LSU EE 4755 Homework 2 Due: 8 October 2019

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`*. For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2019/hw02.v.html`.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account, copy the assignment, and run the Verilog simulator and synthesis program on the unmodified homework file, `hw02.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Correction (December 2019)

When assigned in October 2019 this assignment defined clz backward, starting at the least-significant bit. That has been corrected in this version and in the posted code.

Homework Overview

A *count leading zeros (clz)* operation returns the number of consecutive zeros starting at the most significant bit of an integer's binary representation. For example, the clz of 00101_2 is 2, the clz of 101_2 is 0, and the clz of 32-bit number 0_2 is 32. The Verilog module below computes the clz of its input:

```
module clz
  #( int w = 19, int ww = $clog2(w+1) )
   ( output var logic [ww-1:0] nlz, input uwire logic [w-1:0] a );

   uwire [w:0] aa = { a, 1'b1 };
   always_comb for ( int i=0; i<=w; i++ ) if ( aa[i] ) nlz = w-i;
endmodule
```

The module was written as behavioral code, but it does turn out to be synthesizable. Nevertheless, one may wonder if the synthesis program will do a good job with this. (Later in the semester we will learn what kind of hardware will be inferred for the description above.) One way to find out is to design a module which *should* be efficient and see how well it compares to what the synthesis program does with the module above. That, and the use of generate statements, is the subject of this assignment.

Testbench Code

The testbench for this assignment, which can be run when visiting the file in Emacs in a properly set-up account by pressing $\boxed{\text{F9}}$, tests the `clz_tree` module at several different widths. All should initially fail. A shortened sample of the testbench output appears below:

```
ncsim> run
** Starting tests for width 1.
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Error for width  1: input 1:  z != 0 (correct).
Error for width  1: input 0:  z != 1 (correct).
Width 1, done with 10 tests, 10 errors.
** Starting tests for width 2.
Error for width  2: input 3:  z != 0 (correct).
Width 2, done with 20 tests, 20 errors.
** Starting tests for width 5.
```

```
[snip]
Error for width 17: input 08959:   z != 0 (correct).
Width 17, done with 170 tests, 170 errors.
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
Total number of errors: 610
```

The testbench prints the details of the first four errors it finds, and after that prints just one detail time per width. A total for each width and a grand total are printed, see the transcript above.

Use Simvision to debug your modules. Feel free to modify the testbench so that it presents inputs that facilitate debugging.

Synthesis

The synthesis script, `syn.tcl`, will synthesize `clz` (for reference) and `clz_tree` (your solution). Each module will be synthesized at three widths, and with two delay targets, an easy 10 ns and a un-achievable 0.1 ns. If a module doesn't synthesize −.001 s is shown for its delay. The script is run using the shell command `genus -files syn.tcl`, which invokes Cadence Genus. If you would like to synthesize additional modules or sizes edit `syn.tcl` near the bottom.

The synthesis script shows area (cost), delay, and the delay target in a neat table. Additional output of the synthesis program is written to file `spew.log`.

Problem 1: Complete module `clz_tree` so that it computes the clz of its input in a tree-like fashion. For the non-terminal case it should instantiate two `clz_tree` modules and each should operate on part of the input, `a`. The outputs of these two modules should be appropriately combined. To help you get started, a recursive solution to Homework 1, `mult_tree`, is in `hw02.v`.

An easy mistake to make is using the wrong sized variable in a module port connection. Previously the Verilog software (`ncelab` to be precise) would issue a warning which was easy to miss. Now a port size mismatch is a fatal error.

For maximum credit do not use adders in your design. Adders can be avoided if the size of the low module is always a power of 2.

See the Verilog code check boxes for additional items to check for.

Problem 2: Run the synthesis program and indicate how your module compares to the behavioral module, `clz`. Indicate which results are expected, and which are not expected, and explain why.