Name Solution *Formatted For 2-Sided Printing*

Staple This Side

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Friday, 30 October 2026, 11:30-12:20 CDT

Problem 1	_____	(10 pts)
Problem 2	_____	(15 pts)
Problem 3	_____	(10 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(15 pts)
Problem 6	_____	(15 pts)
Problem 7	_____	(15 pts)
Problem 8	_____	(10 pts)
Exam Total	_____	(100 pts)

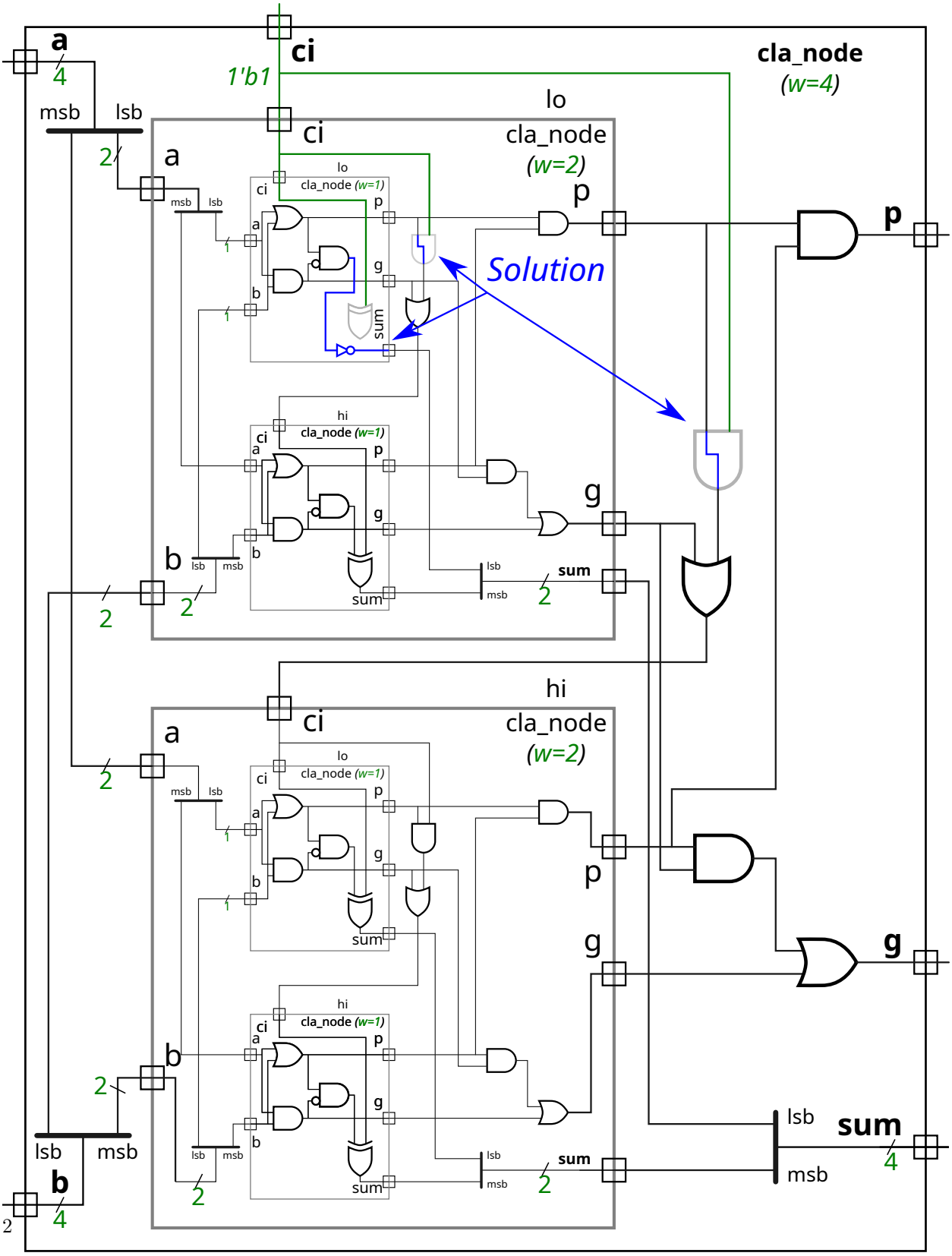
Staple This Side

Alias ZZWARNING_HUMANS_WONT_SURRENDER

Good Luck!

