Name _____ *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 27 October 2021, 11:30-12:20 CDT

Problem 1	_____	(25 pts)
Problem 2	_____	(30 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(10 pts)
Exam Total	_____	(100 pts)

Staple This Side

Alias _____

☣ $r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [25 pts] Appearing in this problem are two variations on hardware that selects one of four inputs, `i`, based on the position of the least-significant `1` in a 4-bit quantity, `fmt`. This is similar to the hardware needed in the solution to Homework 2, except that here `i[3]` can be selected.

```
module nn_sparse #( int w = 20 )
   ( output logic [w-1:0] o, input uwire [w-1:0] i[4], input uwire [3:0] fmt );
```

(*a*) Show the hardware that will be inferred for `is0` and show that hardware after optimization.

```
   uwire [w-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : fmt[2] ? i[2] : i[3];
```

☐ Show inferred hardware.

☐ Show optimized hardware. Hardware can be re-arranged to reduce delay.

☐ Use only basic logic gates and multiplexors.

(*b*) Compute the cost and delay of the optimized hardware for `is0` in terms of w. (That's w, not its default value.)

☐ In terms of w cost is:

☐ In terms of w delay is:

(*c*) Appearing below is an alternative design. Net `is0b` will have the same value as `is0`. Show the hardware below before and after optimization. For `isi0` do not show multiplexors after optimization. For `is0b` use two-input multiplexors (as many as needed).

```
uwire [1:0] isi0 = fmt[0] ? 0 : fmt[1] ? 1 : fmt[2] ? 2 : 3;
uwire [w-1:0] is0b = i[isi0];
```

☐ Show inferred hardware.

☐ Show optimized hardware, optimize to reduce delay.

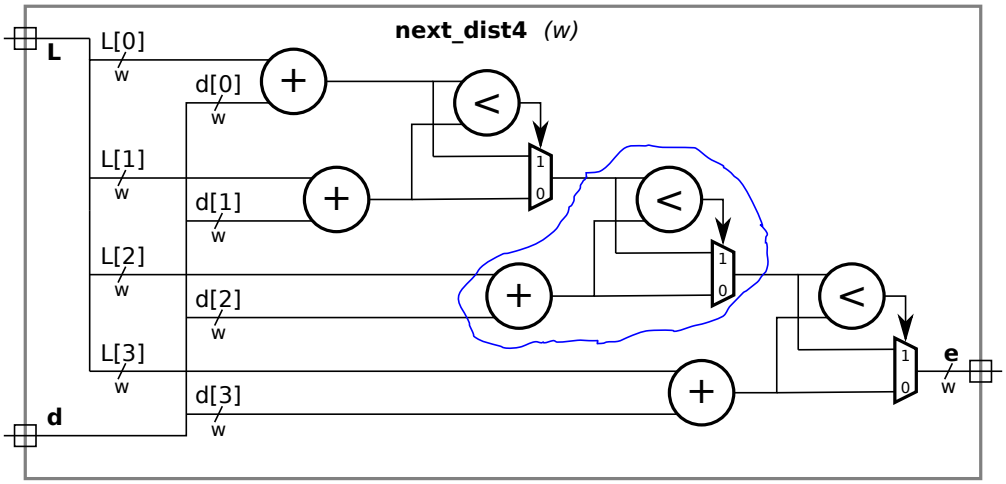
☐ Use basic logic gates and ☐ no muxen for `isi0` and ☐ two-input muxen (plus other logic) for `is0b`.

(*d*) Compute the cost and delay of the optimized hardware (from the previous part) in terms of w. (That's w, not its default value.)

☐ In terms of w cost is:

☐ In terms of w delay is:

Problem 2: [30 pts] The `next_dist4` hardware illustrated below consists of several duplicated pieces of hardware, one of which is circled. Call the circled hardware an *ami* unit (for add-minimum).



(*a*) Compute the cost and delay of the module using the simple model, and show the critical path on the illustration. Assume that the adder and comparison units are based on ripple adders.

☐ Cost in terms of w:

☐ Show critical path. ☐ Delay in terms of w:

☐ Account for any cascading ripple units.

(*b*) Appearing below are two incomplete modules, one is an `ami` module the other is the `next_dist4` module. Complete these modules to match the diagram using as many `ami` modules as needed. The `ami` module can use procedural or implicit structural code. The `next_dist4` module must instantiate and use `ami` modules but can contain procedural or implicit structural code.

☐ Complete the `ami` module so that it matches the circled hardware.

☐ Complete the `next_dist4` module using as many `ami` modules as needed.

☐ Don't forget to ☐ declare any intermediate objects that are used.

☐ Noting that there are four adders and the width of each wire is `w`, ☐ declare and use parameters appropriately.

```
module next_dist4 #( int w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[4], input uwire [w-1:0] d[4] );









endmodule

module ami









endmodule
```

Staple This Side

Staple This Side

(*c*) Incomplete module `next_dist` is a generalization of `next_dist4` to `n` elements per input. The module includes a generate loop. Use that loop to instantiate `ami` modules so that it performs the correct calculation. Keep the loop simple, don't try to fix the delay problem.

☐ Complete module, taking advantage of the generate loop.

