LSU EE 4755 **Homework 4** Solution **Due: 8 Nov 2024**

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for Verilog programming and synthesis examples, and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) It is the students' responsibility to resolve frustrations and roadblocks quickly. (If you get stuck *just ask for help!*)

This assignment cannot be solved by blindly pasting together parts of past assignments. Solving the assignment is a multi-step learning process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of Verilog syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out digital design resources for help on Verilog, digital design, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample Verilog code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Problems start on next page.

Problem 1: Appearing below is the base case from module `pmatch_mark` in the solution to Homework 3 Problem 2.

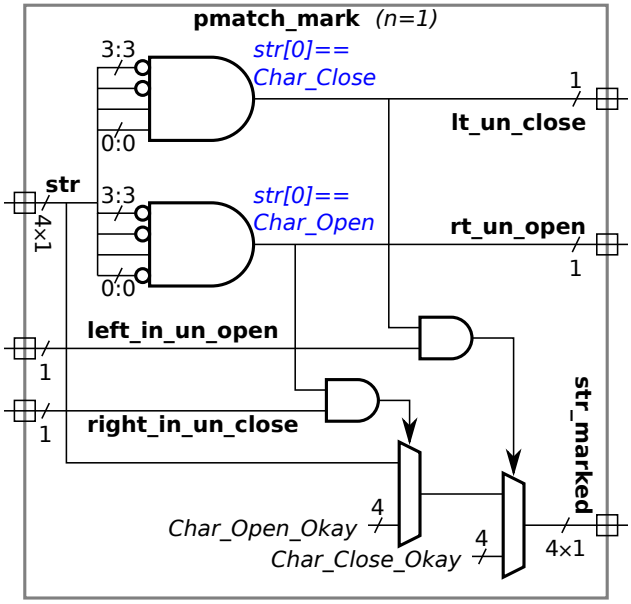
```
typedef enum logic [3:0]
  { Char_Blank = 0,     Char_Dot = 1,
    Char_Open = 2,      Char_Close = 3,
    Char_Open_Okay = 4, Char_Close_Okay = 5 } Char;

module pmatch_mark
  #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin

      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;

      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  :
             str[0];
```

☑ Show the hardware that will be inferred for the base case (`n==1`) shown above.

☑ Show the hardware after optimization and ☑ for the default value of `wn`.

☑ In the optimized hardware do not show comparison units, instead show the individual gates performing the comparison, ☑ optimizing for constant values.

Solution appears to the right with some abbreviated connection names. As in the midterm exam the equality comparisons have been optimized into AND gates. Notice that because $n = 1$ input `str` has a single 4-bit element and so `str[0]` refers to all of `str` which is four bits and carries a single `Char`. Because the default value of `wn` is to be used and `n=1` is given we know that $w_n = \lceil \lg(n+1) \rceil = \lceil \lg(2) \rceil = 1$. That means the `left_in_un_open` (a.k.a. `left_in_n_unmat_open`) is one bit and so it can be directly connected to an AND gate input. If $w_n > 1$ then an OR gate would be needed before the AND gate. *Final exam short answer question?*

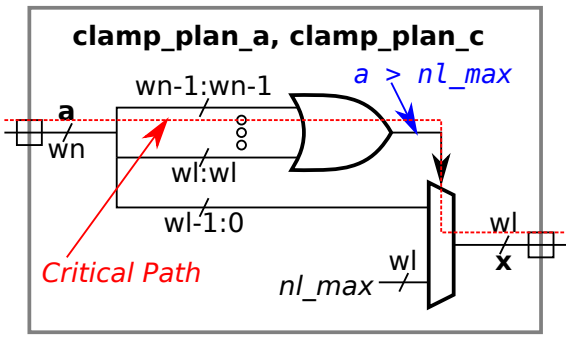


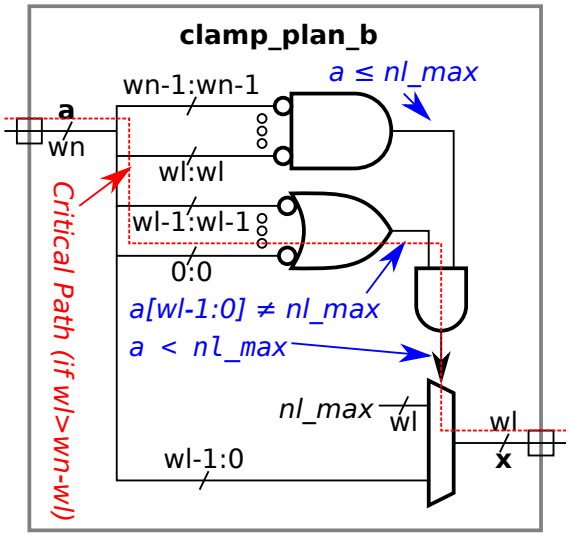
Problem 2: Appearing below are three variations on a module that will set its output to either the input value, or a maximum value if the input is larger. The module will always be instantiated with `wl < wn`. All of them are functionally equivalent, but were synthesized to different costs (by Genus 23.12-s086.1 when similar code was used in the solution to Homework 3). Because they are functionally equivalent a perfect synthesis program would synthesize each to the same hardware (with equal costs).

```
module clamp_plan_a
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a <= nl_max ? a : nl_max;
endmodule

module clamp_plan_b
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = a < nl_max ? a : nl_max;
endmodule

module clamp_plan_c
  #( int wl = 3, wn = 4 ) ( output uwire [wl-1:0] x, input uwire [wn-1:0] a );
   localparam logic [wl-1:0] nl_max = ~(wl)'(0); // Sequence of wl 1s.
   assign x = !a[wn-1:wl] ? a : nl_max;
endmodule
```

