Name Solution

Digital Design Using HDLs
LSU EE 4755
Midterm Examination

Wednesday, 23 October 2024, 11:30-12:20 CDT

Problem 1	_____	(22 pts)
Problem 2	_____	(18 pts)
Problem 3	_____	(20 pts)
Problem 4	_____	(10 pts)
Problem 5	_____	(30 pts)
Exam Total	_____	(100 pts)

Alias The best part is the last part.

Good Luck!

Problem 1: [22 pts] Below is the Homework 3 Problem 1 solution with some object names shortened.

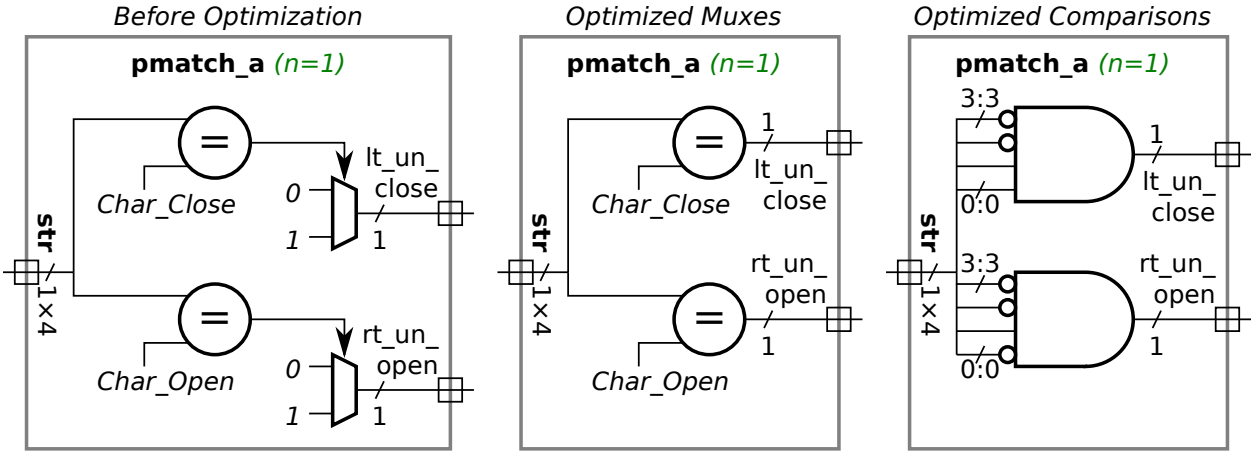
```
typedef enum logic [3:0] {Char_Blank=0, Char_Dot=1, Char_Open=2, Char_Close=3} Char;
module pmatch_a #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,   input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open =  str[0] == Char_Open  ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_a #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_a #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );

      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      assign lt_un_close = delta <  0 ? lt_close - delta : lt_close;
      assign rt_un_open =  delta >= 0 ? rt_open  + delta : rt_open;
   end
endmodule
```

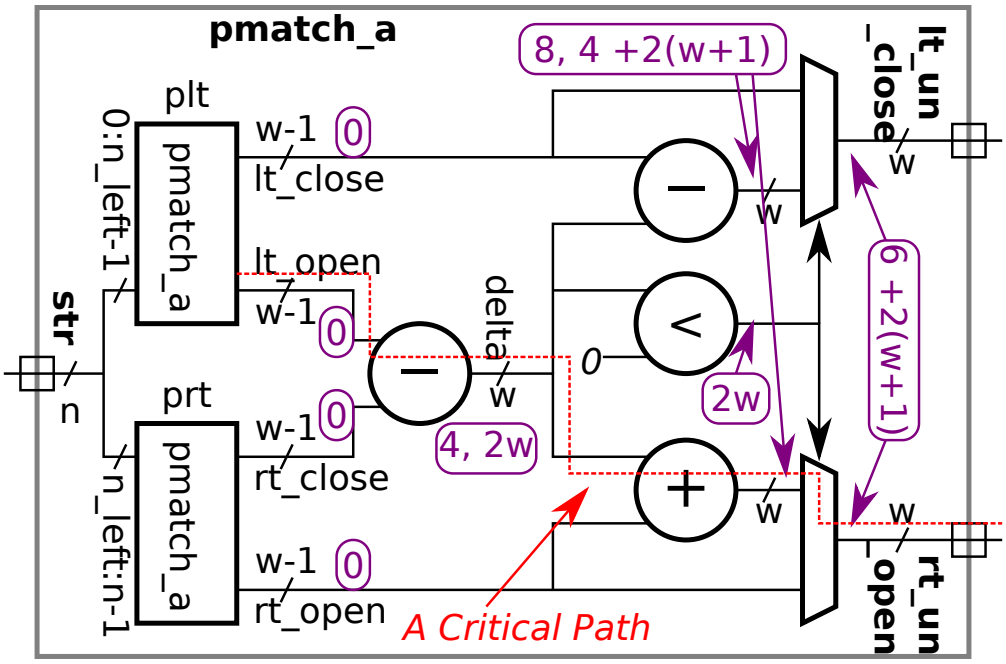
(*a*) Show the hardware that will be inferred for the base case. Show hardware after optimization taking into account constants.