Problem 1: [10 pts] Appearing below is the hardware for a $w = 4$ CLA. Optimize the hardware for `cla_node.ci=1`.

☑ Optimize for `cla_node.ci=1`.



Staple This Side

Staple This Side

Problem 2: [15 pts] The module below is based on `left_justify_custom` from Homework 2 and 3.

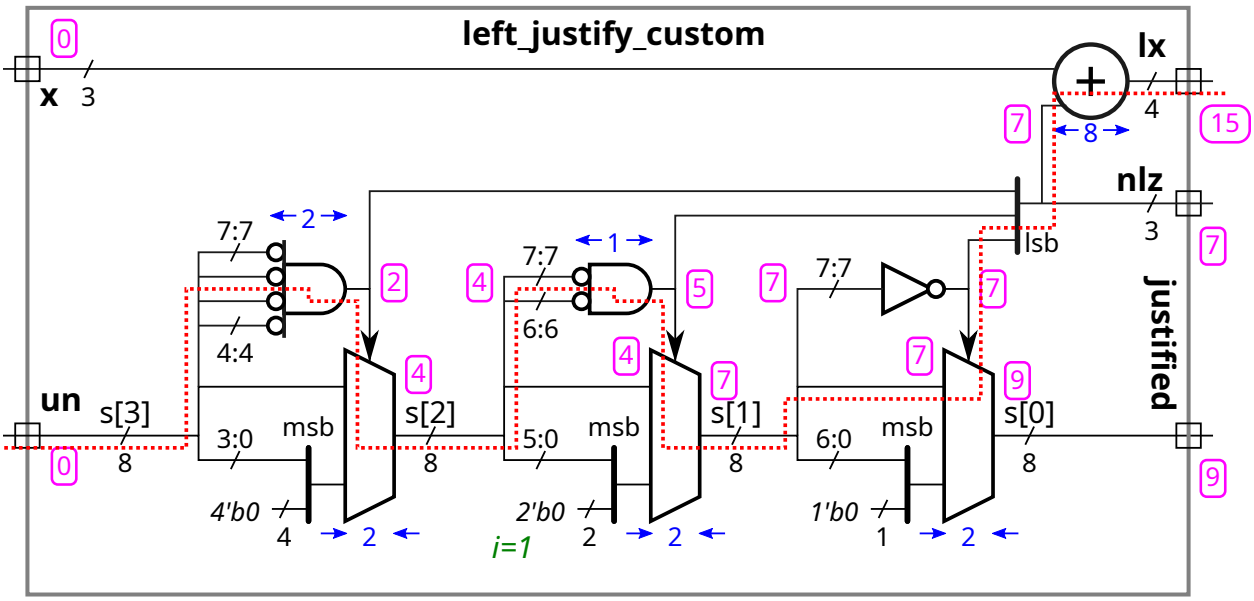
☑ Show ☑ the arrival time at each wire and show a ☑ critical path. ▷ The adder is a ripple adder and the high bit of its output is the carry out.

Solution appears below. A ripple adder delay is $2(w+1)\,\mathrm{u_t}$ for both the sum (MSB) and carry-out bits. Here we are using a 3-bit ripple adder, using the carry-out as the fourth bit. So we use $w = 3$, for a delay of $2(w+1)\,\mathrm{u_t} = 8\,\mathrm{u_t}$. The critical path is shown with a red dotted line.

A common mistake is to not treat multiplexor select inputs as inputs. They are inputs and in this case the critical path passes through them.

☑ Does accounting for constant inputs in the illustrated hardware affect the critical path? ☑ Explain.

No, because the critical path never passes through the multiplexor input bits having constant inputs (the less significant bits of the lower multiplexor inputs). For example, the critical path passes through the 2-input AND gate, which connects to bits 6 and 7 of `s[2]`. As can be seen by looking at the inputs to the first mux, bits 6 and 7 are non-constant for both the upper and lower inputs, and so the mux delay is 2. Note that for bits 0, 1, 2, and 3 the lower input is constant, so the mux delay for them would be 1 rather than 2. But alas those bits are not on the critical path.



