Name Solution

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)

Alias Vwl Shrtg

☣ $V(\text{mRNA}) \quad \Rightarrow \quad R_e < 1$

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.

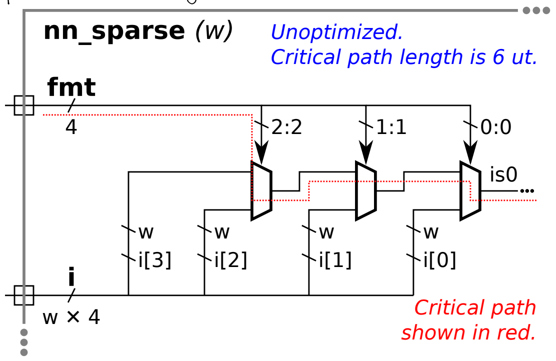
```
   uwire [w-1:0] is0 = fmt[0] ? i[0] : fmt[1] ? i[1] : fmt[2] ? i[2] : i[3];
```

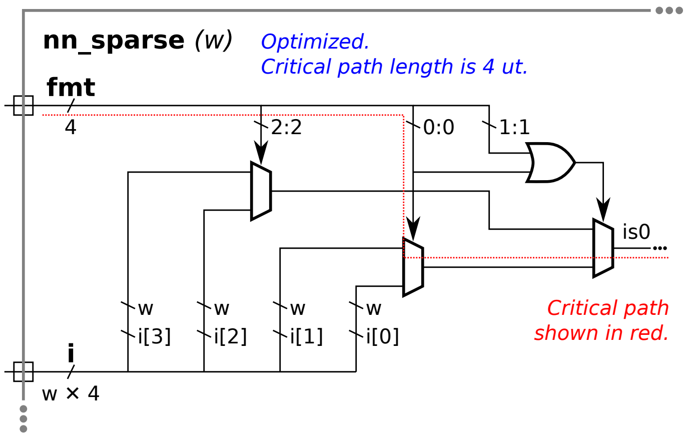
☑ Show inferred hardware.

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

☑ Use only basic logic gates and multiplexors.

Solution appears below. The unoptimized hardware follows the rules for inference of the conditional operator (`?:`). In the optimized version the critical path is reduced by two units by rearranging the three multiplexors into a reduction tree and using an OR gate to provide a control signal for the mux at the root.





(*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:

Each multiplexor (optimized or not) cost $3w\,\mathrm{u_c}$ and the OR gate cost $1\,\mathrm{u_c}$. The total cost for the unoptimized version is $9w\,\mathrm{u_c}$ and the total cost for the optimized version is $[9w+1]\,\mathrm{u_c}$.

☑ In terms of w delay is:

The delay through a 2-input multiplexor is $2\,\mathrm{u_t}$. In the unoptimized version the critical path passes through three multiplexors, for a delay of $6\,\mathrm{u_t}$. In the optimized version the critical path passes through just 2 muxen, for a delay of $4\,\mathrm{u_t}$.

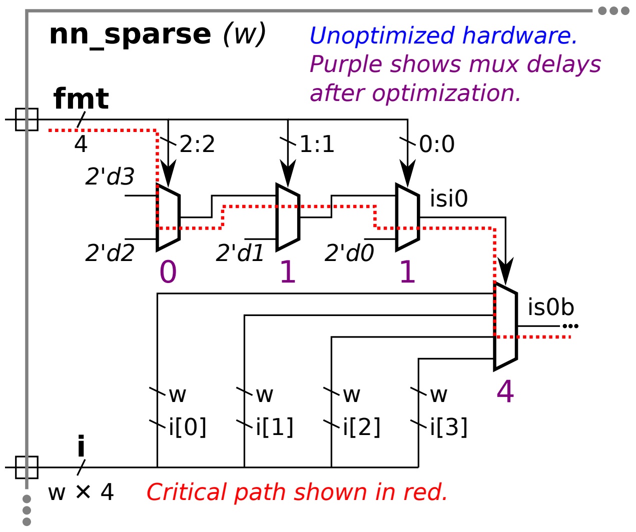
Note that the delay is not a function of w. Be sure that you thoroughly understand why this is true.