☑ Show inferred hardware for base (`n==1`) case of the module above. ☑ Show input and output ports.

☑ Optimize taking into account ☑ constant values of all kinds. ☑ Don't miss the `Char` definition above the module. ☑ Don't show a comparison unit such as `==`, instead show the gates from which it was made and ☑ optimize them, ☑ taking into account the number of bits on each output port.

Solution appears below. The left-most version is without optimization, in the middle version the multiplexors have been optimized to wire, and in the rightmost version the comparison units have been optimized to AND gates.





Appearing above is hardware that will be inferred for the non-base case.

(*b*) Compute the cost of the hardware at this level (ignore what's inside `plt` and `prt`) based on the simple model using the bit widths from the diagram, such as `w-1`.

☑ Show the cost of each component *except for hardware inside of* `plt` *and* `prt`.

☑ Be sure to show the cost of the optimized comparison unit!

So far as computing cost here is concerned there are three types of components: the multiplexors, the comparison unit (in this case), and the adder and subtractors.

Multiplexors: The cost of a w-bit mux is $3w\,\mathrm{u_c}$. There are two multiplexors, so the total multiplexor cost is $6w\,\mathrm{u_c}$.

Comparison: The comparison unit checks if `delta` is negative. To do that just check if the most-significant bit (sort of a sign bit) is `1`. So, the comparison cost is zero.

Adder and subtractors: The cost of an x-bit ripple adder is $9x\,\mathrm{u_c}$. Note that an x-bit ripple unit has x-bit inputs and, when one includes the carry-out, computes an $(x+1)$-bit result. The unit that computes `delta` is $w-1$ bits, and lest this get too tedious, treat the other subtractor and adder as having w-bit inputs. So, the ripple units' combined cost is $9(w-1+2w)\,\mathrm{u_c} = [27w-9]\,\mathrm{u_c}$.

(*c*) Compute the delay through the module **starting from** launch points `lt_close`, `lt_open`, `rt_close`, and `rt_open`. The capture points are `lt_un_close` and `rt_un_open`. Use the bit widths from the diagram, such as `w-1`.

☑ Show the arrival time at each wire from launch to capture.

☑ Take into account cascaded ripple units and ☑ and the optimized comparison unit.

The timing appears on the diagram above. Also shown, though not asked for, is the critical path (in red). For arrival time at the outputs of the ripple units two times are shown: the time of the least-significant bit and the time of the most-significant bit. For example, the LSB of `delta` is arrives at $4\,\mathrm{u_t}$ and the MSB arrives at $2w\,\mathrm{u_t}$. For clarity the unit, $\mathrm{u_t}$, is omitted from the diagram. The comparison unit just passes the MSB of `delta` (it does not add any additional delay).

