

Ⓞ Paper copies will not be accepted. E-mail your solution to koppel@ece.lsu.edu. A single PDF file is preferred.

Problem 1: In the Module-Port-versus-Module-Parameter section of lecture code <https://www.ece.lsu.edu/koppel/v/2020/1005-review.v.html> there are several module designs for computing $c_1x + c_2y$, where c_1 and c_2 are constants and x and y are module inputs. The point of that section and of the modules was to illustrate the SystemVerilog differences between module parameters and ports (syntax issues, for example) and also how they relate to the hardware being modeled.

(a) Draw a diagram of module `c1x_c2y_good`, shown below, using its default parameter values (which are different than the ones in the lecture code). Show the contents of all instantiated modules and appropriately label ports and wires. (See 2016 Homework 1 Problem 3 for a diagram showing instantiated modules. Also see module `arb_exp` and the illustration that follows in <https://www.ece.lsu.edu/koppel/v/2020/1015-syn-comb-str.v.html>.)

- Use the default parameter values of the module `c1x_c2y_good` shown below.
- Use the appropriate parameter values for the `mult_by_c` instances. *Hint: appropriate is not a synonym for default.*
- Show the ports for all modules.
- Show the number of bits in each wire.
- Label wires with the symbols used below (such as `p1` and `prod`) and take care to place the label on the correct side of a module boundary. (In the `two_pie` illustration from <https://www.ece.lsu.edu/koppel/v/2020/1005-review.v.html> look at the wire carrying labels `x`, `i1`, and `a`.)

Continued on next page.

```

module mult_by_c
  #( int w = 8, int c = 16, int w2 = w+$clog2(c) )
  ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a );
  assign prod = a * c;
endmodule

module c1x_c2y_good
  #( int c1 = 4, int c2 = 7, int w = 15,
    int w2 = w + $clog2(c1) + $clog2(c2) )
  ( output logic signed [w2-1:0] s, input uwire signed [w-1:0] x, y );

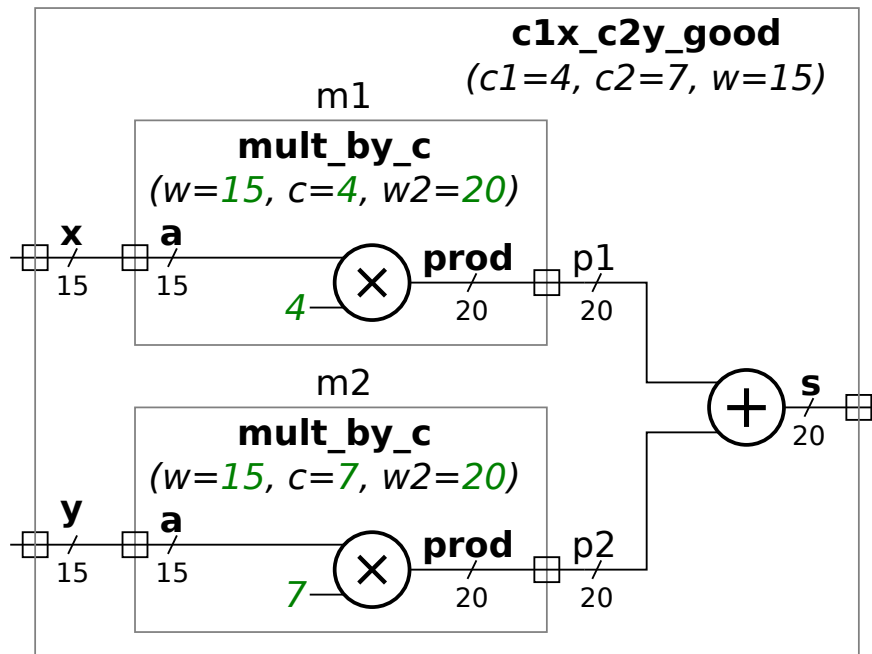
  uwire [w2-1:0] p1, p2;

  mult_by_c #(w,c1,w2) m1(p1,x);
  mult_by_c #(w,c2,w2) m2(p2,y);

  assign s = p1 + p2;
endmodule

```

Solution appears below. Notice that parameters are not shown as module inputs. For example, *c1* is not shown as an input to *m1*.



