Name ____________________________________

Digital Design using HDLs
LSU EE 4755
Midterm Examination

Friday, 26 October 2018 9:30–10:20 CDT

Problem 1	__________	(22 pts)
Problem 2	__________	(20 pts)
Problem 3	__________	(23 pts)
Problem 4	__________	(10 pts)
Problem 5	__________	(25 pts)
Exam Total	__________	(100 pts)

Alias ____________________________________

Good Luck!

Problem 1: [22 pts] The illustration below shows some of the inferred hardware for the `behav_merge` module from the solution to Homework 6. The hardware that's shown is for typical iterations `i` and `i+1`. Show the hardware for iterations `i=0` and `i=1` with optimizations applied.

☐ Show hardware for iterations `i=0` and `i=1`.

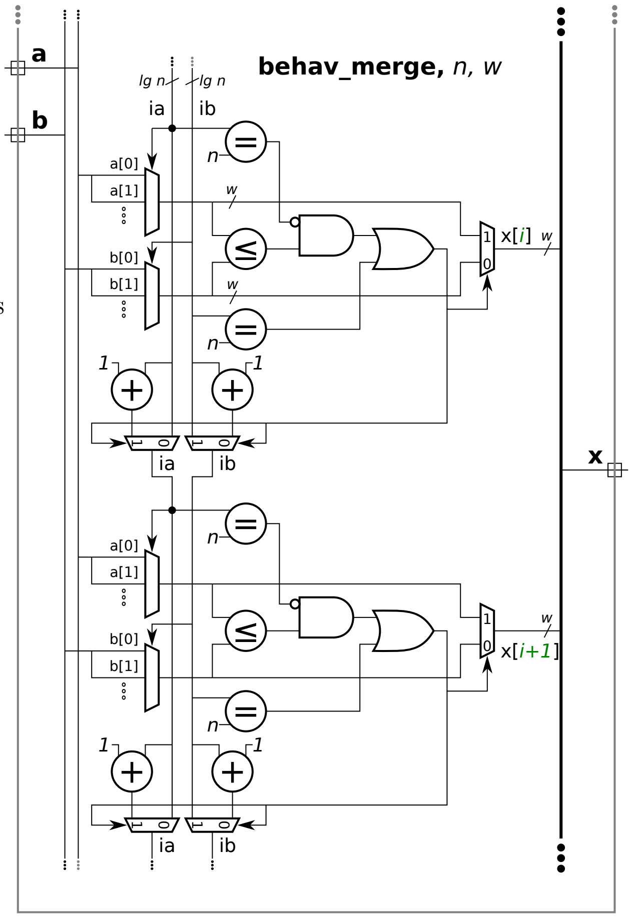
☐ Also show hardware for code before `for` loop.

☐ Optimize hardware. Take into account possible values of `ia` and `ib`.

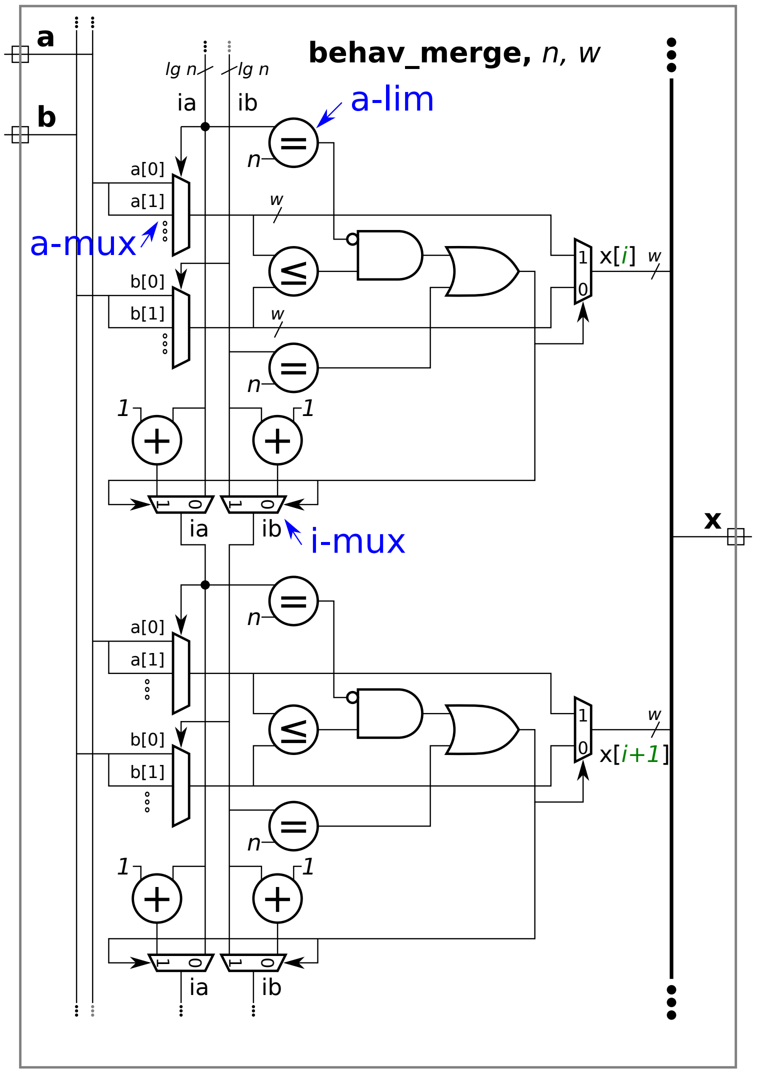
```
module behav_merge
 #( int n = 4, int w = 8 )
  ( output logic [w-1:0] x[2*n],
    input uwire [w-1:0] a[n], b[n] );

  logic [$clog2(n+1)-1:0] ia, ib;
  always_comb begin
   ia = 0; ib = 0;
   for ( int i = 0;  i < 2*n;  i++ )
     if ( ib==n || ia!=n && a[ia]<=b[ib] )
       x[i] = a[ia++]; else x[i] = b[ib++];
  end

endmodule
```