Problem 2: [18 pts] Appearing below is an alternative solution to Homework 3 Problem 1. The only difference is the last few lines.

```
module pmatch_b #( int n = 5, wn = $clog2(n+1) )
   ( output logic [wn-1:0] lt_un_close, rt_un_open,
     input uwire [3:0] str[0:n-1] );
   if ( n == 1 ) begin
      assign lt_un_close = str[0] == Char_Close ? 1 : 0;
      assign rt_un_open = str[0] == Char_Open ? 1 : 0;
   end else begin
      localparam int n_left = n/2;
      localparam int n_right = n - n_left;
      localparam int wl = $clog2(n_left+1),  wr = $clog2(n_right+1);
      uwire [wl-1:0] lt_close, lt_open;
      uwire [wr-1:0] rt_close, rt_open;
      pmatch_b #(n_left,  wl) plt( lt_close, lt_open, str[0:n_left-1] );
      pmatch_b #(n_right, wr) prt( rt_close, rt_open, str[n_left:n-1] );
      uwire logic signed [wn-1:0] delta = lt_open - rt_close;
      // Lines above are identical to pmatch_a.

      uwire [wn-1:0] delta_n = delta <  0 ? delta : 0;
      uwire [wn-1:0] delta_p = delta >= 0 ? delta : 0;
      assign lt_un_close = lt_close - delta_n;
      assign rt_un_open  = rt_open  + delta_p;
   end
endmodule
```

(*a*) Show the hardware that will be inferred for `pmatch_b`. For your convenience the hardware for `pmatch_a` is shown in the upper right. *Note: In the original exam the condition for* `delta_n` *was* `delta <= 0` *and the condition for* `delta_p` *was* `delta > 0`. *Though the hardware computed the correct result, the comparison would have been more expensive since it would have had to check for a zero condition, not just negative.*

☑ Show inferred hardware on the facing page.

Solution appears on the facing page. One major difference is that the addition and subtraction are done *after* the multiplexors. Also notice that each multiplexor has a constant input, zero. Finally, notice that the output of the comparison unit for `delta<0` can be used for the line needing `delta>=0` since one or the other is true.

(*b*) Compute the simple-model cost of the hardware.

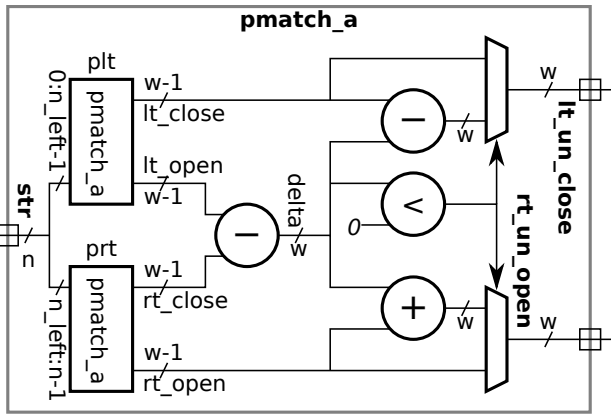
☑ Write *same* next to components that cost the same as corresponding components in `pmatch_a` and ☑ compute the cost of other components ☑ after optimization.