(b) Draw a diagram of module `c1x_c2y_okay` below using its default parameter values (which are different than the defaults used in the lecture code). Show the same details, such as ports, as was requested for the previous part.

```

module mult
  #( int w = 8, int w2 = 2 * w )
  ( output uwire signed [w2-1:0] prod, input uwire signed [w-1:0] a, b );
  assign prod = a * b;
endmodule

```

```

module c1x_c2y_okay
  #( int c1 = 4, int c2 = 7, int w = 15,
    int w2 = w + $clog2(c1) + $clog2(c2) )
  ( output logic signed [w2-1:0] s, input uwire signed [w-1:0] x, y );

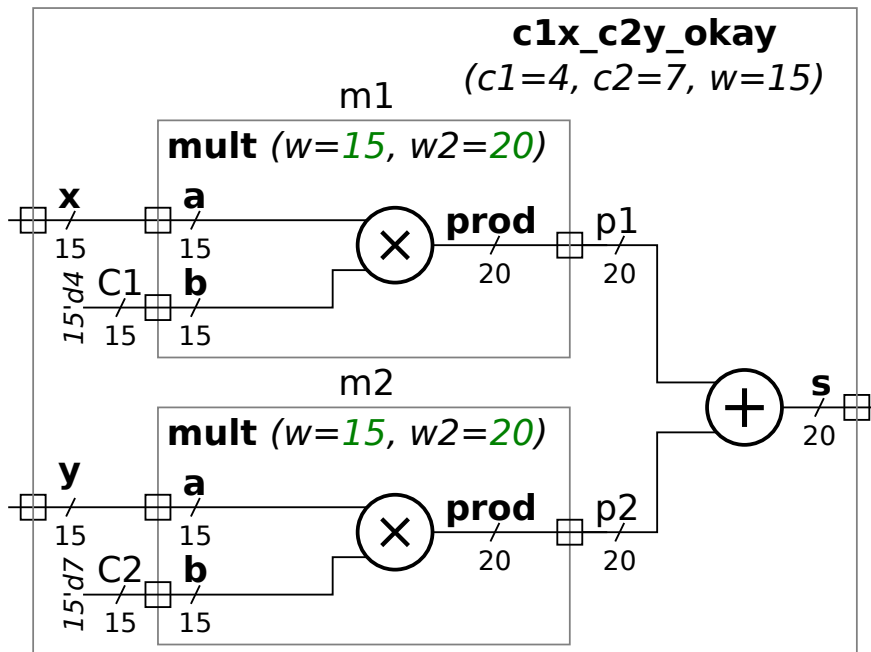
  uwire [w2-1:0] p1, p2;
  uwire [w:1] C1 = c1, C2 = c2; // Convert constants to desired size.

  mult #(w,w2) m1(p1, x, C1);
  mult #(w,w2) m2(p2, y, C2);

  assign s = p1 + p2;
endmodule

```

Solution appears below. Here, `C1` and `C2` are inputs to `m1` and `m2`. A lazy synthesis program, or less judgmentally, a synthesis program set to optimize at a low effort level might not take advantage of the fact that in `m1` the `b` input is 4. That would result in much more expensive hardware.

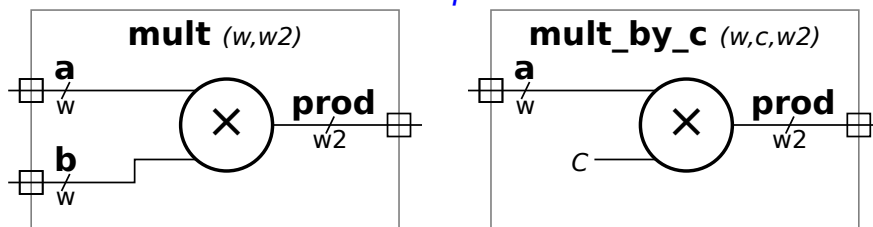


Problem 2: Synthesis programs optimize a design to minimize cost while meeting timing constraints. The illustration below for the `mult` and `mult_by_c` modules (used in the slides) show how the multiplier can be simplified when one of the inputs is a convenient constant, 1.

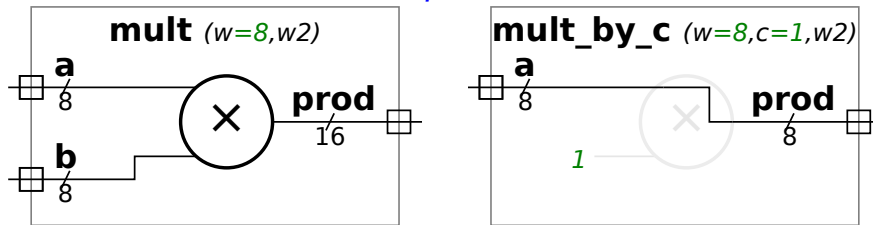
Show how the `c1x_c2y_good` module from the first problem can be optimized based on the default `c1=4` and `c2=7` values. To do so show the multiplier replaced by much simpler hardware, such as adder(s). A correct solution uses only one adder for both multipliers, *plus the adder used to combine p1 and p2*.

