Name ______________________________

Digital Design using HDLs
EE 4755
Midterm Examination

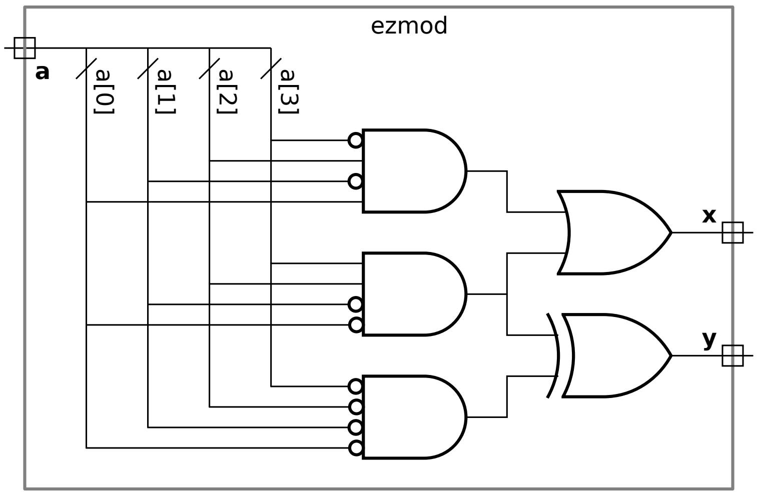
Wednesday, 28 October 2015 11:30–12:20 CDT

Problem 1	________	(20 pts)
Problem 2	________	(20 pts)
Problem 3	________	(20 pts)
Problem 4	________	(20 pts)
Problem 5	________	(20 pts)
Exam Total	________	(100 pts)

Alias ______________________________

Good Luck!

Problem 1: [20 pts] Complete the Verilog description of the hardware illustrated below. It's okay—and a time saver—to use the == operator.



☐ Complete the port declarations.

☐ Complete the module.

```
module ezmod

  ( output                                                  // DON'T FORGET
    input                                               );  // THE PORTS




endmodule
```

Problem 2: [20 pts] Consider the module below.

```
module ssum #( int n = 3,      int f = 4,      int swid = f + $clog2(n) )
             ( output logic [swid-1:0] sum,
               input uwire [n-1:0] mask,      input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

(*a*) Show the hardware that will be synthesized without optimization and using default parameters.

☐ Hardware without optimization.

(*b*) Show the hardware that will be synthesized using the default parameters with optimization. In particular, try to make use of a four-input multiplexor for the first two iterations of the `i` loop.

☐ Hardware with optimization and using a four-input mux.

Problem 3: [20 pts] Appearing below is the `ssum` module from the previous problem and the start of a recursive version of the module, `ssum_rec`. Finish `ssum_rec` so that it performs the same computation, but does so using a tree connection of hardware rather than the linear connection that `ssum` describes. (For **partial credit only** use a generate loop to instantiate `ssum` modules of a fixed size; for full credit use recursion.)

```
module ssum #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,       input uwire [f-1:0] a[n] );
   always @* begin
      sum = 0;
      for ( int i=0; i<n; i++ ) if ( mask[i] ) sum += a[i];
   end
endmodule
```

☐ Complete module so that it describes a tree structure specified using recursion.

```
module ssum_rec
  #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,
     input logic [f-1:0] a [n-1:0] );









endmodule
```

Problem 4: [20 pts] Show the hardware that will be synthesized for the module below.

```
module yam( output logic [7:0] x, y, z,
            input uwire [7:0] a, b, c,   input uwire [1:0] op,   input uwire run, clk );
   logic [7:0] x1, x2, e;

   always_ff @( posedge clk ) begin
      e = b;
      z = a + b;
      if ( op == 0 )          e = z;
      else if ( op == 1 )   e = a + x;
      else if ( op == 2 )   e = a + x1;
      x2 = x1;
      x1 = x;
      if ( run ) x = e;
   end

   always_comb y = x1 + x2 - c;

endmodule
```

☐ Show hardware, including ☐ registers and ☐ module ports.

Problem 5: [20 pts] Answer each question below.

(*a*) Show the values of `a`, `b`, and `c` when the code reaches Point 1 and Point 2.

```
module short_answers;
   int a, b, c;
   initial begin
      a = 0;  b = 0; c = 0;
      a = 1;
      a <= 2;
      a <= #3 3;     //
      b = a + 10;    //   ---a---    ---b---    ---c---
      c <= a + 20;   //

      // Point 1:

      #1;

      // Point 2.
   end

   my_prog my_prog_instance(a,b,c); // Ignore for part (a).

endmodule
```

☐ At Point 1, values for ☐ a, ☐ b, and ☐ c.

☐ At Point 2, values for ☐ a, ☐ b, and ☐ c.

(*b*) The definition of the `my_prog` program from the previous part appears below. Show the contents of the Verilog event queue at Point 1 in the code from the previous part, include the effect of code in `short_answers` as well as `my_prog`. Show events in the form "$t = 1969$, region=NL-East, Resume Point 3" and "$t = 2015$, region=X, Update variable z," but use real region names.

```
program my_prog(input int a, b, c);
   initial forever @( a or b or c ) begin
      // Point 3;
      $display("Let's go Mets!");
   end
endprogram
```

☐ Contents of event queue at Point 1, show ☐ region names and ☐ time stamps.

(*c*) The module below is in explicit structural form, in which only primitive gates (and module instantiations) are used. Will the synthesis program synthesize exactly that arrangement of gates? Explain.

```
module bfa_structural( output uwire sum, cout, input uwire a, b, cin );
   uwire term001, term010, term100, term111;
   uwire ab, bc, ac;
   uwire na, nb, nc;
   not n1( na, a);
   not n2( nb, b);
   not n3( nc, cin);
   and a1( term001, na, nb, cin);
   and a2( term010, na, b,  nc);
   and a3( term100, a,  nb, nc);
   and a4( term111, a,  b,  cin);
   or o1( sum, term001, term010, term100, term111);
   and a10( ab, a, b);
   and a11( bc, b, cin);
   and a12( ac, a, cin);
   or o2( cout, ab, bc, ac);
endmodule
```

☐ Will synthesis program emit exactly these gates? ☐ Explain.

(*d*) Based on a hand analysis of `my_mut` we expect it to have a clock period of 12 ns. Shown below is an excerpt from the testbench for `my_mut` that includes the code for generating a clock. Assume that the Verilog time unit is set to 1 ns. How does the clock declaration below affect the timing of the synthesized hardware?

```
module testbench();
   logic clock;
   initial clock = 0;
   always #5 clock = !clock;
   // Other declarations omitted.
   my_mut woof(x,y,a,b,clock);
```

☐ The effect of the declaration of `clock` on timing of synthesized hardware is ... ☐ because ....