(*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.

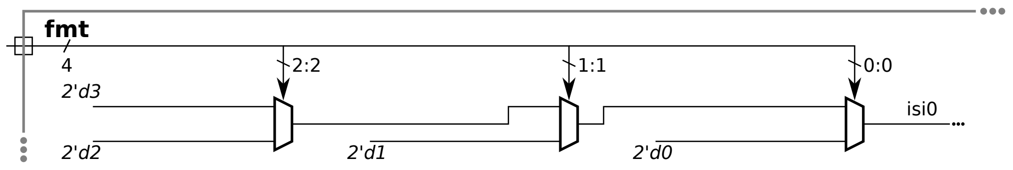
The inferred hardware appears below. The logic computing `isi0` is similar to the logic computing `is0` in the previous part, except that its inputs are constants rather than elements of `i`. The inferred logic for `is0` here is a four-input multiplexor.

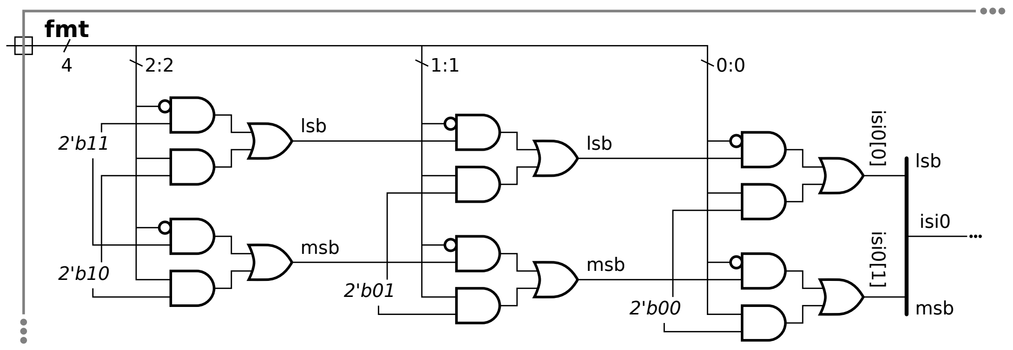


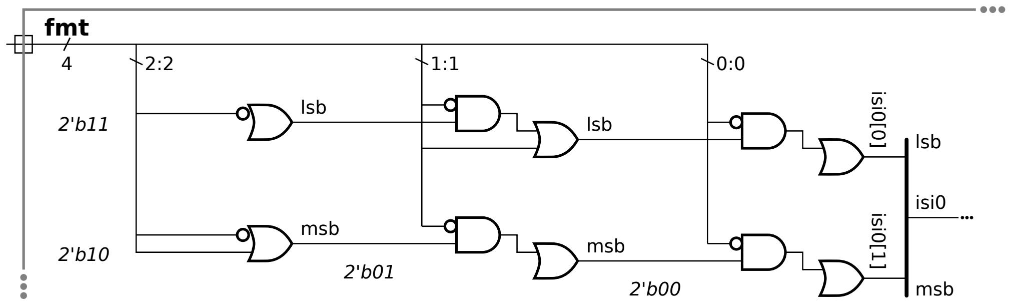
[✓] Show optimized hardware, optimize to reduce delay.

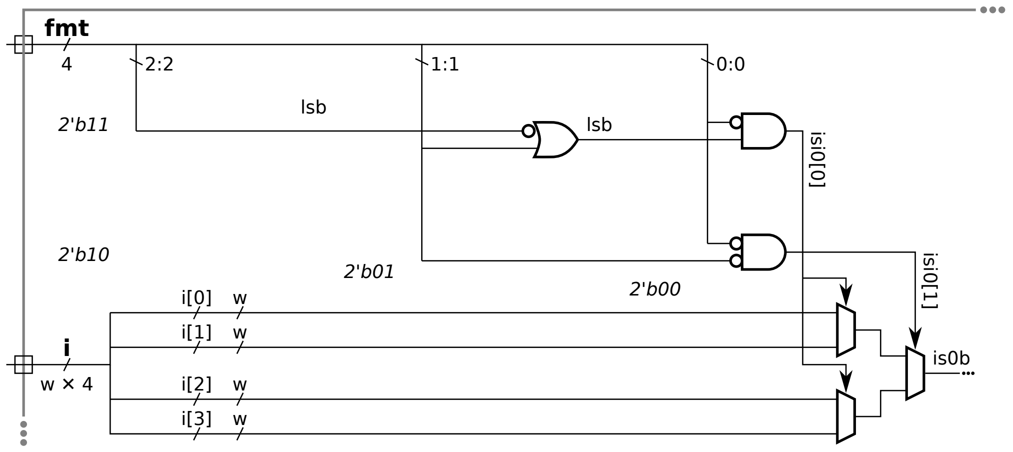
[✓] Use basic logic gates and [✓] no muxen for `isi0` and [✓] two-input muxen (plus other logic) for `is0b`.

