LSU EE 4755 Homework 2 Solution Due: 16 September 2015

The Verilog part of the solution to this assignment can be found in `/home/faculty/koppel/pub/ee4755/hw/2015f/hw02/hw02/hw02-sol.v` and a syntax-highlighted version can be found at `http://www.ece.lsu.edu/koppel/v/2015/hw02-sol.v.html`.

Problem 0: Follow the instructions for account setup and homework workflow on the course procedures page, `http://www.ece.lsu.edu/koppel/v/proc.html`. Run the testbench on the unmodified file. There should be errors on all but the `min_4` (Four-element) module. Try modifying `min_4` so that it simulates but produces the wrong answer. Re-run the simulator and verify that it's broken. Then fix it.

Note: There are no points for this problem.

Problem 1: Module `min_n` has an `elt_bits`-bit output `elt_min` and an `elt_count`-element array of `elt_bits`-bit elements, `elts`. Complete `min_n` so that `elt_min` is set to the minimum of the elements in `elts`, interpreting the elements as unsigned integers. Do so using a linear connection of `min_2` modules instantiated with a `genvar` loop. (A linear connection means that the output of instance i is connected to the input of instance $i+1$.)

Verify correct functioning using the testbench.

Solution appears below.

```
module min_n
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts [ elt_count ] );

   /// SOLUTION

   // Declare wires to interconnect the instances of min_2 instantiated
   // in the genvar loop.
   //
   uwire [elt_bits-1:0] im[elt_count:0];  // im: Inter-Module
   assign               im[0] = elts[0];

   // Instantiate elt_count-1 min_2 modules. The inputs of the first
   // module (i=1) connect to elt[0] and elt[1]. Subsequent modules
   // connect to an elt and the module instantiated in the previous
   // iteration.
   //
   for ( genvar i = 1; i < elt_count; i++ )
     min_2 #(elt_bits) m( im[i], elts[i], im[i-1] );

   // Connect the output of the last instance to the module output.
   //
   assign elt_min = im[elt_count-1];

endmodule
```

Problem 2: Module `min_t` is to have the same functionality as `min_n`. Complete `min_t` so that it recursively instantiates itself down to some minimum size. The actual comparison should be done by a `min_2` module.

Verify correct functioning using the testbench.

Solution appears below. In this solution recursion ends when `elt_count` is 1, in which case the module output, `elts_min` is connected directly to the module input, `elts[0]`. Otherwise two smaller `min_t` modules are instantiated.

```
module min_t
  #( int elt_bits = 4,
     int elt_count = 8 )
   ( output uwire [elt_bits-1:0] elt_min,
     input uwire [elt_bits-1:0] elts [ elt_count-1:0 ] );

   /// SOLUTION

   if ( elt_count == 1 ) begin

      // Recursion ends here with one elt. Of course, it is the
      // minimum. (And the maximum, and the average, and the median.)
      //
      assign elt_min = elts[0];

   end else begin

      // If there are at least two elements instantiate two smaller
      // modules.

      // Compute the number of elements to be handled by each
      // module. (Note that elt_count can be odd, which is why we need
      // a separate elt_hi and elt_lo.)
      //
      localparam int elt_hi = elt_count / 2;
      localparam int elt_lo = elt_count - elt_hi;

      // Wires for interconnection of modules.
      uwire [elt_bits-1:0] minl, minh;

      // Recursively declare two modules.
      //
      min_t #(elt_bits,elt_hi) mhi(minl,elts[elt_count-1:elt_lo]);
      min_t #(elt_bits,elt_lo) mlo(minh,elts[elt_lo-1:0]);

      // Combine the output of the two modules above.
      //
      min_2 #(elt_bits) m2(elt_min,minl,minh);

   end

endmodule
```

Problem 3: By default the synthesis script will synthesize each module for two array sizes, four

elements and eight elements.

(*a*) Run the synthesis script unmodified. Use the command `rc -files syn.tcl`. Explain the differences in performance between the different modules.

The output of the synthesis script appears below.

We should expect the cost and performance of `min_n` and `min_b` to be about the since they should synthesize to the same hardware. That can be seen by comparing the `if` statement in `min_b` to the `assign` in `min_2`: both will synthesize to a multiplexor. The behavioral `for` loop in `min_b` and the generate loop in `min_n` should interconnect those multiplexors in the same way. From the table below we see that the synthesis program output is consistent with our expectations.

We should expect the cost of `min_n` and `min_t` to be about the same since they have the same number of comparison units, they are just connected in a different order. But we should expect `min_t` to be faster since the critical path is through $\log_2 n$ `min_2` modules. The delay numbers match our expectations for the eight-element version, but at four elements the linear versions are faster. One reason for this might be that for some reason, the synthesis program is using a higher-cost comparison unit in the linear versions, adding to their cost and improving their performance. In the four-element versions that added performance puts them ahead of the tree version. But for the eight-input versions the tree version is clearly faster.

Possible Test Question: Estimate the critical path in the tree and linear versions of the `min` units.

The second table below shows the synthesis of the modules at a much higher delay target so that the synthesis program will be optimizing primarily for area. In this case the both the cost and performance differences between the tree and linear versions meet our expectations.

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits4_elt_count4	8592	1416	100
min_b_elt_bits4_elt_count4	14360	1367	100
min_n_elt_bits4_elt_count4	14360	1367	100
min_t_elt_bits4_elt_count8	25536	1935	100
min_b_elt_bits4_elt_count8	29460	3712	100
min_n_elt_bits4_elt_count8	29460	3712	100

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits4_elt_count4	5180	2413	50000
min_b_elt_bits4_elt_count4	5152	3280	50000
min_n_elt_bits4_elt_count4	5152	3280	50000
min_t_elt_bits4_elt_count8	11784	3609	50000
min_b_elt_bits4_elt_count8	12176	7796	50000
min_n_elt_bits4_elt_count8	12176	7796	50000

(*b*) Modify and re-run the synthesis script so that it synthesizes the modules with `elt_bits` set to 1.

The synthesis program should do a better job on the behavioral and linear models *in comparison to the tree model*. Why do you think that is? *Hint: The 1-bit minimum module is equivalent to another common logic component that the synthesis program can handle well. Note: the phrase about the tree model was not in the original assignment.*

In the table below we see that with a 1-bit element size all three modules have identical cost and performance.

With a one-bit element size the circuit acts as an AND gate, and this is something the synthesis program can figure out. Since the synthesis program sees that `min_n` and `min_b` are performing AND operations it can apply the same kind of tree reduction technique that we incorporated by hand in `min_t`, and so all modules are the same.

Note that the key insight here is that in the general case the synthesis program could not figure out that the minimum operation is associative, and so it could not apply a tree reduction. But with the element size set to 1, it converted minimum to AND, which it did recognize as associative.

Module Name	Area	Delay Actual	Delay Target
min_t_elt_bits1_elt_count4	288	155	100
min_b_elt_bits1_elt_count4	288	155	100
min_n_elt_bits1_elt_count4	288	155	100
min_t_elt_bits1_elt_count8	912	292	100
min_b_elt_bits1_elt_count8	912	292	100
min_n_elt_bits1_elt_count8	912	292	100