LSU EE 4755 Homework 4 Solution Due: 12 October 2016

Problem 0: First, follow the instructions for account setup and homework workflow on the course procedures page, `https://www.ece.lsu.edu/koppel/v/proc.html`.

Look through the code in `hw04.v`. Module `lookup_behav` in file `hw04.v` has a w-bit input `char` and an n-element array of w-bit quantities named `chars`. (Parameter `nelts` is n and parameter `charsz` is w.) The module also has a 1-bit output `found` which is logic `1` iff any element of `chars` is equal to `char`. Finally, the module has a $\lceil \lg n \rceil$-bit output `index` which is set to the element number of `chars` that matches `char`, or `0` if `found` is `0`. Assume that no two elements of `chars` are identical.

For example, suppose input `char` is set to `102` and that `chars` is `{63,124,102,92}`. Then output `found` will be `1` and `index` will be `2`. If `char` were `7` `index` would be `0` and `found` would be `0`, if `char` were `63` `index` would be `0` and `found` would be `1`, etc. The alert student will have recognized that $n = 4$ and that $w \geq 7$ in these examples.

Module lookup is coded in synthesizable behavioral form that describes combinational logic. The `hw04.v` file contains two other modules which are to do the same thing, `lookup_linear` and `lookup_tree`, but those modules are not yet finished.

The testbench tests all of these modules. It tests them for sizes (n) of 4, 5, 10, 15, 16, 30, 40, and 64. To change which sizes are tested (or the order in which they are tested) edit the `testbench` module.

To have the testbench test only some of these modules (say, skip the `lookup_tree` tests until after `lookup_linear` is working) look for the `for` loop with `mut=0` and modify it appropriately. (It should be easy to figure out the numbers.)

A synthesis script is provided that will synthesize all three modules at different sizes and both with and very lax timing constraint and a very strict timing constraint. The script can be run using the command `rc -files syn.tcl`. Initially it will stop with an error. To see it run to completion before starting the assignment have it only synthesize `lookup_behav` (see below). Pre-set synthesis options (in file `.synth_init`) were chosen to reject any design that is not combinational.

If there is an error when using the synthesis script then follow the manual synthesis steps on the procedures page and look for error messages.

To change which modules are synthesized edit the `set modules` line (near the bottom) in file `syn.tcl`. The values for `nelts` and other items can also be changed by editing the file.

Note: There are no points for this problem.

Problem 1: Complete `lookup_linear` so that it does the same thing as `lookup_behavioral` but by using as many copies of `lookup_elt` as it needs. That is, `lookup_linear` should use generate statements to instantiate `lookup_elt` and it should include whatever other code is needed to use these instances to compute the correct outputs.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

(The complete solution Verilog code is in the assignment directory and at `https://www.ece.lsu.edu/koppel/v/2016/hw04-sol.v.html`.) There are two approaches to solving this problem. In the easy approach, which is sufficient to get full credit, generate statements are used to instantiate the `lookup_elt` modules but behavioral code is used to compute `index`.

In the alternative solution (`lookup_linear_alt`), generate statements are used both to instantiate the modules and compute index. To compute index an array of wires, `[idx_sz-1:0]idx_i[nelts-1:-1]` is declared. Element `idx_i[i]` is the value of `index` taking into account elements 0 to `i`.

```
module lookup_linear
  #( int charsz = 8,
     int nelts = 15,  // Pronounced en-elts.
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION – Easy
   //
   //  Instantiate nelts modules, but use use behavioral code to examine
   //  their found (match) outputs.

   // Declare wires to connect to the found outputs of the instantiated modules.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);

   always_comb begin
      found = 0;
      index = 0;
      for ( int i=0; i<nelts; i++ )
        if ( match[i] ) begin index = i; found = 1; end
   end

endmodule
```

```
module lookup_linear_alt
  #( int charsz = 8,
     int nelts = 15,  // Pronounced en-elts.
     int idx_sz = $clog2(nelts) )
   ( output logic found,
     output logic [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts-1:0] );

   /// SOLUTION – Alternative
   //
   //  Use generate statements to instantiate the modules and to
   //  generate logic to find the index.


   // Instantiate nelts lookup_elt modules and compute found.
   //
   uwire    [nelts-1:0] match;

   for ( genvar i=0; i<nelts; i++ )
     lookup_elt #(charsz) le(match[i],char,chars[i]);

   assign found = | match;


   // Instantiate logic to find the index of the last matching character.
   //

   uwire [idx_sz-1:0]   idx_i[nelts-1:-1];

   assign idx_i[-1] = 0;

   for ( genvar i=0; i<nelts; i++ )
     // If no match pass along previous idx_i, otherwise replace it with i.
     assign idx_i[i] = match[i] ? i : idx_i[i-1];

   assign   index = idx_i[nelts-1];

endmodule
```

Problem 2: Complete module `lookup_tree` so that it performs the lookup using recursive instantiations of itself. Take care so that `index` is computed efficiently. *Hint: think about how to compute index efficiently when n (`nelts`) is a power of 2, then get the same efficiency for any n.*

If completed correctly, the cost and especially the performance at larger sizes should be better than `lookup_behavioral` and (unless you did an unexpectedly good job) better than `lookup_linear`.