The optimized logic computing `isi0` appears below after several steps in the optimization process. At the last step the logic for `is0b` is also shown, but that logic is not fully optimized. The optimization shown below is based on the Verilog code above. A synthesis program that has not been provided with the limit on the values of `fmt` could do no better. (With knowledge that exactly one bit of `fmt` will be `1` a synthesis program (or human) would optimize the two lines above into the logic given for the solution to part (a) of this problem.)









(*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.)

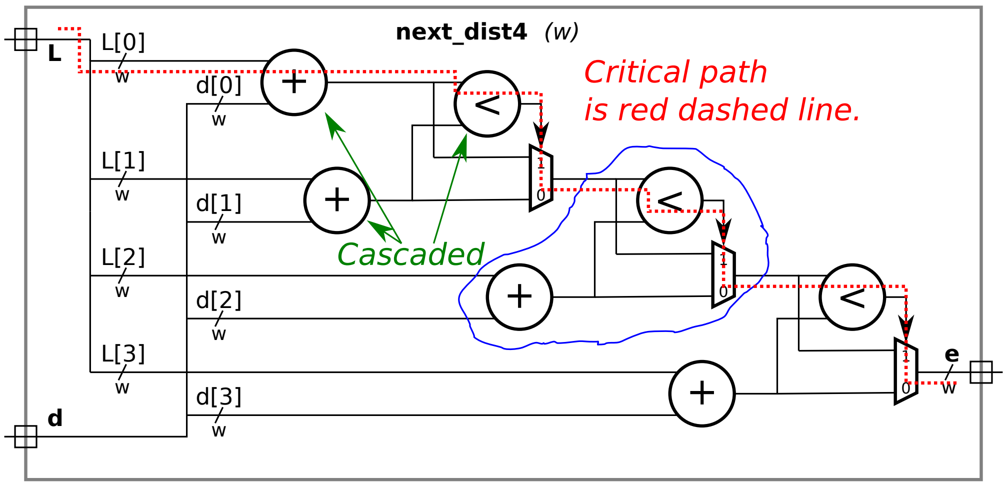
☑ In terms of w cost is:

The cost of the logic in the last section of the illustration above the cost is $[3+3\times 3w]\,\mathrm{u_c}$. The cost would drop to $[1+3\times 3w]\,\mathrm{u_c}$ if the select signals to the first two multiplexors were connected as shown in the optimized solution to part (a).

☑ In terms of w delay is:

The delay of the hardware in the last section of the illustration above is $[1+1+2+2]\,\mathrm{u_t}$, with the critical path passing through the logic generating `isi0[0]`. One cycle can be saved by switching the positions of `isi0[0]` and `isi0[1]` and correspondingly rearrange the order of the `i` inputs to the first two multiplexors to `i[0], i[2], i[1], i[3]`. That would reduce the critical path by 1.

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:

The module consists for four adders, three comparison units and 3 multiplexors. Each of these devices operate on w bits. Based on the slides describing the simple model, the cost of a w-bit ripple adder is $9w\,\mathrm{u_c}$, the cost of a w-bit comparison unit is $4w\,\mathrm{u_c}$, and the cost of a w-bit 2-input multiplexor is $3w$. The total cost is $\boxed{[4\times 9w+3\times 4w+3\times 3w]\,\mathrm{u_c}=57w\,\mathrm{u_c}}$.

