Problem 1: Consider the pair of logic functions (yes, they are from Homework 5).

$$f_0 = \left(a + bc + \overline{b}cd\right)\overline{e}$$

$$f_1 = (a + bc + b\overline{c}d)\,\overline{e}$$

(a) Write a Verilog structural description of a single module implementing these functions. Take advantage of the common terms in the two functions $(a + bc)$.

The solution appears below. Notice that the bc term is shared by the two outputs (the same signal is used for generate `sop0` and `sop1`). The ne value is also shared. It would also have been possible to compute $a + bc$ and share that result too.

```
// SOLUTION:
module hw6p1a(f1,f0,a,b,c,d,e);
   input a, b, c, d, e;
   output f1, f0;

   wire   nb, nc, ne, bc, nbcd, bncd, sop0, sop1;

   not n1(nb,b);
   not n2(nc,c);
   not n3(ne,e);

   and a1(bc,b,c);
   and a2(nbcd,nb,c,d);
   and a3(bncd,b,nc,d);

   or o1(sop0,a,bc,nbcd);
   or o2(sop1,a,bc,bncd);

   and a4(f0,sop0,ne);
   and a5(f1,sop1,ne);

endmodule
```

(b) Notice that one function contains a $\overline{b}cd$ term and the other contains a $b\overline{c}d$ term. Though the variables are different the two terms represent the same operation (a three-input AND with one input to the AND inverted). Write a Verilog structural description for these functions taking advantage of this fact by having two modules. One module will be the three-input AND gate just mentioned. The other will compute the functions using two instances of the AND gate.

Solution appears below. That 3-input and gate is called `minterm011` (because the first input is inverted). Notice that it is instantiated twice in module `hw6p1b` and that the input order is different in the two instances.

Use of this `minterm011` module avoids the need for two inverters in `hw6p1b`. The benefit is that `hw6p1b` is easier for humans to read (assuming they can quickly grasp what `minterm011` does). There is no difference in the circuit being described, so simulation and synthesis of the code in this problem and the previous one should be identical.

```
module minterm011(f,i2,i1,i0)
   input i2, i1, i0;
   output f;

   wire   ni2;

   not n1(ni2,i2);
   and a1(f,ni2,i1,i0);

endmodule

module hw6p1b(f1,f0,a,b,c,d,e);
   input a, b, c, d, e;
   output f1, f0;

   wire   ne, bc, nbcd, bncd, sop0, sop1;

   not n3(ne,e);

   and a1(bc,b,c);

   minterm011 m1(nbcd,b,c,d);
   minterm011 m2(bncd,c,b,d);

   or o1(sop0,a,bc,nbcd);
   or o2(sop1,a,bc,bncd);

   and a4(f0,sop0,ne);
   and a5(f1,sop1,ne);

endmodule
```

Problem 2: Answer the following questions about EDA (CAD).

(*a*) How does a synthesis program know the capabilities of the chip it is targeting, such as the number of gates it can hold?

It reads a *technology kit*, a set of files provided by the technology vendor (chip fabrication facility, FPGA maker, etc).

(*b*) Comment on the correctness the statement below:

"Before writing a design in Verilog one should simplify the combinational logic (perhaps using Karnaugh maps) so that the resulting chip will have lower cost."

The synthesis program will perform simplification. In most cases it will do better than a human, at least for combinational logic (the only kind covered in EE 2720).

(*c*) Why must a timing simulation be performed after synthesis? Why can't the simulator read some files that contain gate propagation delays and use that to perform a timing simulation before synthesis is ever done?

The synthesis program will simplify the logic and so the simulation program does not even know the exact circuit to compute timing information for.

The speed of a gate is determined by the number of ports it's output is connected to (the fan out) and how far away those ports are. The distance is not known until the placement step is performed.

Note: The synthesis program might create a new Verilog (or some other HDL) file with the simplified logic and containing timing information. That can be read by the simulator to perform a timing simulation.