Staple This Side

Staple This Side

Problem 3: [10 pts] Appearing below are CLA modules based on the ones used in class. Notice that module `cla` connects port `.cin` of the `cla_node` instantiation to the constant `1'b0`. We want the synthesis program to optimize for this constant, but it might not because it's from a port, not a parameter. This is especially true for the recursive instantiations. For that reason `cla_node` also has parameters that indicate whether the carry-in value should come from the parameter (so it will surely be optimized) or the port. If `cin_is_constant=0` then `cla_node` should use `cin` (the port), otherwise it should use `cin_cval` (the parameter).

☑ Complete `cla_node` so that it uses the appropriate carry-in value. ▷ Note that within the module `cinv` is used as the carry value. ☑ Don't forget about the recursive instantiations.

☑ Don't overdo it. In a correct solution only a few lines need to be modified.

```
module cla #( int w = 16, wb = 1 )
   ( output uwire [w-1:0] sum,  input uwire [w-1:0] a, b );
   // Do not modify this module.
   cla_node #(w,1,0) c(.sum(sum), .p(), .g(), .cin(1'b0), .a(a), .b(b) );
endmodule

module cla_node #( int w = 16, cin_is_constant = 0, cin_cval = 0 )
   ( output uwire [w-1:0] sum,   output uwire p, g,
     input uwire cin,            input uwire [w-1:0] a, b );

   // SOLUTION - Connect cinv to either the constant value (cin_cval) or to the port (cin).
   uwire cinv = cin_is_constant ? cin_cval : cin;

   if ( w == 1 ) begin
      assign g = a && b;
      assign p = a || b;
      assign sum = ( !g && p ) ^ cinv;
   end else begin

      localparam int wlo = w/2;
      localparam int whi = w - wlo;

      uwire plo, glo, phi, ghi;
      uwire chi = cinv && plo || glo;

      // SOLUTION
      //
      // Pass parameters indicating whether the carry-in is constant
      // to the lo instantiation (but *not* to the hi instantiation).
      cla_node_sol #( wlo, cin_is_constant, cin_cval )
         lo( sum[wlo-1:0], plo,glo, cinv, a[wlo-1:0],b[wlo-1:0] );

      cla_node_sol #(whi) hi(sum[w-1:wlo], phi,ghi, chi, a[w-1:wlo],b[w-1:wlo] );
      //
      // Note that the default value for cin_is_constant is zero, so
      // there is no need to specify it in the hi instantiation.

      assign p = plo && phi;
      assign g = glo && phi || ghi;
   end
endmodule
```

Staple This Side

Staple This Side

Problem 4: [10 pts] Input `f` to the module `tiger_ftoi` is an IEEE 754-like floating-point number using the specified exponent and fraction sizes, similar to Homework 2.

☑ Complete the module so that it sets output `i` to the *unsigned integer representation* the floating-point number on `f`.

☑ Round toward zero, so for $f < 1$ (including negative values) set $i = 0$.

☑ Solution can use procedural or structural code.

Solution appears below.

Staple This Side

```
module tiger_ftoi
  #( int w_i = 10, w_exp = 4, w_sig = 8, w_fp = w_exp + w_sig + 1 )
   ( output uwire [w_i-1:0] i,   input uwire [w_fp-1:0] f );

   localparam int bias = ( 1 << w_exp - 1 ) - 1;

   logic the_sign;
   logic [w_exp-1:0] exp;
   logic [w_sig-1:0] sig;
   always_comb { the_sign, exp, sig } = f;
   // The code above has already extracted the sign, exponent (biased),
   // and the significand.

   // SOLUTION
   //
   uwire [w_exp-1:0] uexp = exp - bias;   // Compute unbiased exponent.
   uwire [w_sig:0] frac = { 1'b1, sig };  // Append 1 to significand.
   //
   // Set i to zero if f is negative or less than 1 (exp<bias) ..
   // .. otherwise ..
   // .. shift right my w_sig to move the 1 we tacked on to the LSB ..
   // .. and then shift left (in effect) by the unbiased exponent.
   //
   assign i = the_sign || exp < bias ? 0 : frac >> ( w_sig - uexp );
   //
   // There is no problem if w_i > w_sig, zeros will automatically be
   // put in the vacated bits.

   // Example: Value = 110 (six),  w_sig = 6, w_i=4,  w_exp = 7
   //          sig =   100000   exp = 65   bias = 63
   //          uexp = 65-63 = 2
   //
   // frac:   { 1, 100000 } = { 1100000 }
   // i:     1100000 >> ( w_sig - uexp )
   //        1100000 >> ( 6 - 2 )
   //        1100000 >> 4
   //            110

endmodule
```

Staple This Side

Problem 5: [15 pts] The `clz_node` module is a variation of the tree `clz` presented in class.

