Name ______________________________

Digital Design using HDLs
LSU EE 4755
Final Examination

Wednesday, 6 December 2017 15:00-17:00 CST

Problem 1	________	(15 pts)
Problem 2	________	(25 pts)
Problem 3	________	(20 pts)
Problem 4	________	(10 pts)
Problem 5	________	(30 pts)
Exam Total	________	(100 pts)

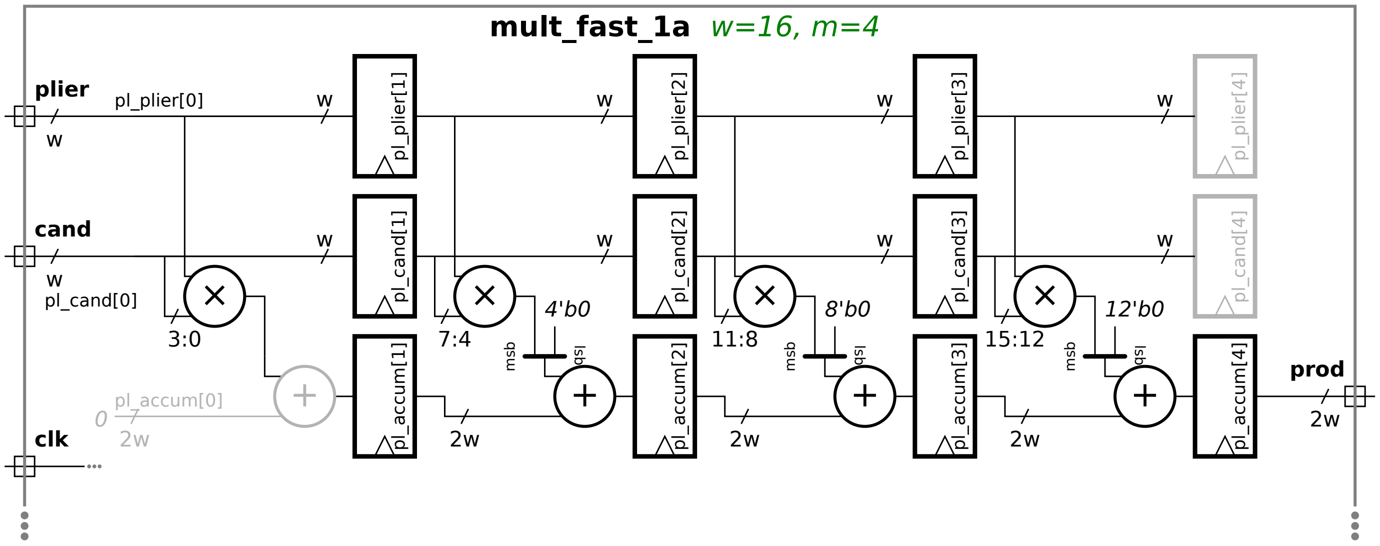
Alias ______________________________

Good Luck!

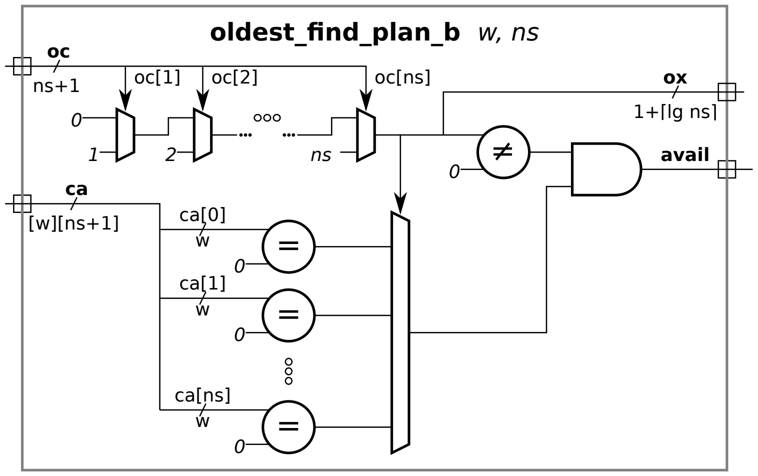
Problem 1: [15 pts] The Verilog code below is the solution to Problem 1a of Homework 7. Below that is the hardware **for a slightly different pipelined multiplier**. Modify the hardware to match the Verilog code. Changes need to be made for each line commented DIFFERS.