(*d*) The same Verilog file for a design (or VHDL) is read by simulation programs (for functional simulation) and synthesis programs, meaning that Verilog must be good for both simulation and synthesis. Why not have separate languages for simulation and synthesis?

The point of functional simulation is to verify that your design works correctly. If a different language is used to specify a design for synthesis, then there could be differences such that the one used for functional simulation is correct but the one used for synthesis is wrong. There is also the waste of time in writing something twice.

Problem 3: A 2^n-input multiplexer can be build using 2^n $n+1$-input AND gates and an 2^n-input OR gate. Each input to this mux is just one bit. In practical applications one might want to switch larger inputs, for example, each input might be an 8-bit quantity. Explain why the cost of an 8-bit 2^n-input mux is less than 8 times the cost of a 1-bit 2^n input mux. Draw a logic diagram to illustrate your answer.

Consider a mux implemented using a decoder, such as the one appearing in Figure 6.19 of the text. The logic implemented in the figure handles just one bit per data input. To make the data inputs two bits one need only duplicate the illustrated AND and OR gate, there would still be just one decoder. So the cost would be less than twice as much (assuming a two-input AND gate costs less than an AND gate with more inputs).

Problem 4: *Note: Don't attempt this problem until you are comfortable with all other material in this course.* Design logic to determine if one 16-bit unsigned integer is greater than another one.

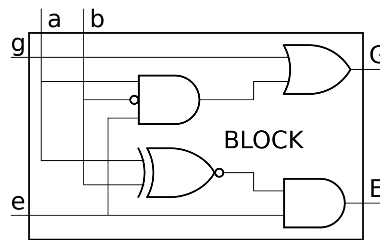
(a) Do this by designing a block with four inputs, a, b, e, g. There will be 16 blocks. Inputs a and b in block i connect to a_i and b_i, the bit at position i in a and b, respectively. Input e in block i is true if bits i up to 15 in a and b are equal, it is false otherwise. (That is, if $a_j = b_j$ for $i < j < 16$.) Input g in block i is true if $a > b$ looking at bits $> i$. (That is, there exists some k such that $a_j = b_j$ for $k < j < 16$ and $a_k = 1$ and $b_k = 0$.) It is false otherwise.

The block will have two outputs, E and G. As the alert student has already guessed, output E of block i connects to input e of block $i - 1$, similarly for G. For block 15 (corresponding to the most significant digit) input e_{15} is 1 and g_{15} can be set to any value (0 if that's less confusing) but a 0 results in a simpler circuit.

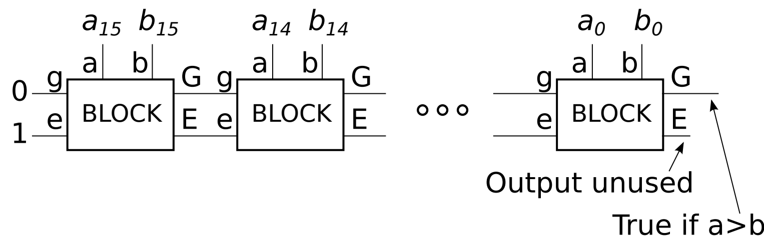
Design the block, and draw a diagram showing how they are connected. Also show how the overall output is generated, a one-bit signal that is true if $a > b$.

The diagram immediately below is one block of the circuit. It connects to a bit each of a and b, as described above, and receives the g (greater-than) and e (equal-to) signals produced by a similar block connected to a more significant digit. It in turn produces G and E signals for the next block, connected to a less significant digit.

The output E signal is true if the input e signal is true and if a and b are equal. (Note that an exclusive-nor gate's output is 1 if its two inputs are equal, and 0 otherwise.) The output G signal is true if the input g signal is true or if the e signal is true (meaning that all of the more significant bits of a and b are equal) and if a is 1 and b is 0.



