Name ______________________

Digital Design using HDLs
LSU EE 4755
Final Examination

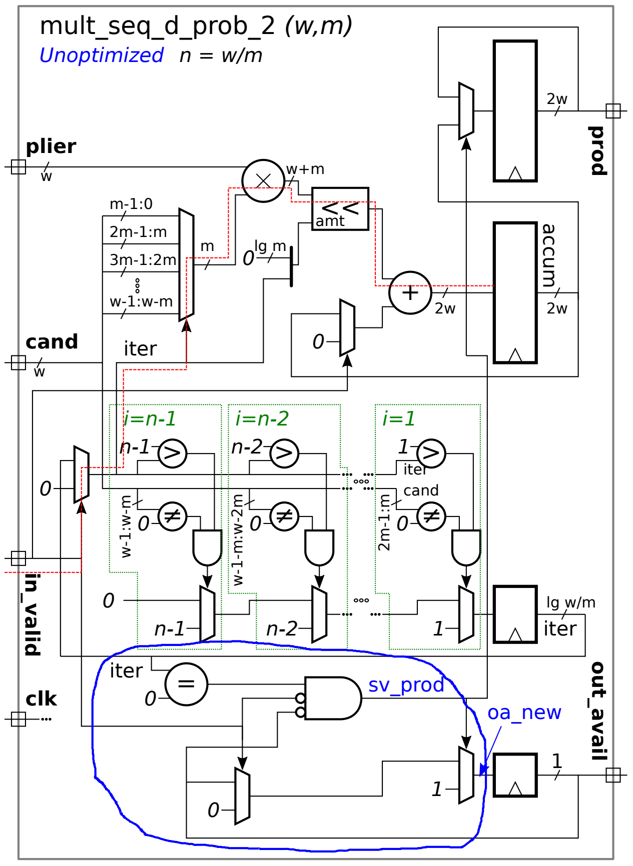
Wednesday, 5 December 2018   15:00-17:00 CST

Problem 1	______	(20 pts)
Problem 2	______	(25 pts)
Problem 3	______	(20 pts)
Problem 4	______	(10 pts)
Problem 5	______	(25 pts)
Exam Total	______	(100 pts)

Alias ______________________

Good Luck!

Problem 1: [20 pts] Appearing to the right is the hardware inferred for the Homework 7 Problem 2 module, the fast sequential multiplier which skipped over zeros in the multiplicand.

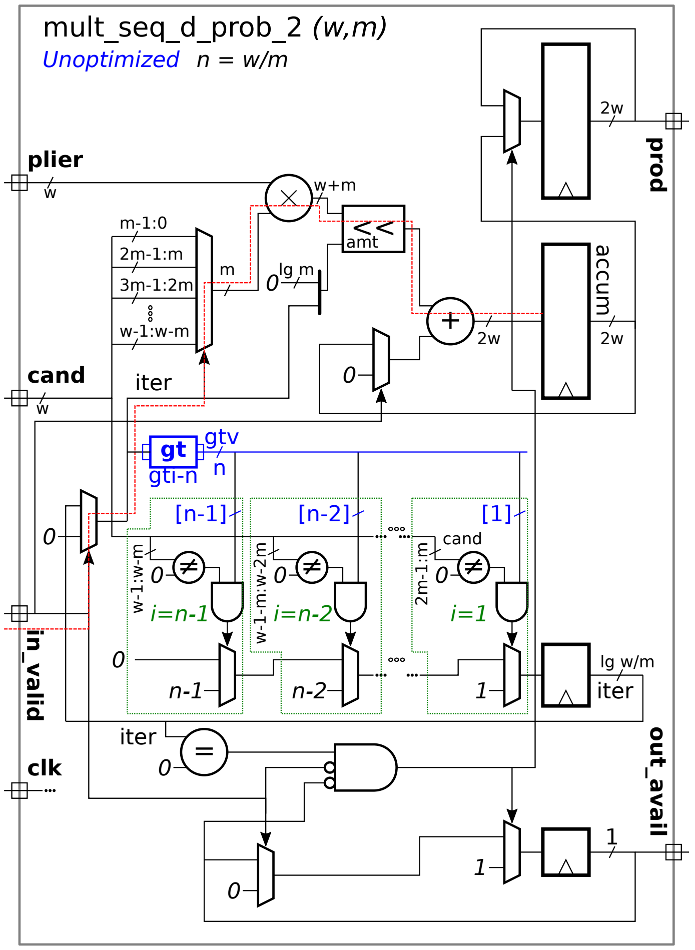


(*a*) Notice that some hardware is circled in blue. Optimize that hardware and show the cost of the optimized hardware. The optimized hardware should generate signals `sv_prod` and `oa_new`. If possible, replace the multiplexors with simpler gates.