☑ Show the inferred hardware for the module with $w > 2$ and w a power of 2.

☑ Show module ports. ☑ Use basic gates, multiplexors, arithmetic units.

☑ Do not confuse elaboration-time computation with computation hardware. ☑ **Do Not Use** `==` nor Verilog operators including concatenation expressions such as {a, b}.

```
module clz_node #( int w = 19, int ww = $clog2(w) )
   ( output uwire [ww-1:0] nlz,   input uwire [w-1:0] a );
   if ( w == 1 ) begin
      assign nlz = ww'(1'b0);
   end else if ( w == 2 ) begin
      assign nlz = !a[1];
   end else begin

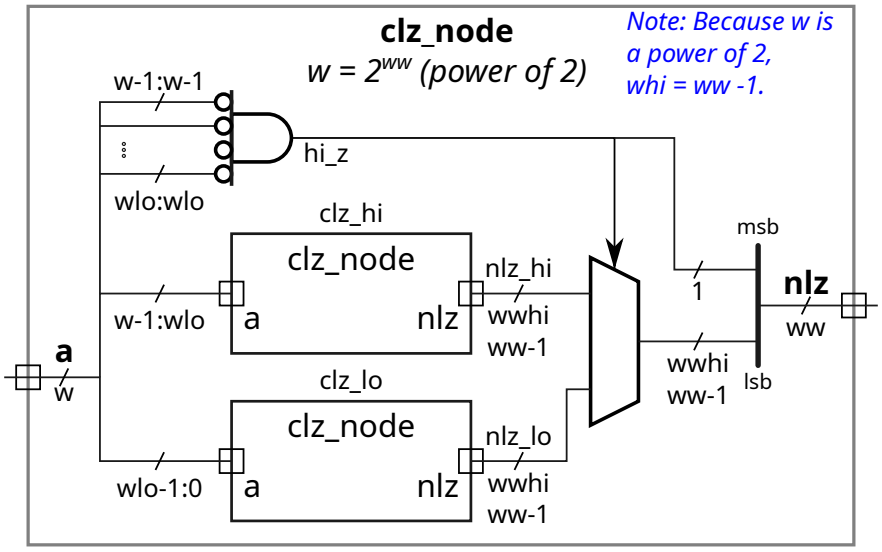
      localparam int lhi = $clog2(w) - 1;
      localparam int whi = 1 << lhi;
      localparam int wwhi = lhi;
      localparam int wlo = w - whi;

      uwire [wwhi-1:0] nlz_lo, nlz_hi;

      clz_node #(whi,wwhi) clz_hi( nlz_hi, a[w-1:wlo] );
      clz_node #(wlo,wwhi) clz_lo( nlz_lo, a[wlo-1:0] );

      uwire hi_z = a[w-1:wlo] == whi'(1'b0);
      assign nlz = { hi_z, wwhi'( hi_z ? nlz_lo : nlz_hi ) } ;
   end
endmodule
```

Solution appears below. Note that elaboration-time expressions such as `whi=1<<lhi` do not describe hardware operations (the left shift by `lhi`, for example). Instead, the result of that expression is used for the number of bits in certain wires. Interesting questions about the hardware are its cost and delay. Final exam problem?



Staple This Side

Staple This Side

Problem 6: [15 pts] Answer the questions below about multi-dimensional packed/unpacked arrays. ☑ Show work for partial credit. ☑ Show bit size in decimal, other values in binary or hexadecimal.

```
module loose_bits_sink_chips;
   logic [4:1][5:0] a [6:0][1:7];


   //  ☑  Number of bits in a:   4*6*7*7 = 1176


   //  ☑  Complete the declaration of g.
   //

   uwire        [5:1]    g    =  a[1][2][3];
   //           -----                          <--- SOLUTION underlined.

   //  ☑  Complete the declaration of h.
   //
   uwire        [4:1][5:0] h [1:7]   =  a[2];
   //           ----------   -----             <--- SOLUTION underlined.

   logic [2:0][8:1] cc;
   logic [3:1][8:1] cf;
   logic [3:0][5:0] d;
   logic [3:0][3:0] e, f;
   int unsigned x;

   initial begin
      cc = 'hfedcba;
      x = cc[1];
      //  ☑  Show value of x:   'h dc

      cf = 'hfedcba;
      x = cf[1];
      //  ☑  Show value of x:   'h ba

      d = cc;
      x = d[0];
      //  ☑  Show value of x:   'b 11 1010

      e = 'hf9ac;
      f = e + 8;
      //  ☑  Show value of f:   'h f9b4

      e[1] = e[1] + 9;
     //  ☑  Show value of e (not just e[1]):   16'h f93c

   end

endmodule
```