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

☑ Account for any cascading ripple units.

The critical path appears on the illustration as a red dashed line.

The start of the path passes through an adder and a comparison unit. In isolation the delay of an adder is $2(w+1)\,\mathrm{u_t}$ and the delay of a comparison unit is slightly less, $[2w+1]\,\mathrm{u_t}$ according to the simple model slides. But because the output of the adder (actually two adders) connects to the comparison unit the cascaded delay can be used, which is $[4+2(w+1)]\,\mathrm{u_t}=[2w+6]\,\mathrm{u_t}$. Because of the multiplexors cascading delays cannot be used for the other two comparison units. That is because their upper inputs don't arrive until the mux select signal stabilizes. So the remaining delay is that of three 2-input muxen and two w-bit comparison units: $[3\times 2+2\times(2w+1)]\,\mathrm{u_t}=[4w+8]\,\mathrm{u_t}$. The total delay is $\boxed{[2w+6+4w+8]\,\mathrm{u_t}=[6w+14]\,\mathrm{u_t}}$.

(*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 ami #( int w = 22 )                         /// SOLUTION
   ( output uwire [w-1:0] s_out,
     input uwire [w-1:0] L, d, s_in );

   // Compute sum ..
   //
   uwire [w-1:0] sum = L + d;
   //
   // .. and connect it to s_out if it's smaller than input value, s_in.
   //
   assign s_out = sum < s_in ? sum : s_in;
endmodule

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

   // Compute first sum. This does not need a comparison, so don't use ami.
   //
   uwire [w-1:0] e0 = L[0] + d[0];

   // Interconnections between ami instances.
   //
   uwire [w-1:0] e1, e2;

   // Instantiate three ami modules and interconnect them properly.
   //
   ami #(w) a1( e1, L[1], d[1], e0 );
   ami #(w) a2( e2, L[2], d[2], e1 );
   ami #(w) a3( e,  L[3], d[3], e2 );
endmodule
```

(*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                          /// SOLUTION
  #( int n = 20, w = 12 )
   ( output uwire [w-1:0] e,
     input uwire [w-1:0] L[n], d[n] );

   localparam logic [w-1:0] mv = ~w'(0);

   uwire [w-1:0] ee[n-1:-1];
   assign ee[-1] = mv;
   assign e = ee[n-1];

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

      ami #(w) a( ee[i], L[i], d[i], ee[i-1] );

   end

endmodule
```

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;
                       //        Sensit.  Execution Time
                       //         List    And Scheduled Lines
   assign g = f | c;   // Line 1   f,c    x           x           x
   assign f = a * c;   // Line 2   a,c     x ->(L1)     x ->(L1)
   assign a = b + c;   // Line 3   b,c      x ->(L2)
                       //                 -----       -----       ---
                       //                 Active      Active      Active
                       //                 List        List        List
endmodule
```

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

☑ `with_assign` could be confusing because:

The dataflow order (order of dependencies) is from bottom to top but humans expect to read these things from top to bottom. (Line 2 depends on Line 3, Line 1 depends on Line 2.) That would be annoying.

(*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? *The following sentence was not in the original exam:* Use sensitivity lists to justify your answer.

☑ About how many times does each line execute? ☑ Explain with sensitivity lists.

See the work in the comments above for this discussion. At $t = 10$ because `b` and `c` change Lines 1-3 are all put first in the inactive list, then in the active list for execution. As a result of their execution Line 1 and Line 2 are placed in the inactive list. That becomes the active list when the first one shown is empty. The execution of Lines 2 causes Line 1 to be scheduled a third time. So in total, Line 1 executes 3 times, Line 2 executes twice and Line 3 once.

(*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 logic [w-1:0] g, input uwire [w-1:0] b, c );

   logic [w-1:0] a, f; // SOLUTION: Change to logic. Also g.

   always_comb begin

      // SOLUTION: Put lines in dataflow order.
      a = b + c;  // Line 3
      f = a * c;  // Line 2
      g = f | c;  // Line 1

   end
endmodule
```

☑ Why does `sans_assign` make less work for the simulator than `with_assign`? Explain using sensitivity lists.