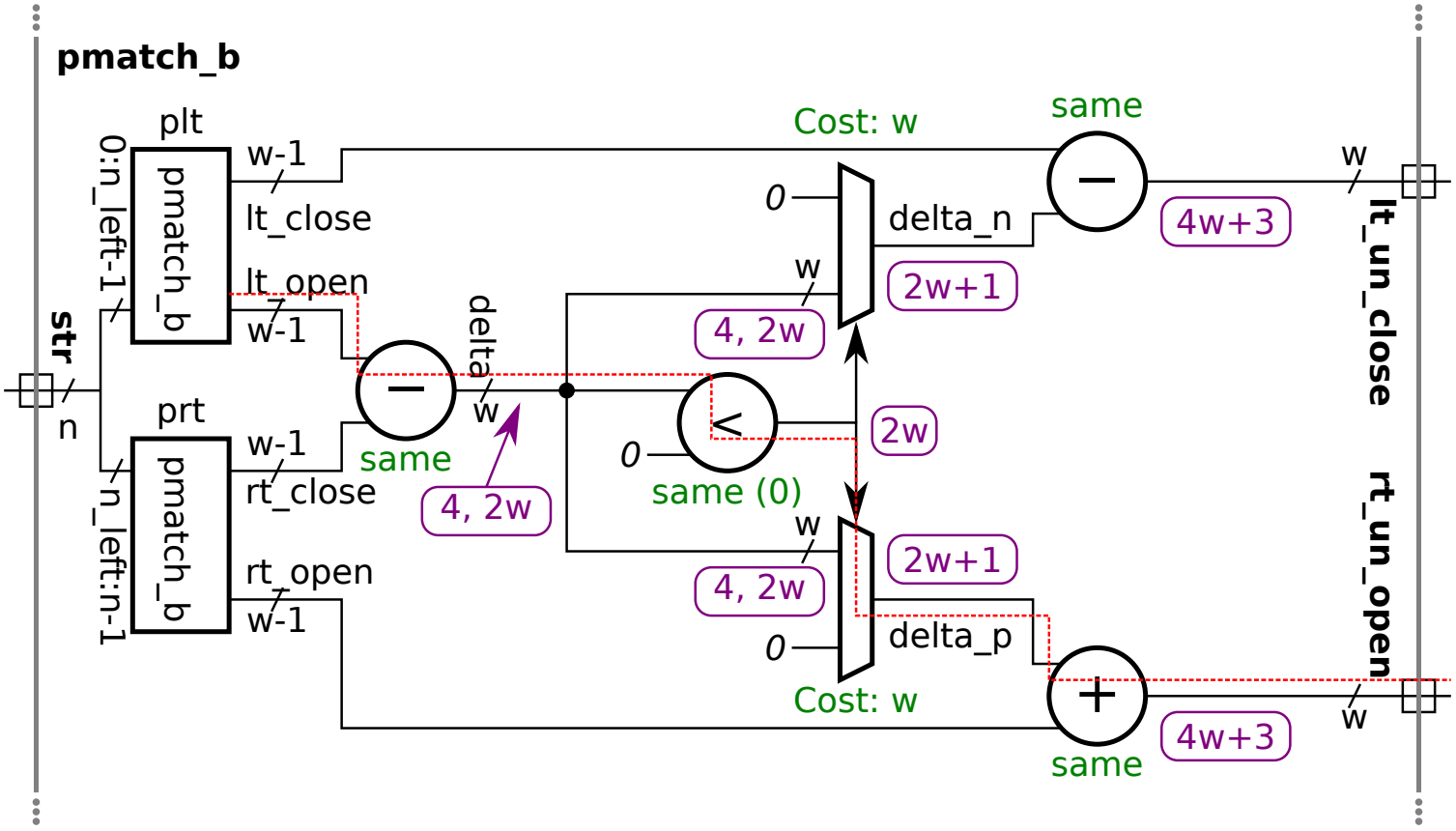
The costs are shown in the diagram in green. The only significant difference is the multiplexors. Because each has a constant input the cost of each is now $w\,\mathrm{u_c}$ (the multiplexors in `pmatch_a` cost $3w\,\mathrm{u_c}$ each).

(*c*) Compare the critical path lengths.

☑ Will the critical path in `pmatch_a` be much different than the one in `pmatch_b`? ☑ Explain.

Yes, the critical paths will be very different. In the diagram the arrival times are circled and the critical path is shown as a red dashed line. Because there is a multiplexor with a late-arriving select signal between the initial subtraction and the subsequent add or subtract, the ripple units are no longer cascaded. That means the least significant bit does not arrive at the adder and subtractor until $[2w+1]\,\mathrm{u_t}$ (due to the multiplexor select signal not stabilizing until $2w\,\mathrm{u_t}$). As a result `pmatch_b` takes nearly twice as long.


