]

Figure 1 shows a 4×4 CSM, which multiplies a quotient $q0 - q3$ by the multiplicand $m0 - m3$ to produce the 8-bit result, $p0 - p7$. The adder inputs/outputs are called 'partial products (pp)'. Your multiplier must have two 8-bit signals x and y as inputs and the 16-bit signal *out* as output.

It is recommended that you design your multiplier using two nested `generate for` loops. To help with this, start by thinking of the wires in the schematic as being a a 2D 'array' of wires. This arrays has 2N-columns, where N is the width of the multiplier. For example, in Figure 1, this 'array' consists of 8 columns (one

[1] Pg. 354, "Computer Organization and Embedded Systems", 6th ed., Hamacher, Vranesic, Zaky and Manjikian"

column per final product bit p), while your 8x8 multiplier would have 16 columns. For the rows, we consider the inputs to each row of full-adder (FA) as well as the final output p as rows. For Figure 1, this would give us 4 rows (as marked), while for your 8x8 multiplier you would have 8 rows. And in each case, you will have 1 fewer number of rows of full-adders (FA), namely 3 rows of FAs for the 4×4 and 7 rows of FAs for the 8×8 unit.

Some code has been provided for you in the 'part1.sv' starter kit file. The line `logic [15:0] pp [9]` creates a 2D 'array' of wires, with 9 rows and 16 columns. The next line sets `pp[0]` to be all zeroes. While we said earlier that the 8×8 multiplier has 8 rows of wires, we are adding an artificial row of all 0s at the top. This does not correspond to anything in your hardware but is done solely to help with coding the `for` loops, as we shall explain later. So, using this 9×16 array, the partial products input to the first row of adders will be $pp[1][15:0]$.

Note the syntax for declaring arrays in System Verilog is different than C++. One dimension goes before the signal name (to denote the bit width, similar to regular `logic` signals), while the number of wires goes after the signal name. Here we have 9 'pp' wires, each of which is 16-bits in size. Thus, $pp[1]$ refers to the entire 16-bits input to the first row of adders. Accessing each bit, however, is similar to accessing 2D arrays in C++. To access the least significant bit of `pp[0]`, you use `pp[0][0]`. We need another 2D array to hold the carry signals between FA blocks, which is declared with `logic [16:0] cin[9]`.

When writing your code, keep in mind that full adder (FA) cells only exist in some positions of the 2D array. You must write the logic to determine where a FA cell exists and connect the right inputs and outputs to it. Where there is no FA cell, the partial product from above passes directly below. Thus, for $p[0]$ in Figure 1, each row will have a pp associated with it, but will simply take the value of the pp from the row above. For row 0, for example, the final output $p0 = pp[4][0] = pp[3][0] = pp[2][0] = pp[1][0] = pp[0][0]$. Hopefully you can see how this can be designed using loops. This is also where setting `pp[0]` to 0 will come in handy. As the logic for each row uses the partial products from the row above, we can use the same logic for the very first row as well, since the row 'before' that (i.e., $pp[0]$) is set to all 0s. Without this, you would need to treat the first row as a special case which would make your loops more complex. This is done to show you that you can declare 'wires' such $pp[0]$ that do not correspond to any hardware, but are used only to make your code simpler.

Once you have completed coding the loops, you will need a 8-bit ripple carry adder to get the upper bits of the multiplier output (i.e., `out[15:8]`). Bits `out[7:0]` are directly set as the partial products from the for loops, without having to be added first.

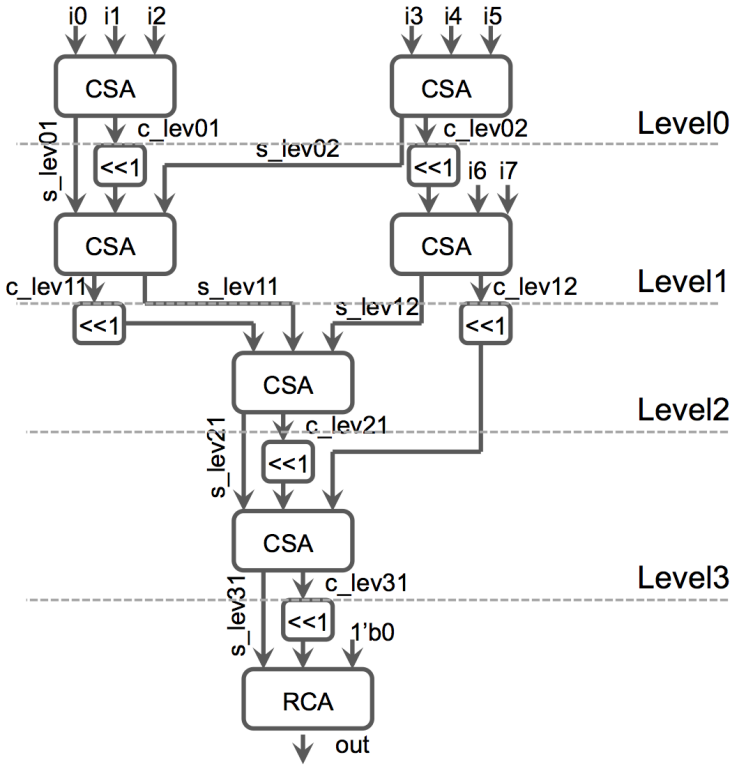
Make sure to write a testbench and test your design as you go. For a design of this complexity, you do not want to code it all up and then find that you need to re-design the whole thing. It may also be useful to start by designing a 4×4 CSM first before trying the 8×8 as the smaller design will be easier to manually debug.