Problem 2: [20 pts] Appearing once again is part of the Homework 6 solution, this time with items labeled in blue. Show the cost and delay of these, as requested below. See the previous problem for the Verilog description. The phrase *most expensive* means for the value of i for which the device needs all inputs, even after optimization. For the mux, show the cost and delay for the tree implementation.



☐ Cost of most expensive `a-mux` in terms of n and w.

☐ Delay of most expensive `a-mux` in terms of n and w.

☐ Cost of most expensive `i-mux` in terms of n and w.

☐ Delay of most expensive `i-mux` in terms of n and w.

☐ Cost of most expensive `a-lim` in terms of n and w ☐ after optimizing for constant inputs.

☐ Delay of most expensive `a-lim` in terms of n and w ☐ after optimizing for constant inputs.

Problem 3: [23 pts] Output `lt` of module `comp`, below, should be 1 iff `a` is strictly less than `b`, and `eq` should be 1 iff `a==b`. Both `a` and `b` are unsigned integers. The module recursively instantiates two instances of itself, one is supposed to compare the low bits of the inputs, the other compares the high bits. Complete the module so that it works for any positive `w`.

☐ Complete the module, don't miss the ☐ `FILL IN` items.

☐ Make sure that it works for odd and even values of `w`.

```
module comp
  #( int w = 8 )
   ( output uwire lt, eq, input uwire [w-1:0] a, b );

   if (       ) begin   // Terminating Case Condition                <----  ☐  FILL IN

      assign lt = !a && b;
      assign eq = a == b;

   end else begin

      uwire llo, lhi, elo, ehi;



      // Instantiate two comp modules,  connect each to about half the inputs.
      //
      //     ----                    --------------      -------------- <--  ☐  FILL IN
      comp #(    ) clo( llo, elo, a[              ],  b[              ] );

      comp #(    ) chi( lhi, ehi, a[              ],  b[              ] );



      assign lt =                        ;                       //    <----  ☐  FILL IN

      assign eq =                        ;                      //    <----  ☐  FILL IN

   end
endmodule
```

Problem 4: [10 pts] The output of `plus_amt`, `x`, is to be set to `b + amt`. Input `b` and output `x` are expected to be in IEEE 754 double FP format (the same format as type real). (Note: the port declarations are not to be modified in the problems below.) Several variations on the module appear below. *Hint: Solution to this problem require the correct use of* `realtobits` *and/or* `bitstoreal`. *Grading Note: The bonus problem was not on the original exam.*

(*a*) The module below does not compute the correct result. Fix the module by modifying the `always_comb` block. The module does not need to be synthesizable.

☐ Fix so that `x` is assigned the correct result, `amt` plus value of `b`.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b ); // DO NOT modify ports.
   // Both x and b are IEEE 754 doubles (reals).

   always_comb begin
      // Change code below.



      x = b + amt;



   end

endmodule
```

(*b*) [0 pts] $\boxed{\text{Bonus Problem}}$ Complete the module below so that it uses the `CW_fp_add` module to do the addition. The parameters to `CW_fp_add` are already correct, just connect the inputs and outputs.

☐ Complete so that it computes the correct result.

```
module plus_amt
  #( real amt = 1.5 )
   ( output logic [63:0] x, input uwire [63:0] b ); // DO NOT modify ports.
   // Both x and b are IEEE 754 doubles (reals).



   uwire logic [7:0] s; // Unused.
   // Computes z = a + b.
   CW_fp_add #(.sig_width(52),.exp_width(11)) // This line correct, don't change.
    fadd( .status(s), .rnd(0),                // This line correct, don't change.
         .z(        ), .a(        ), .b(        ) );



endmodule
```

Problem 5: [25 pts] Show the hardware that will be inferred for the Verilog code below.

☐ Clearly show module ports.

☐ Show inferred hardware. Don't optimize.

☐ Pay close attention to what is and is not inferred as a register.

```
module regs #( int w = 10, int k1 = 20, int k2 = 30 )
   ( output logic [w-1:0] y,
     input logic [w-1:0] b, c,
     input uwire clk );

   logic [w-1:0] a, x, z;

   always_ff @( posedge clk ) begin

      a = b + c;
      if ( a > k1 ) x = b + 10;
      if ( a > k2 ) z = b + x; else z = c - x;
      y = x + z;

   end

endmodule
```