pmatch_a
plt
pmatch_a
0:n_left-1
w-1
lt_close
lt_open
w-1
str
n
prt
pmatch_a
n_left:n-1
w-1
rt_close
w-1
rt_open
delta
w
0
w
lt_un_close
rt_un_open
w


pmatch_b
plt
pmatch_b
0:n_left-1
w-1
lt_close
lt_open
w-1
str
n
prt
pmatch_b
n_left:n-1
w-1
rt_close
rt_open
w-1
same
delta
w
4, 2w
Cost: w
same
0
delta_n
2w+1
w
4, 2w
4w+3
lt_un_close
w
0
same (0)
2w
w
4, 2w
2w+1
0
delta_p
Cost: w
same
4w+3
rt_un_open
w

Problem 3: [20 pts] Appearing below are some of the `dot` modules from the solution to Homework 1. On the facing page is incomplete module `dotn`. Complete `dotn` so that it describes hardware that computes the dot product of `n`-element vectors recursively, where `n` is the parameter. That is, `dotn` must instantiate `dotn` and should instantiate `mult` and `add` where needed.

```
module mult #( int w = 5 ) ( output uwire [w-1:0] p, input uwire [w-1:0] a, b );
   assign p = a * b;
endmodule

module add #( int w = 5 ) ( output uwire [w-1:0] s, input uwire [w-1:0] a, b );
   assign s = a + b;
endmodule

module dot2 #( int w = 5 )
   ( output uwire [w-1:0] dp,    input uwire [w-1:0] a[1:0], b[1:0] );
   // Computes dp = a[0] * b[0] + a[1] * b[1];
   uwire [w-1:0] p0, p1;
   mult #(w) m0(p0, a[0], b[0] );
   mult #(w) m1(p1, a[1], b[1] );
   add #(w) ad(dp, p0, p1 );
endmodule

module dot3 #( int w = 5 )
   ( output uwire [w-1:0] dp,   input uwire [w-1:0] a[2:0], b[2:0] );
   // Computes   dp = a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
   uwire [w-1:0] p0, p2;
   dot2 #(w) d0( p0, a[1:0], b[1:0] );
   mult #(w) m2( p2, a[2], b[2] );
   add #(w) a2(dp, p0, p2 );
endmodule
```

☑ Complete `dotn` so that it describes tree-structured hardware computing an `n`-element dot product. The tree depth should be $\lceil \lg n \rceil$.

☑ Instantiate `mult` for multiplication and `add` for addition, and of course `dotn` for a dot product of a smaller vector.

☑ To keep things easy all wires are `w` bits.

The solution appears below. Note that the multiplication is performed in the base case and that the adds are done in the recursive instances.

```
module dotn
  #( int w = 5, n = 4 )
   ( output uwire [w-1:0] dp,
     input uwire [w-1:0] a[n-1:0], b[n-1:0] );

   // SOLUTION

   if ( n == 1 ) begin

      // Base Case: Just multiply.
      //
      mult #(w) m( dp, a[0], b[0] );

   end else begin

      // Recursive Case: Split inputs between recursive instances ..
      //
      localparam int nlo = n/2;
      localparam int nhi = n - nlo;
      uwire [w-1:0] dplo, dphi;
      dotn #(w,nlo) dlo( dplo, a[nlo-1:0], b[nlo-1:0] );
      dotn #(w,nhi) dhi( dphi, a[n-1:nlo], b[n-1:nlo] );
      //
      // .. and add their outputs ..
      //
      add #(w) a( dp, dplo, dphi );

   end

endmodule
```

Problem 4: [10 pts] Appearing below is the logarithmic shifter presented in class, followed by a version that's supposed to be better (but isn't). The hoped-for improvement is due to instantiating the exact number of multiplexors (`muxw2`) needed, rather than enough for the maximum shift amount.

```
module shift_right_logarithmic #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,   input uwire [lgw-1:0] amt );
   // This module is correct.
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;
   for ( genvar i=0; i<lgw; i++ )
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lgw-1];
endmodule

module shift_right_logarithmic_better_maybe #( int w = 16, lgw = $clog2(w) )
   ( output uwire [w-1:0] shifted,
     input uwire [w-1:0] un,  input uwire [lgw-1:0] amt );
   uwire [w-1:0] s[lgw:-1];
   assign s[-1] = un;

   // Use exactly the number of stages needed!!!
   uwire [lgw-1:0] lg_amt;              // LINE ADDED
   my_clog2 #(lgw) mc( lg_amt, amt );  // LINE ADDED. Set lg_amt = $clog2(amt) = ⌈lg amt⌉;

   for ( genvar i=0; i<lg_amt; i++ )   // LINE DIFFERS
     muxw2 #(w) st( s[i],  amt[i],  s[i-1],  s[i-1] >> ( 1 << i )  );
   assign shifted = s[lg_amt-1];       // LINE DIFFERS
endmodule
```

☑ Why won't the Verilog above compile?