☑ Show the optimized hardware for the low-cost version(s).

Solution appears to the right. Because `nl_max` is w_l bits and $w_l > w_n$ it is possible to determine that `a > nl_max` by checking if any of the bits in positions `wn-1:wl` are 1. If so, `a > nl_max`, otherwise `a <= nl_max`. The Plan C hardware explicitly uses this optimization, and is shown to the right. For Plan A the solution is based on the assumption that the synthesis program figures out the optimization. The hardware is more complicated for Plan B because to compute `a<nl_max` the hardware needs to compute `a<=nl_max` (the AND gate) and `a!=nl_max` (the OR gate). The Plan B solution is based on a synthesis program that can optimize the comparison this way, but one that can't figure out that there was no need to check for the `a!=nl_max` case because if `a==nl_max` either multiplexor input can be used.





Solution continued on next page.

☑ Find the simple-model cost of each after optimization. The costs should be ☑ in terms of `wn` and `wl`.

Plan A and Plan C: Multiplexor (one constant input): $w_l\,\mathrm{u_c}$, OR gate $[w_n - w_l - 1]\,\mathrm{u_c}$.

Plan B: Multiplexor (one constant input): $w_l\,\mathrm{u_c}$, "big" AND gate $[w_n - w_l - 1]\,\mathrm{u_c}$, OR gate $w_l\,\mathrm{u_c}$, little AND gate $1\,\mathrm{u_c}$. Note that the big AND gate might just have one input so not only would it not be big, but it would not be an AND gate at all, just a NOT gate.

☑ Find the simple-model delay of each after optimization. The delays should be ☑ in terms of `wn` and `wl`.

Plan A and Plan C: Delay of OR gate, $\lceil \lg(w_n - w_l) \rceil\,\mathrm{u_t}$. Delay of mux, $1\,\mathrm{u_t}$. Critical path, shown in red on the diagram, is through both so the critical path delay is $[\lceil \lg(w_n - w_l) \rceil + 1]\,\mathrm{u_t}$.

Plan B: Delay of "big" AND gate, $\lceil \lg(w_n - w_l) \rceil\,\mathrm{u_t}$. Delay of OR gate $\lceil \lg(w_l) \rceil\,\mathrm{u_t}$. Delay of AND gate $1\,\mathrm{u_t}$, delay of mux, $1\,\mathrm{u_t}$. Assuming $w_l > w_n - w_l$ the critical path, shown on red in the diagram, is through the OR gate, little AND gate and mux for a total delay of $[\lceil \lg(w_l) \rceil + 1 + 1]\,\mathrm{u_t}$. Notice that the critical path length is much larger in Plan B under the reasonable assumption that $w_l > w_n - w_l$.

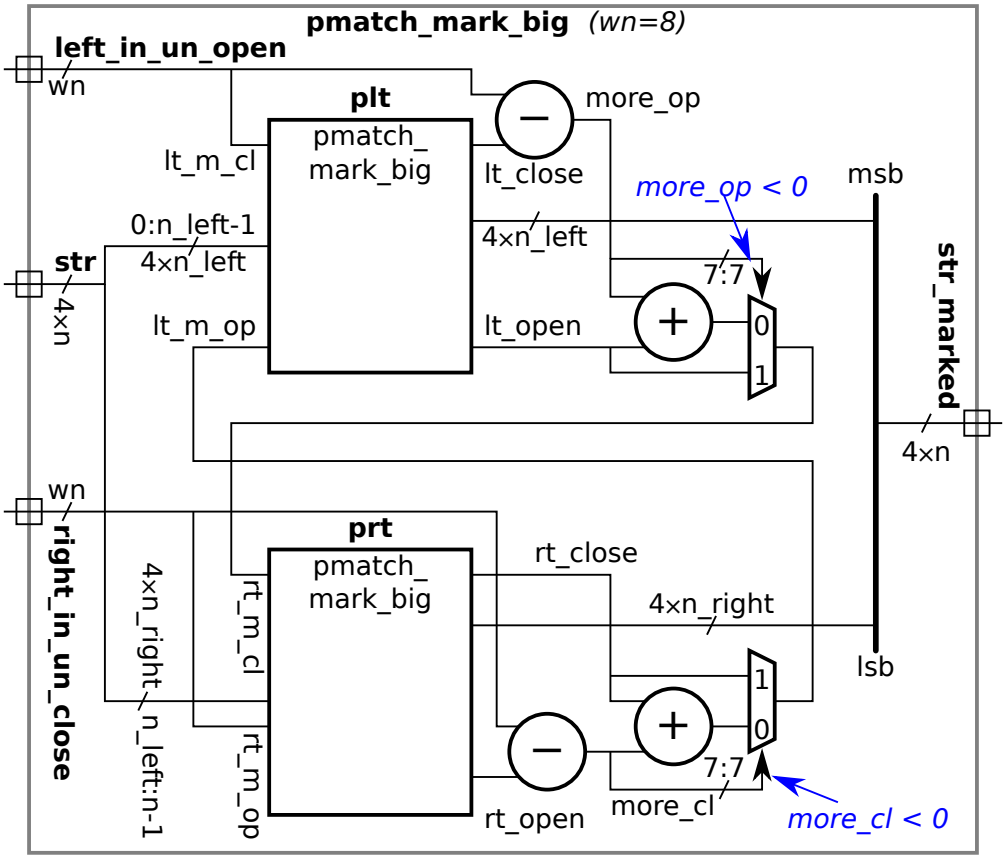
Problem 3: Appearing on the next page is a simplified solution to Homework 3, Problem 2. In this module the number of bits in the connections carrying parentheses counts is hardcoded to 8. Though the hardware is correct for $n < 256$ it is more costly and slower than it needs to be. But for this problem it's good enough.

Show the Homework 3, Problem 2 hardware that will be inferred for this module for $n > 1$ (the non-base case). That is, don't show the hardware computing `left_out_n_unmat_close` and `right_out_n_unmat_open`.

- ☑ Show the inferred hardware at one level for $n > 1$.
- ☑ Feel free to use abbreviations.
- ☑ Don't show the Homework 3 Problem 1 hardware (the last `always_comb`).
- ☑ Don't confuse elaboration-time computation with hardware.

Solution appears below. Note that the condition `more_op<0` can be evaluated by just looking at the most-significant bit, 7 in this case.

To help understand the circuit try tracing the critical path. To do that consider two cases: the root (the top-level instantiation of `pmatch_mark_big`) and a lower-level instantiation (but not the base case). In the root assume that `left_in_un_open` and `right_in_un_close` are available at $t = 0$.



```
module pmatch_mark_big #( int n = 5, wn = 8 )
   ( output logic [wn-1:0] left_out_n_unmat_close, right_out_n_unmat_open,
     output uwire [3:0] str_marked [0:n-1],
     input uwire [wn-1:0] left_in_n_unmat_open, right_in_n_unmat_close,
     input uwire [3:0] str [0:n-1] );

   if ( n == 1 ) begin
      assign left_out_n_unmat_close = str[0] == Char_Close;
      assign right_out_n_unmat_open = str[0] == Char_Open;
      assign str_marked[0] =
             str[0] == Char_Close && left_in_n_unmat_open   ? Char_Close_Okay :
             str[0] == Char_Open  && right_in_n_unmat_close ? Char_Open_Okay  : str[0];
   end else begin

      localparam int n_left = n/2,   n_right = n - n_left;
      localparam int wl = 8,         wr = 8;  // Note: this is wasteful.

      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      logic [wl-1:0] lt_matched_op, lt_matched_cl;
      logic [wr-1:0] rt_matched_op, rt_matched_cl;

      pmatch_mark_big #(n_left,wl) plt   // Recursive Instantiation
        ( lt_close, lt_open, str_marked[0:n_left-1],
          lt_matched_cl, lt_matched_op, str[0:n_left-1] );
      pmatch_mark_big #(n_right,wr) prt  // Recursive Instantiation
        ( rt_close, rt_open, str_marked[n_left:n-1],
          rt_matched_cl, rt_matched_op, str[n_left:n-1] );

      always_comb begin
         logic signed [wn-1:0] more_op, more_cl;

         lt_matched_cl = left_in_n_unmat_open;
         rt_matched_op = right_in_n_unmat_close;

         more_op = left_in_n_unmat_open - lt_close;
         rt_matched_cl = more_op < 0 ? lt_open : more_op + lt_open;

         more_cl = right_in_n_unmat_close - rt_open;
         lt_matched_op = more_cl < 0 ? rt_close : more_cl + rt_close;
      end

      always_comb begin          // Same as Homework 3 Problem 1
         logic signed [wn-1:0] delta;
         delta = lt_open - rt_close;
         left_out_n_unmat_close = delta >= 0 ? lt_close : lt_close - delta;
         right_out_n_unmat_open = delta < 0  ? rt_open  : rt_open + delta;
      end
   end
endmodule
```