Staple This Side

Staple This Side

Problem 7: [15 pts] The modules in this problem compute a totient. *Note: The totient (rhymes with quotient) of n is the number of integers in $[1, n)$ that are relatively prime to n (don't divide n evenly).*

☑ Explain why the module below is not synthesizable.

It is not synthesizable because the number of iterations in the loop, `n/2`, is not constant. It's not constant because `n` is a module input port.

```
module totient #( int w = 3 )
   ( output logic [w-1:0] toti,   input uwire [w-1:0] n );
   always_comb begin
      toti = 0;
      for ( int i=2; i<n/2; i++ )
        if ( n / i * i != n ) toti = toti + 1;
   end
endmodule
```

☑ Fix the synthesizability issue using `nmax`. ▷ Don't worry about hardware cost.

```
module totient_syn #( int w = 10 )
   ( output logic [w-1:0] toti,    input uwire [w-1:0] n );
   localparam logic [w-1:0] nmax = ( (w+1)'(1'b1) << w ) - 1;

   always_comb begin

      toti = 0;

      for ( int i=2; i<nmax/2; i++ )   // SOLUTION: Use nmax instead of n.

        // SOLUTION: Put in ``i<n/2'' in if condition.
        if ( i < n/2  &&  n / i * i != n ) toti = toti + 1;

   end
endmodule
```

☑ Why is hardware cost a concern if `totient_syn` were made synthesizable with a small change?

Because there would be an enormous number of iterations. For example, for a reasonable value of $w = 32$ there would be $2^{32-1} = 2147483648$ iterations, and for each iteration a mux, divider, and adder are needed. That's way beyond what's possible as of this writing.

Staple This Side

Staple This Side

☑ What is the cost of the module below *after* optimization? ☑ Explain.

☑ Don't waste time on a lengthy answer.

```
module totient_plan_bb #( int w = 10, logic [w-1:0] n = 4 )
   ( output logic [w-1:0] toti );
   always_comb begin
      toti = 0;
      for ( int i=2; i<n/2; i++ )
        if ( n / i * i != n ) toti = toti + 1;
   end
endmodule
```

The cost is zero because the output, `toti`, only depends on constants and so the synthesis program can compute an exact value of `toti` given the instantiation parameters `w` and `n`.

Grading Note: Some observed that at the default value of `n` there would be zero iterations in the loop. That explains why the module at the default parameter values has zero cost but **it does not explain** what the cost of the module is with other instantiation parameters.

☑ Does module `totient_plan_b`, below, solve the synthesizability issue in `totient`? ☑ Explain.

```
module totient_plan_b #( int w = 10 )
   ( output logic [w-1:0] toti,  input uwire [w-1:0] n );
   totient_plan_bb #(w,n) tpb( toti );
endmodule
```

No, it does not fix the synthesizability issue, in fact it makes things worse. Module `totient` at least could be simulated, but `totient_plan_b` can't even be simulated because instantiation parameter `n` for instance `tpb` of `totient_plan_bb` is not an elaboration time constant.

Staple This Side

Staple This Side

Problem 8: [10 pts] Answer each question below.

(*a*) Appearing below are synthesis runs on variations of the Homework 2 left-justify operation.

Module Name	Area	Delay Actual	Delay Target	Synth Time
left_justify_procedural_w4	3995	0.44	3.0 ns	1 s
left_justify_parts_w4	3995	0.44	3.0 ns	1 s
left_justify_one_w4	3995	0.44	3.0 ns	1 s
left_justify_custom_w4	2762	0.40	3.0 ns	1 s
left_justify_procedural_w31	170006	2.94	3.0 ns	48 s
left_justify_parts_w31	86047	2.93	3.0 ns	14 s
left_justify_one_w31	68402	2.98	3.0 ns	11 s
left_justify_custom_w31	59156	3.00	3.0 ns	9 s

☑ In the table above why are three of the modules' delays identical at $w = 4$, but different from each other at $w = 31$?