The Verilog won't compile because the `for` loop is a generate loop and its stop condition, `i<lg_amt`, is not an elaboration-time constant expression due to `lg_amt` being a module output (of the `mc` instance of the `my_clog2` module). The loop would have to be a generate loop because it is in module scope and because the iterator is declared `genvar`.

☑ Is it possible to fix the Verilog error in such a way that cost is lower with smaller shift amounts? ☑ Explain.

No. The cost is determined by the amount of hardware needed to synthesize the module. Once the module is synthesized, manufactured, and shipped to a customer its cost (meaning the amount of hardware) can't change. This isn't Hogwarts, we're muggles (at least I am).

☑ Is it possible to fix the Verilog error in such a way that the delay reported by a synthesis program is lower? ☑ Explain.

No, because the delay reported by the synthesis program will be based on the critical path, which is the longest path.

☑ Is it possible to fix the Verilog error in such a way that the delay actually is lower? ☑ Explain.

If you must have a shorter delay for a lower shift amount that is possible but (1) it won't be much of an improvement below galactic sizes and (2) the logic outside the module will have to be designed to take advantage of the lower delay with sorter shift amounts. So, though the answer to this question is yes, as a practical matter the answer is still no.

Problem 5: [30 pts] Answer the following Verilog questions.

(*a*) The module below uses multidimensional arrays.

```
//                          2       1                       3    4       1    2
module mda( input uwire [2:1] c [5:1],    input uwire [7:1][2:1] a [5:1][3:1] );
   //   ☑ Add dimension(s) to the declaration of e so that the assignment is correct.
   //
   //         SOLUTION
   uwire      [2:1] e [5:1]      = c;


   //   ☑ Add dimension(s) to the declaration of b so that the assignment is correct.
   //
   //         SOLUTION
   uwire      [2:1] b            = a[1][1][1];


   logic g [7:0];
   logic [7:0] h;

   initial begin
//   ☑ Which is correct,
      ◯ only the assignment to g,   ◯ only the assignment to h, or   ⊗ both are correct.
     ☑ Explain.

      g[1] = h[1];
      h[1] = g[1];
   end

endmodule
```

☑ What is the size of `c`, in bits? ☑ What is the size of `a`, in bits?

The size of object `c` is $2 \times 5 = 10$ bits. The size of object `a` is $7 \times 2 \times 5 \times 3 = 210$ bits.

Explanation for multiple-choice question about `g` and `h`. It is true that `g` is an unpacked array and `h` is a packed array and the two kinds of array work differently. But in this case both `g[1]` and `h[1]` refer to 1-bit quantities, and so assigning one to the other is no problem. (Possible final-exam question: ask about `h[1:0]` and `g[1:0]`.)

(*b*) In the module below indicate whether each code fragment is correct.

```
module kinds #( logic [31:0] pg = 123 )
   ( output uwire [31:0] o0,  input uwire [31:0] ik );

   //  ☑ Is the line below correct?      ⊗ Yes   ◯ No   ☐ If not, explain.

   localparam logic [31:0] z02 = pg + 4755;



   //  ☑ Is the line below correct?      ◯ Yes   ⊗ No   ☑ If not, explain.

   localparam logic [31:0] z03 = ik;
```

A localparam can only be assigned an elaboration-time constant expression (including literals and elaboration-time constants). Since **ik** is a module input it cannot be an elaboration-time constant.

```
   //  ☑ Are the lines below correct?    ◯ Yes   ⊗ No   ☑ If not, explain.

   localparam logic [31:0] z04;
   assign z04 = pg;
```