- Behavioral or structural code can be used.
- The module must be synthesizable.
- Assume that all elements of `chars` are different.

(The complete solution Verilog code is in the assignment directory and at `https://www.ece.lsu.edu/koppel/v/2016/hw04-sol.v.html`.) First, we need to use generate statements to split elaboration into two cases: $n = 1$, and $n > 1$. For $n = 1$ `index` will always be zero (there's only one element in the array and its index is zero), and `found` can directly be assigned the expression `char == chars[0]`.

Two solutions will be described. In `lookup_tree_simple` work is split evenly between the two instantiated modules but this results in a more costly computation of `index` than is necessary. In `lookup_tree`, the size (value of `nelts`) of one instantiated module is forced to be a power of 2, reducing cost.

For $n > 1$ we need to split the input array, `chars`, between two instantiated `lookup_tree` modules and combine their `found` and `index` outputs. In `lookup_tree_simple` the array is split in half, the approach used in the `pop_n` module presented in class. Objects `lo_sz` and `hi_sz` are the sizes of the instantiated modules, note that these are used to compute the number of bits in the index outputs.

Logic is also needed to take the found and index outputs of the two instantiated modules, named `lo_f`, `hi_f`, `lo_idx`, and `hi_idx`, and compute the `found` and `index` outputs of the module. A mistake that many students make when trying to solve this problem is to try to take into account what is happening in all instantiated modules at every level when designing this logic. Instead, just assume that `lo_f`, `hi_f`, `lo_idx`, and `hi_idx` are correct, and use them to compute `found` and `index`. If such logic can be found, then the module will work at any size.

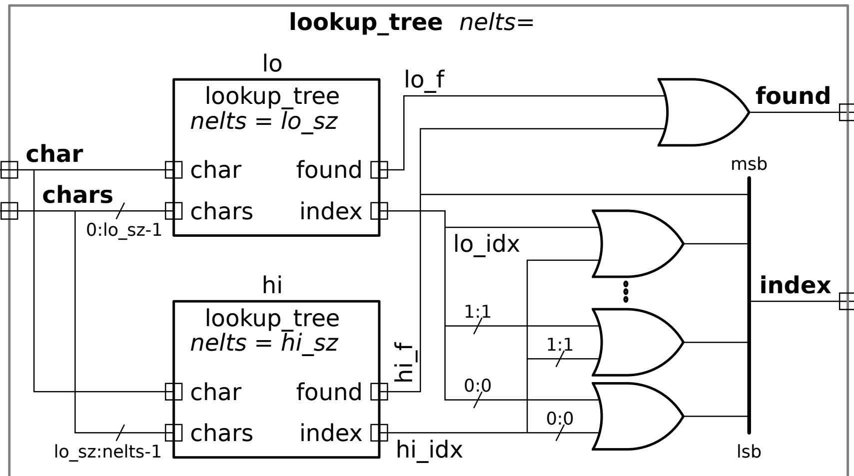
The output `found` is simply the OR of `lo_f` and `hi_f`. If `lo_f` is 1, then `index` is `lo_idx`, but if `hi_f` is 1 then `index` is `lo_sz + hi_idx`. We don't need to worry about both `lo_f` and `hi_f` being one (the problem statement said it couldn't happen). If `hi_f` and `lo_f` are both 0 then `lo_idx` and `hi_idx` will both be 0 and `index` should be set to zero. Therefore, `index` can be set to `hi_f ? lo_sz + hi_idx : lo_idx`. That's it for the simple solution.

The problem though was to find a solution that computed `index` efficiently. Consider the sum `lo_sz + hi_idx`. If `lo_sz` were chosen to be a power of 2, and `lo_sz >= hi_sz` then instead of adding we would just be putting a 1 in bit position `lo_bits`: `{1'b1,hi_idx}`. We can re-write this as `{hi_f,hi_idx}` since this is the case where `hi_f` is 1. And since `hi_f` is 1 we know `lo_idx` is all zeros, so we can use the expression `{ hi_f, lo_idx | hi_idx }`. As the alert student may have realized, that expression also is correct for the case where `lo_f` is 1 and the case where both are 0. The OR gates are much less expensive than an adder and a multiplexor, even an adder with a constant input.