The diagram below shows each block connected to a different bit position in a and b. To understand how the circuit works start at the most significant bit, a_{15} and b_{15}. If $a_{15} = 1$ and $b_{15} = 0$ (which means that $a > b$) then the G output of the leftmost block will be 1 and that signal will propagate through the OR gates in the other blocks until reaching the output on the right. If $a_{15} = 0$ and $b_{15} = 1$ (meaning that $a < b$) then the E output will be 0, and that 0 value will propagate through the AND gates to all the blocks, preventing any G output from reaching 1. If $a_{15} = b_{15}$ (meaning that we can't tell whether $a > b$ by just looking at bit position 15) then the E output will be 1 and the G output will be 0, and so the e and g inputs to the next block (bit position 14) are the same as for position 15, meaning that we can apply the argument above to position 14, and by induction to the bit positions down to 0.

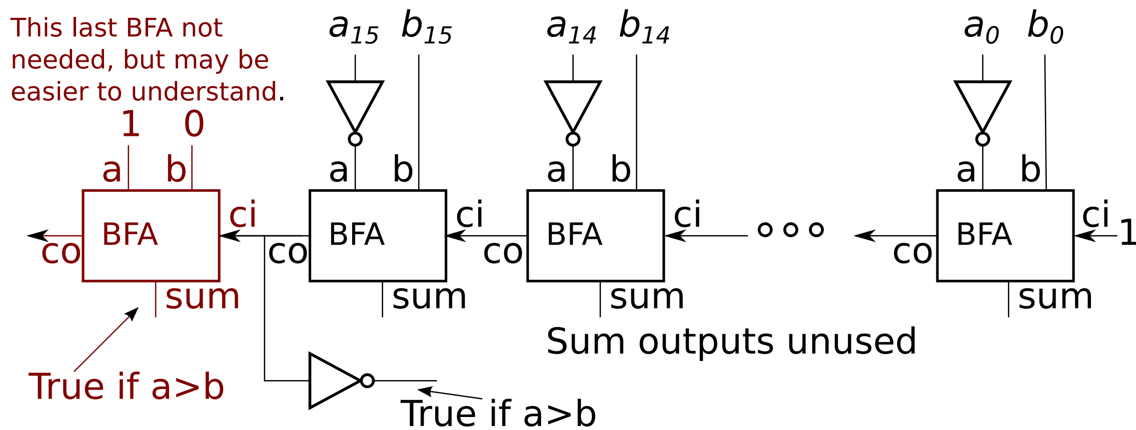


(b) Design the $a > b$ circuit using the binary full adder (BFA) as a building block. Negate b using inverters and by taking advantage of the unused carry-in. This design should consist of a bunch of

BFAs (just show a box for them) and inverters. Don't forget to show the one-bit $a > b$ signal. One might need to review 2's complement arithmetic before completing the problem.

The idea is to compute $b - a$ and check if the difference is negative. To compute $b - a$ use a ripple adder to compute the sum $b + (-a)$ (that is, $-a$ is input to the ripple adder). Recall that to compute the n-bit 2's complement representation of $-a$ given a in binary, we negate each bit position and then add 1. For example, for $a = 5$ in an 8-bit representation: $5 = 101_2 = 0000\,0101 \xrightarrow{\text{invert bits}} 1111\,1010 \xrightarrow{\text{add 1}} 1111\,1011 = -5$. In the circuit, a is inverted by using a NOT gate on each bit. To add 1 we will set the carry-in input of the ripple adder to 1.

If the sum produced by the ripple adder is negative then $a > b$, that can be tested for by looking at the sign bit of the sum. All of the other bits of the sum are ignored (and so we eliminate the logic producing the sum in all but one BFA). Using exactly this approach would require 17 BFAs, the 17'th one is used only to detect the sign bit. A less costly approach is to look at the carry out at bit position 15. If that is 1 then $a \leq b$ if it is 0 then $a > b$. Both approaches are used in the diagram below. The last BFA on the left, in red is for the extra 17'th bit. The output is based on the sum (the sign bit). The less costly approach is just looks at the carry-out of the BFA at bit position 15.



An interesting thing to notice that in the first solution (part a) signals propagate from the most to the least significant bit. In the solution with the adder signals propagate from the least to the most significant bit.