Note: As originally assigned, and until Tuesday, 15 September 2020 at about 16:15, the problem stated that a correct solution uses only one adder, implying but not specifically stating that the one adder was the replacement for the multipliers and that there would also be an adder computing $p1+p2$, for a total of two adders.

Before instantiation and optimization.



After instantiation and optimization.



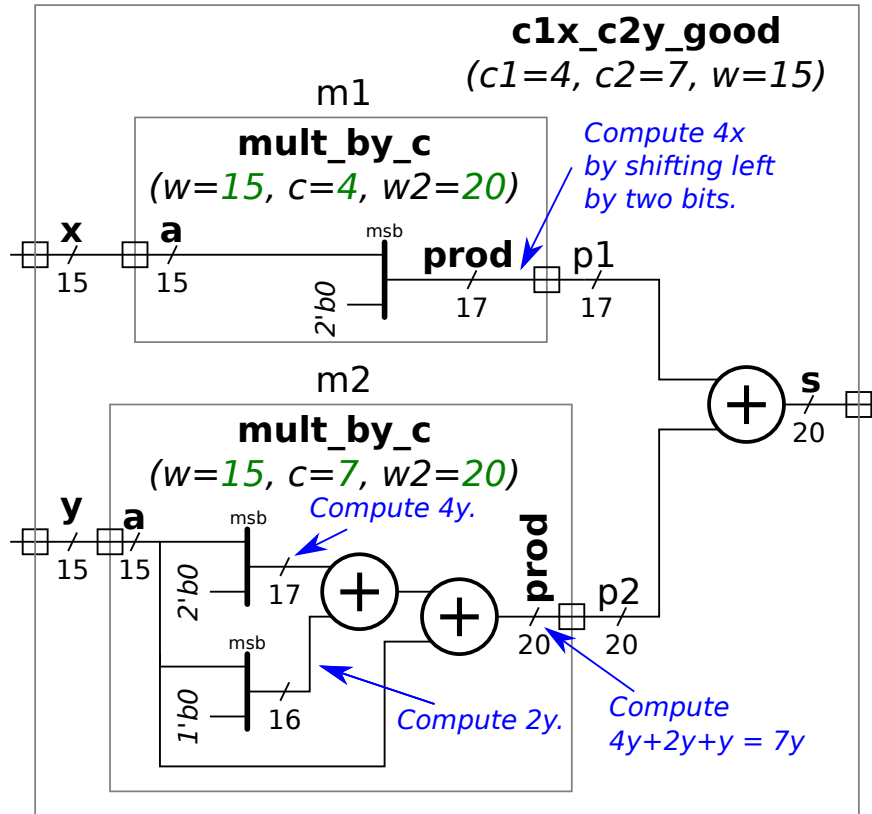
Solution on next page.

Two solutions appear below. The first is easier to understand, but uses two adders for **m2**. The second uses one adder for **m2**.

Both solutions take advantage of the fact that multiplication by a power of 2, such as 4, can be achieved by left-shifting. To compute $4x$ the value of x is left-shifted by two positions. The hardware for achieving that is trivial: relabel bit position i to $i + 2$ and set bits at positions 0 and 1 to the constant 0. Both solutions do this in **m1**. Make sure that the notation for re-labeling bits used in **m1** is understood.

The solution below computes $7y$ using two adders: $4y + 2y + y = 7y$.

Both solutions use adders that have unequal port sizes. For example in the first solution the adder computing **s** has one 17-bit input and one 20-bit input. That's not an unreasonable assumption to make.



Better one-adder-m2 solution on next page.

The solution using one adder for $m2$ appears below. Recall that $m2$ computes $7y$. That can be done with one adder by computing $8y + (-y) = 7y$. But to compute a 2's complement representation of $-y$ one needs to negate each bit and then add 1. Negating each bit is easy. A wasteful solution would use an adder just to compute $(-y - 1) + 1$. There's no need for that here, instead the solution computes $(8y + 1) + (-y - 1) = 7y$. The quantity $8y + 1$ is obtained by left-shifting by 3 bits and then putting a 1 in the least-significant bit position. Negating the bits of 2's complement number y results in $-y - 1$, which is what we need. Notice that the hardware computing $-y - 1$ produces an 18-bit quantity by sign-extending the 15-bit quantity. The need to do sign extension in the diagram below could have been eliminated by using an adder with an 18- and 15-bit input. The adder would do the sign-extension internally.