Though **pg** is an elaboration time constant, it must be assigned to **z04** in the declaration statement (the statement starting starting with **localparam**). Put another way every **localparam** declaration must include the constant value.

```
   //  ☑ Are the lines below correct?    ◯ Yes   ⊗ No   ☑ If not, explain.

   uwire [31:0] z10 = pg;
   assign z10 = ik;
```

Object **z10** is declared **uwire**. A **uwire** must have exactly one driver. Both lines above are drivers, which is one too many. To keep the **assign** line one would need to remove **=pg** from the **uwire** declaration (but keep the rest).

```
   //  ☑ Is the line below correct?      ⊗ Yes   ◯ No   ☐ If not, explain.

   uwire [31:0] z13 = ik;



endmodule
```

(*c*) When we run a synthesis program we specify a delay target. In class we often synthesize twice, once with a delay target of 100 ns and a second time with a target of 0.1 ns. What is the harm in specifying a delay target lower (faster) than one needs? Isn't faster better?

☑ Harm in setting delay target too low is:

Short answer: The harm is synthesized hardware that's more costly than it would have been with the correct delay.

The delay target should be set to the delay that's needed by a design. For example, the design team may have decided that they would like a 2.5 GHz clock frequency and so they would set the delay target to $\frac{1}{2.5\,\text{GHz}} = 400\,\text{ps}$. The synthesis program will optimize delay until the 400 ps target is met, and then optimize cost. Typically the lower the delay, the more the hardware will cost.

In Scenario A the team needs 400 ps but the person running the synthesis program specified a target of 200 ps and didn't tell anyone. When the more-costly-than-they-need-to-be manufactured components are used in a product they are clocked at 400 ps, wasting their potential. The competition might have a less expensive product that runs at the same speed.

In Scenario B the team needs 400 ps, the person running the synthesis program specifies a target of 200 ps, and told the team. So the design is run at 200 ps. Though the hardware runs faster, it's of no benefit because the toaster makes great toast with either component. There's no point in sampling the image sensors any faster to 20 Hz to make good toast.

(*d*) A 32-bit signed integer, say i, is converted into a 32-bit IEEE 754 floating-point format (8-bit exponent, 23-bit significand) and then back into a 32-bit integer, j.

☑ Is it guaranteed that $i = j$ for all $-2^{31} \le i < 2^{31}$? ☑ Explain based on the FP representation.

No. The significand is only 23 bits, meaning the fraction is 24 bits (counting the implicit 1). Consider two integers $253{,}969{,}770_{10} = f23{,}456a_{16}$ and $253{,}969{,}771_{10} = f23{,}456b_{16}$. Each is seven hexadecimal digits and so would need at least 28 bits to represent. To represent these in the FP format the significand would be set to the most significant 24 bits, $f2{,}3456_{16}$ (including the implicit 1), omitting the least-significant hex digit a or b. (The exponent would be set to 154 for both numbers.) Since the two numbers have identical FP representations converting them back to integers will yield the same number, $f23{,}4570_{16}$.

Appearing below is the complete FP representation of $253{,}969{,}770_{10} = f23{,}456a_{16}$ and $f23{,}456b_{16} = 253{,}969{,}771_{10}$ in 32-bit (single, which is the format described in the problem) and 64-bit (double, a higher-precision format) formats. Notice that the fractions (F) of the numbers' single representations are the same, but the fractions of their double representations are different.

```
Value 253969770.0000000000000000 == 2.539698e+08
  Single:  0x4d723457           1299330135
           S 0   E 0x9a = 154     F 0x723457
  Double:  0x41ae468ad4000000  4732797820489170944
           S 0   E 0x41a = 1050   F 0xe468ad4000000

Value 253969771.0000000000000000 == 2.539698e+08
  Single:  0x4d723457           1299330135
           S 0   E 0x9a = 154     F 0x723457
  Double:  0x41ae468ad6000000  4732797820522725376
           S 0   E 0x41a = 1050   F 0xe468ad6000000
```