☐ Be sure to instantiate `ami` modules, ☐ connect the first `ami` correctly, ☐ and don't leave `e` unconnected.

```
module next_dist #( int n = 20, w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[n], input uwire [w-1:0] d[n] );

   localparam logic [w-1:0] mv = ~w'(0); // Can use as input to first ami.

   uwire [w-1:0]

   for ( genvar i=0; i<n; i++ ) begin

   end

endmodule
```


Staple This Side

Problem 3: [10 pts] Consider the `with_assign` module below.

```
module with_assign #( int w = 10 )
                     ( output uwire [w-1:0] g, input uwire [w-1:0] b, c );

   uwire [w-1:0] a, f;

   assign g = f | c;  // Line 1
   assign f = a * c;  // Line 2
   assign a = b + c;  // Line 3

endmodule
```

(*a*) Why might the module confuse or annoy humans?

☐ `with_assign` could be confusing because:

(*b*) The module makes extra work for simulators too. Suppose that the input values to `with_assign`, `b` and `c`, change at $t = 10$. About how many times will each line below execute in a worst-case scenario? Don't waste time looking for a precise answer.

☐ About how many times does each line execute? ☐ Explain.

(*c*) Complete the `sans_assign` routine below so that it does the same thing as `with_assign` but is less confusing and less work for simulators.

☐ Complete routine below. (Yes, it's easy but not trivial.)

```
module sans_assign #( int w = 10 )
                     ( output uwire [w-1:0] g, input uwire [w-1:0] b, c );

   uwire [w-1:0] a, f;

   always_comb begin




   end
endmodule
```

☐ Why does `sans_assign` make less work for the simulator than `with_assign`?

Problem 4: [10 pts] Appearing below is an ordinary multiplier, followed by a multiplier that is naïvely designed to take advantage of special cases (first operand is 0 or 1), followed by a module that instantiates both.

```
module mult #( int w = 32 )
             ( output logic [w-1:0] p, input uwire [w-1:0] a, b );
   always_comb p = a * b;
endmodule

module mult_1a #( int w = 32 )
               ( output logic [w-1:0] p, input uwire [w-1:0] a, b );

   always_comb begin
      if ( a == 0 ) p = 0;
      else if ( a == 1 ) p = b;
      else p = a * b;
   end

endmodule

module nm #( int w = 32, logic [w-1:0] c = 12 )
          ( output uwire [w-1:0] prods[4], input uwire [w-1:0] a[4], b[4] );
   mult #(w)      m1 ( prods[0], a[0], b[0] );
   mult #(w)      m2 ( prods[1], c,    b[1] );
   mult_1a #(w)   ma1( prods[2], a[0], b[0] );
   mult_1a #(w)   ma2( prods[3], c,    b[1] );
endmodule
```

☐ Explain why `m1` will be faster (lower delay) than `ma1`, even when possible values of `a[0]` include `0`, `1`, and other values. Assume good synthesis programs.

☐ How will the cost and performance of `m2` and `ma2` compare (to each other) using good synthesis programs? That is, ☐ which should be chosen when delay is the only concern and, ☐ which of the two should be chosen when cost is the only concern. The answer should not depend on any particular value of `c`.

Staple This Side

Staple This Side

Problem 5: [15 pts] Answer the following questions about Verilog syntax and semantics.

(*a*) Appearing below are four variations on a multiplier with a constant input. Most have errors that would prevent them from compiling. For each indicate whether there is an error, and if so, what the error is and a minimal fix.

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2a #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          p = 0;
   else if ( a == 1 )     p = b;
   else                   p = a * b;

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2b #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   always_comb begin
      if ( a == 0 )       p = 0;
      else if ( a == 1 )  p = b;
      else                p = a * b;
   end

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2c #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( b == 0 )          p = 0;
   else if ( b == 1 )     p = a;
   else                   p = a * b;

endmodule
```

☐ Module is ◯ *correct* or ◯ *has the following error and fix:*

```
module mult_2d #( int w = 32, logic [w-1:0] a = 12 )
   ( output uwire [w-1:0] p, input uwire [w-1:0] b );

   if ( a == 0 )          assign p = 0;
   else if ( a == 1 )     assign p = b;
   else                   assign p = a * b;

endmodule
```


Staple This Side

(*b*) Show the values of `b` and `c` where requested below.

```
module assortment;
   logic [15:0] a;
   logic [0:15] b;
   logic [16:1] c;

   initial begin

      a = 16'h1234;
      b = a;
      c = a;
      //  ☐   Show value of b and c after line above executes:




      #1; // Not really needed.
      for ( int i=0; i<16; i++ ) b[i] = a[i];
      //  ☐   Show value of b after line above executes:



   end
endmodule
```

Staple This Side

Staple This Side

Problem 6: [10 pts] Answer the following synthesis questions.

(*a*) Cadence Genus defines the following three synthesis steps: `syn_gen` (generic), `syn_map` (mapped, or technology mapping), and `syn_opt` (optimized). Answer the following questions about technology mapping.

☐ Explain what happens during technology mapping.

☐ Even if optimization were done before technology mapping why is it important optimize after technology mapping?

(*b*) What is the big disadvantage of setting the delay target too low when performing synthesis? (The small disadvantage is that it takes a longer time to run.)

☐ Disadvantage of setting delay target too low during synthesis: