Problem 0: Following instructions at [https://www.ece.lsu.edu/koppel/v/proc.html](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 Overview

A **count leading zeros (clz)** operation returns the number of consecutive zeros starting at the least significant bit of an integer’s binary representation. For example, the clz of 10100 \(_2\) 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:

```verilog
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 = { 1'b1, a };
  always_comb for ( int i=w; i>=0; i-- ) if ( aa[i] ) nlz = 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 `F9`, tests the `clz_tree` module at several different widths. All should initially fail. A shortened sample of the testbench output appears below:

```
nccsim> 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.
nccsim: *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 on next page.
**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.

The solution appears below. The partial-credit solution, using an adder, appears first.

```verilog
/// SOLUTION – With Adder. Two points would be deducted.
module clz_tree_fat
    #( int w = 19, int ww = $clog2(w+1) )
    ( output uwire [ww:1] nlz, input uwire [w:1] a );

    if ( w == 1 ) begin
        assign nlz = ~ a;
    end else begin

        localparam int wlo = w/2;
        localparam int whi = w - wlo;
        localparam int wlo = $clog2(wlo+1);
        localparam int whi = $clog2(whi+1);

        uwire [wwlo:1] lz_lo;
        uwire [wwhi:1] lz_hi;

        clz_tree_fat #(wlo) clo( lz_lo, a[wlo:1] );
        clz_tree_fat #(whi) chi( lz_hi, a[w:wlo+1] );

        assign nlz = lz_lo < wlo ? lz_lo : wlo + lz_hi;
    end
endmodule
```

The better solution, without the adder, is on the next page.
The solution below avoids an adder by setting the size of the lo module to a power of 2. If all of the low bits are zero, then the clz is the count of the number of high bits, plus a power of 2. The power of 2 to add is parameter $llo$ (see the code).

// SOLUTION – Without Adder

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

if ( w == 1 ) begin
    assign nlz = !a[0];
end else if ( w == 2 ) begin
    assign nlz = { !a[0] && !a[1], !a[0] && a[1] };
end else begin
    // Set wlo to the largest power of 2 strictly less than w.
    //
    localparam int llo = $clog2(w) - 1;
    localparam int wlo = 1 << llo;
    localparam int wwlo = llo + 1;

    // Then set whi to the number of remaining bits.
    //
    localparam int whi = w - wlo;
    localparam int wwhi = $clog2(whi+1);

    uwire [wwlo-1:0] nlz_lo;
    uwire [wwhi-1:0] nlz_hi;

    // Declare recursive modules.
    //
    clz_tree #(wlo) clo( nlz_lo, a[wlo-1:0] );
    clz_tree #(whi) chi( nlz_hi, a[w-1:wlo] );

    // Split the nlz_lo and nlz_hi outputs into "overflow" bits,
    // ov_lo and ov_hi, and the remaining bits lh_lo and lh_hi.
    // There is no overflow bit if the size is not a power of two.
    // Because wlo is always a power of two it will always have an
    // overflow bit.
    //
    uwire ov_lo, ov_hi;
    uwire [llo-1:0] lh_lo, lh_hi;

    // The lo bits are easy because there’s always an overflow.
    assign {ov_lo,lh_lo} = nlz_lo;

    if ( wlo == whi ) assign {ov_hi,lh_hi} = nlz_hi;
else
    assign lz_hi = nlz_hi;

// If the low bits are all zeros, use the count from the high
// bits and then further below effectively add wlo to nlz.
//
assign nlz[llo-1:0] = ov_lo ? lz_hi : lz_lo;

// Next, add on the overflow bits without using an adder.
//
// The first case, wlo==whi, is for when both wlo and whi (and
// w) are all powers of two. In that case we must compute two
// more bits of nlz. If ov_lo is zero they are both zero because
// the number of leading zeros will be strictly less than wlo.
// If ov_lo is one then the high two bits will be 1 + ov_hi.
//
// In the second case, (the else which means whi<wlo), there is
// no ov_hi bit. So the most significant bit is 0 if ov_lo is
// zero and 1 if ov_lo is one.

if ( wlo == whi )
    assign nlz[llo+1:llo] = ov_lo ? { ov_hi, !ov_hi } : 2'b0;
else
    assign nlz[llo] = ov_lo;

end

endmodule
**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.

Attention students studying for exams: A good practice problem would be to show the synthesized hardware for these modules.

The behavioral model looks at bits sequentially, starting at the most-significant bit. The hardware as initially inferred would have a chain of multiplexors either selecting $i$ if $aa[i]$ were 1, or the prior value of $nlz$ otherwise. The $nlz$ output would pass through $w$ multiplexors, for a delay of $tw$, after optimizing for the fact that $i$ is constant.

In contrast the critical path through the tree modules passes through $\lceil \lg w \rceil$ units, and so that should be faster. In the 0.1 ns delay target results, shown below, the behavioral model is fastest at $w = 30$ bits and the adder-less *clz_tree* module is only fastest at $w = 32$ bits. At best, *clz_tree* never does poorly when delay is a priority. The behavioral module however is consistently more costly than *clz_tree*.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Area</th>
<th>Delay</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Actual</td>
<td>Target</td>
<td></td>
</tr>
<tr>
<td>clz_w30</td>
<td>18540</td>
<td>1.653</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_w30</td>
<td>17977</td>
<td>1.653</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w30</td>
<td>17977</td>
<td>1.653</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_w32</td>
<td>26290</td>
<td>3.110</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_w32</td>
<td>21706</td>
<td>1.425</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w32</td>
<td>21401</td>
<td>1.296</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_w35</td>
<td>23140</td>
<td>1.300</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_w35</td>
<td>22578</td>
<td>1.300</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w35</td>
<td>26073</td>
<td>2.094</td>
<td>10.000 ns</td>
</tr>
<tr>
<td>clz_w30_1</td>
<td>30053</td>
<td>0.504</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_w30_4</td>
<td>38532</td>
<td>0.650</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w30_1</td>
<td>37798</td>
<td>0.861</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_w32_1</td>
<td>36476</td>
<td>1.007</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_w32_5</td>
<td>37356</td>
<td>0.577</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w32_2</td>
<td>32254</td>
<td>0.634</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_w35_1</td>
<td>37008</td>
<td>0.606</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_w35_6</td>
<td>37008</td>
<td>0.606</td>
<td>0.100 ns</td>
</tr>
<tr>
<td>clz_tree_fat_w35_1</td>
<td>37008</td>
<td>0.606</td>
<td>0.100 ns</td>
</tr>
</tbody>
</table>