Also, you must not use the * and + operators in the design of your circuit. Your TA will check your code during the in-lab demo and any designs that use either of these operations will receive no marks. You may use the full adder module provided to you but you must write code for all the other necessary hardware.

Part II: Wallace Tree Multiplier

For Part II, you must design an even faster multiplier than the CSM, namely the Wallace Tree Multiplier (WTM). As adding up the partial products is the slowest stage of multiplication, the WTM uses fewer stages to perform addition compared to a using a CSA (such as as the CSM in Part I). Similar to Part I, for Part II you must design an unsigned 8×8 Wallace Tree Multiplier.

Note: The design of the WTM shown here may be different from the one you saw in class. This is to show you to there are lots of ways to optimize such designs which are all correct but offer different trade-offs between area and latency. For your demo, you must build the CSM shown here.



$x = 1010\_1100$

$y = 1100\_0101$

$i0 = 0000\_0000\_1010\_1100$

$i1 = 0000\_0000\_0000\_0000$

$i2 = 0000\_0010\_1011\_0000$

$i3 = 0000\_0000\_0000\_0000$

$i4 = 0000\_0000\_0000\_0000$

$i5 = 0000\_0000\_0000\_0000$

$i6 = 0010\_1011\_0000\_0000$

$i7 = 0101\_0110\_0000\_0000$

Figure 2: 8-bit Wallace Tree Multiplier Architecture

Figure 2 shows the block diagram of a 8×8 WTM, which uses a pipeline of carry save adders (CSA) followed by a ripple carry adder (RCA) as the final stage. The WTM takes the rows of the partial products as inputs (shows as $i0 - i7$) in the Figure. The values on the left of Figure 2 show $i0 - i7$ for sample values of x and y. You will see that each row of i corresponds to 1 bit of y. That is, $i_n = 0$ for $y_n = 0$, where n is a bit of y. For values of i_n that are not 0, they are the values of x left-shifted by n.

You must first calculate all the values of i in your code. You will also need to create a carry-save adder for your design. You can re-use the full-adder cells and the ripple-carry adder you designed for Part I here. As there are not many CSA units in this design, you are free to do this design 'manually' without using `for` loops.

The provided 'part2.sv' kit file declares a multi-dimensional 'array' for the 's_lev' and 'c_lev' signals shown in the Figure. There are a few things to note about how these signals are declared. The indices for both signals are `[0:3]` and `[1:2]`. This is to show you that indices can go from any number to any number. However, keep in mind that now `s_lev[0][1]` is the most-significant bit and `s_lev[3][2]` the least-significant bit. Declaring the signal this way allows you to write your code to match the Figure. So s_lev11 in the Figure can be `s_lev[1][1]` in your code. If instead the signals were declared as `s_lev[3:0][1:0]`, such a mapping would not be possible. Also, you may notice that the declaration creates $4 \times 2 = 8$ array elements but we only use 6 in the Figure. This is because of the irregular nature of the WTM. But this does not add any hardware overhead; any unused wires will simply be optimized away by the compiler.

If you find the WTM conceptually difficult, you are encouraged to learn more about the WTM from online resources as well. Some resources will be provided for you on Quercus to help with this. However, if you find other resources yourself, make sure the design is similar to the one shown in Figure 2. Some online resources shown WTM designs that use 7 or 8 stages to calculate an 8×8 product, which is incorrect.

In-Lab Demo

For this lab, demo your code and testbenches, showing that your multiplier design works for all possible inputs.

1. **Part I**: (5 marks) Demo your Carry Save Multiplier.
2. **Part II**: (5 marks) Demo your Wallace Tree Multiplier.