☐ Show optimized hardware.

☐ Cost of optimized hardware:

(*b*) In the version of the module appearing below the $\boxed{>}$ units have been replaced by one module, **gt**, the changed hardware appears in blue. As can be inferred from the diagram bit `i` of the output of `gt`, `gtv`, is 1 iff `i>iter`. In the Verilog code below `gt` is instantiated but it is not being used. Modify the Verilog code so that **the existing `for` loop** uses the output of `gt` instead of the > operators. Pay attention to the version of `iter` used by `gt`.



☐ Use `gt` output in existing `for` loop.

☐ Make sure that `gt` uses correct `iter` version.

```
module mult_seq_d_prob_2
  #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     output logic out_avail,
     input uwire clk, in_valid,
     input uwire [w-1:0] plier, cand );

   localparam int n = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(n);
   uwire [n-1:0][m-1:0] cand_2d = cand;
   bit [iter_lg-1:0] iter, next_iter;
   logic [2*w-1:0] accum;

   uwire [n-1:0] gtv;

   uwire [iter_lg-1:0] gt_iter = 0;                //  ☐  FILL IN

   gt #(n,iter_lg) gti( gtv, gt_iter );

   always_ff @( posedge clk ) begin

      if ( in_valid ) begin
         iter = 0;   accum = 0;   out_avail = 0;
      end else if ( !out_avail && iter == 0 ) begin
         prod = accum;   out_avail = 1;
      end

      accum += plier * cand_2d[iter] << ( iter * m );

      next_iter = 0;

      for ( int i=n-1;  i>0;  i-- ) if ( i>iter && cand_2d[i] ) next_iter = i;


      iter = next_iter;
   end
endmodule
```

Problem 2: [25 pts] The point of the **gt** module in the previous problem was to reduce cost, just in case the synthesis program didn't notice that the cost of computing each of `n-1>iter`, `n-2>iter`, ..., `2>iter`, `1>iter`, would be less than $n-1$ times the cost of computing one of them. The recursive module below computes these quantities and can be used for the **gt** module from the previous problem.

```
module gtd_rec #( int n = 16, int lgn = $clog2(n) )
   ( output logic [n-1:0] gt, input uwire [lgn-1:0] iter );
   localparam int nh = n / 2;    // Note: n must be a power of 2.
   if ( n == 2 ) begin
      assign gt[0] = 0;
      assign gt[1] = !iter[0];
   end else begin
      uwire [nh-1:0] gtlo;
      gtd_rec #(nh) glo( gtlo, iter[lgn-2:0] );
      localparam logic [nh-1:0] zeros = 0, ones = -1;
      assign gt = iter[lgn-1] ? { gtlo, zeros } : { ones, gtlo };
   end
endmodule
```

(*a*) Show the hardware that will be inferred for this module for an arbitrary value of n. In this case, do not show what is inside the recursively instantiated module.

☐ Show hardware for arbitrary $n > 2$. (Don't show recursive module contents.)

(*b*) There should be a significant optimization opportunity in the hardware above. Show it.

☐ Show how the hardware will be optimized. The result should be AND, OR, and other basic logic gates.

(*c*) Show the hardware that will be inferred for $n = 8$ after elaboration. That is, show the hardware inside all of the recursive instantiations.

☐ Show hardware for $n = 8$. Show the contents of all recursively instantiated modules.

(*d*) Compute the cost and delay using the simple model. Show these in terms of n assuming that n is a power of 2.

☐ Cost and ☐ delay in terms of n.

Problem 3: [20 pts] Consider the module below.

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,     input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin
      a <= b + c;    // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin
      e = a * f;
   end

endmodule
```

(*a*) Show the hardware that will be inferred for the module above.

☐ Show inferred hardware. ☐ Pay attention to what is and is not a register. ☐ Clearly show module ports.