☐ Modify hardware to reflect Verilog.

```
module mult_fast_1a #( int w = 16, int m = 4 )
   ( output uwire [2*w-1:0] prod,
     output uwire out_avail,        input uwire clk, in_valid,      // ☐ DIFFERS
     input uwire [w-1:0] plier, cand );
   localparam int nstages = ( w + m - 1 ) / m;
   logic [2*w-1:0] pl_accum[0:nstages];
   logic [w-1:0] pl_plier[0:nstages], pl_cand[0:nstages];
   logic pl_occ[0:nstages];                                        // ☐ DIFFERS

   assign prod = pl_accum[nstages];
   assign out_avail = pl_occ[nstages];                             // ☐ DIFFERS

   always_ff @( posedge clk ) begin
      pl_occ[0] = in_valid;                                        // ☐ DIFFERS
      pl_accum[0] = 0;    pl_plier[0] = plier;    pl_cand[0] = cand;

      for ( int stage=0; stage<nstages; stage++ ) begin
         pl_plier[stage+1] <= pl_plier[stage];
         pl_accum[stage+1] <= pl_accum[stage] + ( pl_plier[stage]
             * pl_cand[stage][m-1:0] << stage*m );                 // ☐ DIFFERS
         pl_cand[stage+1]  <= pl_cand[stage] >> m;                 // ☐ DIFFERS
         pl_occ[stage+1]   <= pl_occ[stage];                       // ☐ DIFFERS
      end
   end
endmodule
```



Problem 2: [25 pts] Module `oldest_find_plan_b`, illustrated below, is based on **an alternative solution** to Homework 7 Problem 1b. Below the hardware illustration is incomplete Verilog code for this module. The Verilog code uses abbreviated names, such as `ns`, comments show the original names from the assignment, such as `nstages`. Complete the module. *Note: This problem can be solved without having ever seen Homework 7, though not as quickly.*



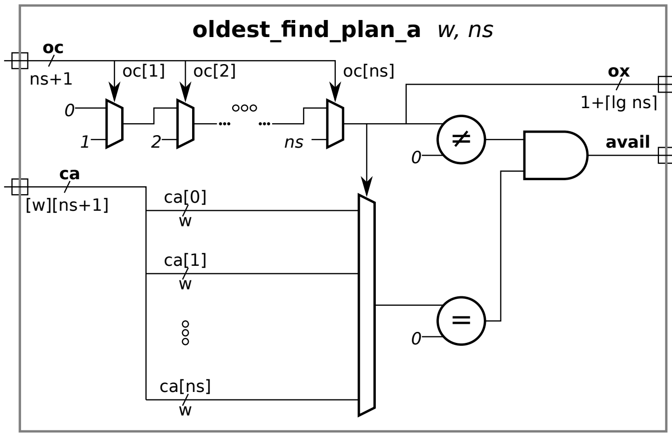
☐ Complete the module so that it matches the hardware above.

```
module oldest_find_plan_b
  #( int w = 15, int ns = 3            /* nstages */ )
   ( output logic [$clog2(ns):0] ox,   // oldest_idx
     output uwire avail,               // out_avail
     input uwire oc[0:ns],             // pl_occ
     input uwire [w-1:0] ca[0:ns] );   // pl_cand









endmodule
```