The code for the two modules appears below, along with the inferred hardware for the second module (that computes index efficiently.)

```
module lookup_tree_simple
  #( int charsz = 8,
     int nelts = 15,
     int idx_sz = $clog2(nelts) )
   ( output uwire found,
     output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,
     input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION – Unoptimized

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;

   end else begin

      // Split the character array between recursive instantiations.
      //
```

```
    localparam int lo_sz = nelts / 2;
    localparam int lo_bits = $clog2(lo_sz);
    localparam int hi_sz = nelts - lo_sz;
    localparam int hi_bits = $clog2(hi_sz);
    //
    // Note that we need to compute lo_bits and hi_bits correctly so
    // that we can declare index connections, lo_idx and hi_idx, of
    // the correct size.

    uwire       lo_f, hi_f;
    uwire [lo_bits-1:0] lo_idx;
    uwire [hi_bits-1:0] hi_idx;

    lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
    lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

    assign     found = lo_f || hi_f;

    assign     index = hi_f ? lo_sz + hi_idx : lo_idx;

  end

endmodule
```




```
module lookup_tree
  #( int charsz = 8,  int nelts = 15,  int idx_sz = $clog2(nelts) )
   ( output uwire found,  output uwire [idx_sz-1:0] index,
     input uwire [charsz-1:0] char,  input uwire [charsz-1:0] chars[nelts] );

   /// SOLUTION -- Preferred

   if ( nelts == 1 ) begin

      assign found = char == chars[0];
      assign index = 0;  // Actually, we are assigning a zero-bit vector.

   end else begin

      // Make the size of the first lookup_tree (lo) a power of two.
      localparam int lo_bits = idx_sz - 1;
      localparam int lo_sz = 1 << lo_bits;

      // Compute the size of the second lookup_tree (hi).
      localparam int hi_sz = nelts - lo_sz;
      localparam int hi_bits = $clog2(hi_sz);

      uwire       lo_f, hi_f;
      uwire [lo_bits-1:0] lo_idx;
      uwire [hi_bits-1:0] hi_idx;

      lookup_tree #(charsz,lo_sz) lo( lo_f, lo_idx, char, chars[ 0:lo_sz-1 ] );
      lookup_tree #(charsz,hi_sz) hi( hi_f, hi_idx, char, chars[lo_sz:nelts-1]);

      assign     found = lo_f || hi_f;

      if ( lo_bits == 0 ) assign index = hi_f;
      else                assign index = { hi_f, hi_idx | lo_idx };
   end

endmodule
```

Problem 3: Run the synthesis script and characterize the strengths and weaknesses of each module. (For example, module X has lowest cost for low-speed designs.)

In a follow-on homework assignment additional questions will be asked about these modules.

The cost of the tree solution is almost always lower than the other designs, the performance is usually but not always better. For the low-cost (large delay) configurations behavioral design is usually most expensive, but is less expensive than the linear designs for the high-performance designs.

```
Note: linear_tree below is linear_tree_simple above,
  and linear_tree_opt below is linear_tree above.
Module Name                             Area   Delay   Delay
                                               Actual  Target
lookup_behav_charsz8_nelts4             9152     927   10000
lookup_linear_charsz8_nelts4            9012     990   10000
lookup_tree_charsz8_nelts4              8916    1026   10000
lookup_tree_opt_charsz8_nelts4          8988     952   10000
lookup_behav_charsz8_nelts15           35444    2348   10000
lookup_linear_charsz8_nelts15          34996    2338   10000
lookup_tree_charsz8_nelts15            34280    2606   10000
lookup_tree_opt_charsz8_nelts15        33532    2238   10000
lookup_behav_charsz8_nelts32           74648    3691   10000
lookup_linear_charsz8_nelts32          74212    3257   10000
lookup_tree_charsz8_nelts32            70932    2480   10000
lookup_tree_opt_charsz8_nelts32        71084    2443   10000
lookup_behav_charsz8_nelts40           94028    3862   10000
lookup_linear_charsz8_nelts40          94288    2585   10000
lookup_tree_charsz8_nelts40            95996    3501   10000
lookup_tree_opt_charsz8_nelts40        89292    2778   10000
lookup_behav_charsz8_nelts60          143268    5913   10000
lookup_linear_charsz8_nelts60         141792    5638   10000
lookup_tree_charsz8_nelts60           142828    3963   10000
lookup_tree_opt_charsz8_nelts60       138288    3501   10000
lookup_behav_charsz8_nelts4            12304     621     100
lookup_linear_charsz8_nelts4           13344     594     100
lookup_tree_charsz8_nelts4             13280     598     100
lookup_tree_opt_charsz8_nelts4         10888     640     100
lookup_behav_charsz8_nelts15           46896    1136     100
lookup_linear_charsz8_nelts15          47528    1120     100
lookup_tree_charsz8_nelts15            45268    1151     100
lookup_tree_opt_charsz8_nelts15        41696    1003     100
lookup_behav_charsz8_nelts32          105032    1247     100
lookup_linear_charsz8_nelts32         108688    1288     100
lookup_tree_charsz8_nelts32            96980    1093     100
lookup_tree_opt_charsz8_nelts32        96408    1056     100
lookup_behav_charsz8_nelts40          120132    1523     100
lookup_linear_charsz8_nelts40         131344    1114     100
lookup_tree_charsz8_nelts40           134444    1260     100
lookup_tree_opt_charsz8_nelts40       116320    1144     100
lookup_behav_charsz8_nelts60          184892    1726     100
lookup_linear_charsz8_nelts60         210512    1461     100
```

```
lookup_tree_charsz8_nelts60           185628     1890      100
lookup_tree_opt_charsz8_nelts60       176544     1500      100
```