Short Answer, sufficient for full credit: At $w = 4$ there are far fewer possible ways of applying transformations, and so the three modules were able to reach the same optimized place. At $w = 31$ there are many more ways to apply transformations, and so there was not enough time for the designs to reach the same point.

More Detailed Explanation: At both sizes the synthesis program will optimize the design to achieve the delay target at minimum cost. It optimizes by applying transformations, such as replacing $(ab)(cd)$ (in which there is a delay of 2 for all four inputs (a, b, c, and d) with $((ab)c)d$ in which d has a delay of 1 while a and b have a delay of 3. This would be a good transformation if d were on the critical path. Ideally all input modules that compute the same result, such as the four justify variations, will be optimized into the exact same design, a design that meets the delay target and has the minimum possible cost. Though that has not happened at $w = 4$, the first three modules seem to have been transformed into identical designs (based on area and delay, so it's not certain) at some point on their journey towards optimiality. Based on the table they are not optimal because the custom module has lower cost.

At $w = 32$ the number of ways that one can apply transformations is much larger and so it is much less likely that two very different designs (such as the ones above) would be transformed into the same design in a reasonable amount of time. So the different designs remain distinct.

☑ For the delay target of 3 ns and $w = 31$ which module should be chosen? ☑ Explain.

Module `left_justify_custom` would be chosen because it has the lowest cost. Though `left_justify_parts` has lower delay than 3 ns that lower delay probably does not provide any advantage to the design since the target was set at 3 ns not something lower.

Staple This Side

Staple This Side

(*b*) In an execution of the Verilog below inputs to `ripple3` are stable up until $t = 100$ when input `a` changes (but the other inputs hold steady). Note that there are three `bfa` instances, `ripple3.f0` `ripple3.f1`, and `ripple3.f2`. Call them `f0`, `f1`, and `f2` for short.

```
module bfa(output uwire c, s, input uwire a, b, cin);
   assign {c,s} = { a & b | a & cin | b & cin, a ^ b ^ cin };
endmodule

module ripple3 #( int w = 3 )
   ( output uwire [w-1:0] sum,   output uwire cout,
     input uwire [w-1:0] a, b,   input uwire cin );

   bfa f0( c0,   sum[0], a[0], b[0], cin );
   bfa f1( c1,   sum[1], a[1], b[1], c0 );
   bfa f2( cout, sum[2], a[2], b[2], c1 );

endmodule
```

☑ Show the order of execution of the `bfa` instances starting at $t = 100$ (which is when `a` changes).

☑ Note that an instance can execute more than one.

☑ In your answer show the points at which the inactive region is copied into the active region.

Start of answer: `f0`, `f1`,

The solution appears below in wholesome monospaced plain text. The entire execution shown below is at $t = 100$. The only way to advance time is with a delay statement, such as `initial begin a=1;b=3;#5;a=2;b=4;#6;a=3;b=8;end`. In that snippet $t = 0$ when `a=1`, $t = 5$ when $a = 2$, and $t = 13$ when `a=3`. But in the code fragments shown for this problem there are no delay statements so the event queue remains at $t = 100$.

Notice that `f1` executes twice and `f2` three times. An omniscient (and perhaps non-Verilog standard) scheduler might just execute `f0`, `f1`, `f2` just once since when `f1` executes the first time it would be using the `c0` written by `f0`. But in class for problems like this follow the Verilog scheduler rules and place processes into the inactive list when objects on their sensitivity lists change, even if the same process is in the active list.

```
SOLUTION
t=100
 Inactive: f0, f1, f2     Active:
 Inactive:                Active: f0, f1, f2    -> Execute f0, schedule f1 (c0)
 Inactive: f1             Active: f1, f2        -> Execute f1, schedule f2
 Inactive: f1, f2         Active: f2            -> Execute f2
 Inactive: f1, f2         Active:               -> Copy inactive to active
 Inactive:                Active: f1, f2        -> Execute f1, schedule f2
 Inactive: f2             Active: f2            -> Execute f2
 Inactive: f2             Active:               -> Copy inactive to active
 Inactive:                Active: f2            -> Execute f2
 Inactive:                Active:
```