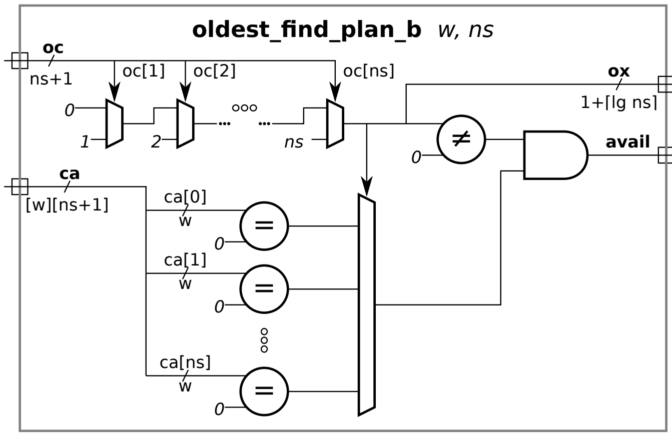
Problem 3: [20 pts] Appearing below are two variations on the oldest index module from the previous problem. The Plan A version is based on the code from the posted Homework 7 solution. The Plan B module is slightly different.

(*a*) Compute the cost of each module based on the simple model after optimizing for constant values. Use symbol w (for `w`) and n (for `ns`). Base the cost of an α-input, β-bit multiplexor on the tree (recursive) implementation. Recall that the tree implementation consists of $\alpha - 1$ two-input multiplexors arranged in a tree.

☐ Plan A cost in terms of w and n. ☐ Show cost components on diagram, such as cost of big mux, ☐ don't forget to account for the constant inputs, and ☐ for the number of bits in each wire.

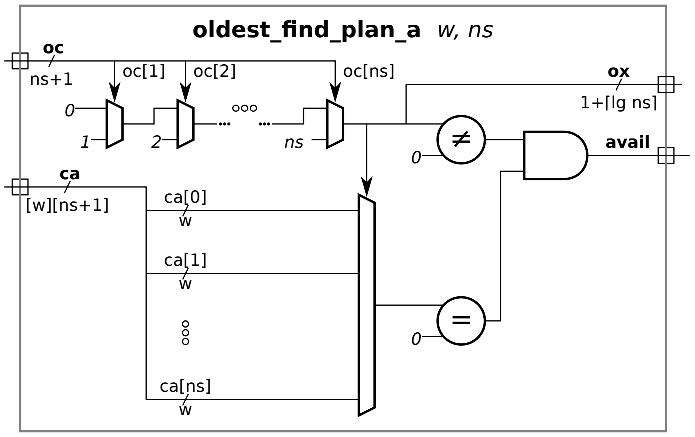


☐ Plan B cost in terms of w and n. ☐ Show cost components on diagram, such as cost of big mux, ☐ don't forget to account for the constant inputs and, ☐ for the number of bits in each wire.

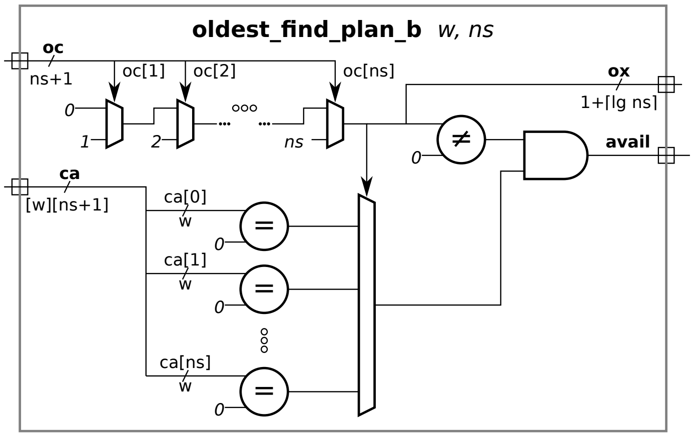


(*b*) Show the delay along all paths and show the critical path. Compute delay based on the simple model after optimizing for constant values. Use the tree mux described in the previous part.

☐ Plan A: ☐ show delay along all paths, ☐ highlight the critical path, ☐ and show the delay through each component. Show these ☐ in terms of w and n, and ☐ account for constant inputs such as the zeros in the equality units.



☐ Plan B: ☐ show delay along all paths, ☐ highlight the critical path, ☐ and show the delay through each component. Show these ☐ in terms of w and n, and ☐ account for constant inputs such as the zeros in the equality units.



Problem 4: [10 pts] Explain why each of the modules below is not synthesizable by Cadence Encounter (or similar tools) and modify the code so that it is *without changing what the module does*. *Note: The warning about not changing what the module does was not in the original exam.*

```
module one_run #( int w = 16, int lw = $clog2(w) )
   (output logic all_1s, input uwire [w-1:0] a, input uwire [lw:0] start, stop );
   always_comb begin

      all_1s = 1;

      for ( int i=start; i<stop; i++ )
        all_1s = all_1s && a[i];

   end
endmodule
```

☐ Reason code above is not synthsizable:

☐ Modify code so that it is.

```
module running_sum #( int w = 32 )
   ( output logic [w-1:0] rsum,
     input uwire [w-1:0] a,     input uwire reset, clk );

   always @( posedge clk ) begin
      if ( reset ) rsum <= 0;
   end

   always @( posedge clk ) begin
      rsum <= rsum + a;
   end

endmodule
```

☐ Modify code so that it is synthsizable.

☐ Reason code above was not synthsizable:

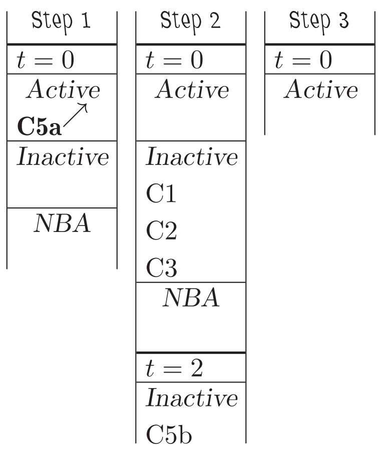
☐ Explain assumption about intended behavior of this module.

Problem 5: [30 pts] Answer each question below.

(*a*) Show when each piece of code below executes (use the `C` labels) up until the start of `C5c`, and show when and in which region each piece is scheduled. See the table below.

```
module eq;
   logic [7:0] a, b, c, d, x, y, x1, x2, y1, y2, z2;
   always_comb begin         // C1
      x1 = a + b;
      y1 = 2 * b;
   end
   assign x2 = 100 + a + b;  // C2
   assign y2 = 4 * b;        // C3
   assign z2 = y2 + 1;       // C4
   initial begin
      //                        C5a
      a = 0;
      b = 10;
      #2;
      //                        C5b
      a = 1;
      b <= 11;
      #2;
      //                        C5c
      a = 2;
      b = 12;
   end
endmodule
```

☐ Continue the diagram below so that it shows scheduling up to the point where `C5c` executes.



(*b*) Which of the two modules does what it looks like it's trying to do? Explain.

```
module sa1(input logic [7:0] a, b, c, d, output wire [7:0] x, y );
   assign x = a + b;
   assign y = 2 * x;
   assign x = c + d;
endmodule

module sa2(input logic [7:0] a, b, c, d, output logic [7:0] x, y );
   always_comb begin
      x = a + b;
      y = 2 * x;
      x = c + d;
   end
endmodule
```

☐ Module that is probably correct is:

☐ Major problem with other module.

☐ Provide a possible wrong answer from other module.

(*c*) Define throughput and latency and indicate where each is preferred. Provide examples appropriate for pipelined systems.

☐ Throughput is:

☐ For example:

☐ Latency is:

☐ For example,

☐ If the goal is to improve throughput is higher throughput good or bad?

☐ If the goal is to improve latency, is higher latency good or bad?

☐ In what situation is latency more important than throughput?

(*d*) When we synthesize we specified a target delay, for example, 400 ns.

☐ Does specifying a larger delay mean that there will be less optimization?

☐ Explain.