```
module misc #( int n = 8 )
   ( output logic [n-1:0] a, g, e,
     input uwire [n-1:0] b, c, j, f,    input uwire clk );

   logic [n-1:0] z;

   always_ff @( posedge clk ) begin   // Code Position Label: alf
      a <= b + c;   // Note: nonblocking assignment.
      z = a + j;
      g = z;
   end

   always_comb begin                  // Code Position Label: alc
      e = a * f;
   end

endmodule
```

(*b*) Suppose that the event queue is empty at $t = 10$ when simulating the module above. Show the contents of the event queue for the code above based on the following changes: At $t = 10$ `j` changes. At $t = 12$ `clk` changes from 0 to 1. At $t = 14$ `f` changes.

☐ Show the state of the event queue from $t = 10$ until it is empty.

Problem 4: [10 pts] Answer each question below.

(*a*) The module below is not compilable. Explain why and fix it based on what it looks like it is trying to do.

```
module more
  ( input uwire [5:0] w,
    input uwire [w-1:0] a, b,
    output uwire [w:0] s );

   assign s = a + b;

endmodule
```

☐ Fix the problem.

☐ Describe the problem:

(*b*) The module below is supposed to count cycles but it won't work as written. Describe the problem and fix it.

```
module tic_toc
  ( output logic [7:0] cycles,
    input uwire clk, reset );

   always_comb begin

      if ( reset ) cycles = 0;
      else if ( clk ) cycles = cycles + 1;

   end

endmodule
```

☐ Describe the problem:

☐ Fix the problem.

Problem 5: [25 pts] Answer each question below.

(*a*) Appearing below is synthesis data showing the clock period of degree-m sequential workfront multipliers and degree-m sequential regular (dm) multipliers for sizes $m = 1$, $m = 2$, $m = 4$, and $m = 8$.

Module Name	Area	Period Target	Period Actual	Total Latency
mult_seq_wfront_m_w32_m1	191334	1000	3766	241024
mult_seq_wfront_m_w32_m2	205303	1000	3857	123424
mult_seq_wfront_m_w32_m4	260182	1000	5266	84256
mult_seq_wfront_m_w32_m8	351910	1000	7031	56248
mult_seq_dm_w32_m1	246818	1000	31113	995616
mult_seq_dm_w32_m2	279486	1000	30994	495904
mult_seq_dm_w32_m4	314724	1000	32127	257016
mult_seq_dm_w32_m8	408659	1000	31251	125004

As m increases the clock period of the workfront multiplier increases by a significant amount, while the period of the sequential multiplier barely changes. Why?

☐ Why does the workfront period increase so much more than that of the regular multiplier?

Let $p_w(m)$ and $p_r(m)$ denote the clock period of the degree-m workfront and regular multipliers. Show expressions for $l_w(m)$ and $l_r(m)$, the latencies of these multipliers.

☐ Finish the following expression for latency: $l_w(m) = p_w(m)$

☐ Finish the following expression for latency: $l_r(m) = p_r(m)$

(*b*) The reasoning in the statement below is, as of this writing, incorrect. Provide the correct reason to not spend time on multiplier modules.

"One should not spend time trying to develop efficient multiplication hardware because the synthesis program is very good at optimizing logic and will synthesize something at least as good as a human can."

☐ When working on a design that makes heavy use of multiplication one should just use multiplication operators and not try to implement your own because:

(*c*) Sequential multipliers S0 and S1 have the same latency and cost, but the clock period for S1 is lower than S0.

☐ Which is preferred? ☐ Explain.

Pipelined multipliers P0 and P1 have the same latency and cost, but the clock period for P1 is lower than P0.

☐ Which is preferred? ☐ Explain.

(*d*) In the module below notice that `cand_2d` is no longer available. Modify the line updating `accum` to use `cand` instead.

```
module mult_seq_dm #( int w = 16, int m = 2 )
   ( output logic [2*w-1:0] prod,
     input uwire [w-1:0] plier, cand, input uwire clk);

   localparam int iterations = ( w + m - 1 ) / m;
   localparam int iter_lg = $clog2(iterations);

   // uwire [iterations-1:0][m-1:0] cand_2d = cand;

   bit [iter_lg:1] iter;
   logic [2*w-1:0] accum;

   always @( posedge clk ) begin

      if ( iter == iter_lg'(iterations) ) begin
        prod = accum; accum = 0; iter = 0;
      end

      //  ☐   Fix line below

      accum += plier *       cand_2d[iter]   << ( iter * m );

      iter++;
   end
endmodule
```