The sensitivity list of the `always_comb` consists of just `b` and `c`. Objects `a`, `f`, and `g` are not in the sensitivity list (because their values when `begin` is reached are not used). The `always_comb` block will only be scheduled for execution when `b` or `c` changes. So for the $t = 10$ scenario the block—and each line—is executed just once.

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.

The critical path of `mult` goes through just a multiplier. The critical path of `mult_1a` goes through a multiplier and a multiplexor, and so the critical path is longer. The fact that the output is available sooner for the two special cases does not change the critical path.

✓ 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`.

In `m2` and `ma2` the `a` input is a constant. The synthesis program will then be able to determine, for `mult_1a`, which part of the `if/else` chain executes and synthesize only for that. If `a=3` then it will be the `a*b` part, and so the two modules are identical. In both `m2` and `ma2` the synthesis program can see that the `a` input is a constant and will optimize the multiplier appropriately. That means if `a=1` `ma2` will have no advantage.

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:*]

The assignment statements, such as `p=0;`, in the module below are an error in a module context.

```
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
```

One solution is to switch to a continuous assignment (an **`assign`** statement), that has been done below.

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

   if ( a == 0 )          assign p = 0;      // SOLUTION: Use assign.
   else if ( a == 1 )     assign p = b;
   else                   assign p = a * b;

endmodule
```

☑ Module is ◯ *correct* or [✓] *has the following error and fix:*

A procedural assign is being used on a net kind (**uwire** in this case). So, unlike **mult_2a**, the kind of assignment statement is correct here since the assignment occurs in procedural code. The problem is kind of object being assigned.

```
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
```

A simple fix is to change `p` to a var. Note that `uwire` is short for `uwire logic` and that `logic` is short for `var logic`.

```
module mult_2b #( int w = 32, logic [w-1:0] a = 12 )
   ( output logic [w-1:0] p, input uwire [w-1:0] b );
   // SOLUTION: Change p from "uwire logic" to "var logic".

   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:]

The `if` in the code below is a generate `if`, and its condition, `b==0`, is not an elaboration-time constant. (The expression `b==0` is not an elaboration-time constant because `b` is a module input.)

```
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
```

A fix is to make the code procedural by wrapping it in an always block and making `p` a var.

Note: Changing `b` to `a` in the `if` condition is NOT an appropriate fix because it changes what the module does.

```
module mult_2c #( int w = 32, logic [w-1:0] a = 12 )
   ( output logic [w-1:0] p, input uwire [w-1:0] b );
   always_comb   /// SOLUTION: Change generate if to procedural if.
    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
```

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

The solution appears below. The difference between `a`, `b`, and `c` are in how the bits are numbered. That only impacts the use of the indexing (bit select) operator, `[i]`. It does not affect assignments and other references to the objects. For that reason `b` and `c` on the first assignment are the same as `a`. However, the second assignment of `b` refers to the bits, so they are reversed.

```
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:

      // SOLUTION:
      //  b = 16'h1234
      //  c = 16'h1234


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

      // SOLUTION
      //  b = 16'h2c48
      //    = 16'b_0010_1100_0100_1000
      //
      //  Note that:
      //  a = 16'b_0001_0010_0011_0100

   end
endmodule
```

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.

In technology mapping generic gates (say, a 3-input AND gate) are replaced by gates in the target technology. The replacement can also happen at a higher level, so a generic adder module would be replaced by an adder in the target technology, if such a thing is provided.

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

The optimization would be on generic gates, which might be available at any size. The target technology might have gates with, say, 2, 4, or 6 inputs, but not 3 or 5 inputs. So another round of optimization might find a way to use those wasted inputs. Also, after technology mapping the delay of gates are known, and so delay optimization can occur.

(*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:

With a very large delay target the optimization program can minimize cost. As the delay is lowered the optimization will have to substitute higher-cost alternatives to meet the delay target. (For example, substituting a carry lookahead adder for a ripple adder.) Making the delay smaller than it needs to be can result in costs higher than they need to be.