Verilog Design Assignment #1
Due Tuesday 1/21 at 9pm.

This is an individual assignment. Your solution must be your own. That said, you are highly encouraged to discuss the specification and help each other with the Verilog language. This does not mean you may or should share solutions—you will learn this language best if you solve these problems on your own.

This assignment is worth 2% of your course grade. Late submissions are not accepted.

Background
In this project you will be designing a number of different priority selectors. A priority selector (or priority arbiter) has N pairs of request and grant lines. Of those request lines which are asserted, the selector chooses one and asserts its grant line. In the most general case, more than one device might receive a grant, though for this project the number of grants given will always be one.

In this assignment you will be asked to design and test the following devices:
- A 4-bit fixed-priority selector
  - Using assign statements for the combinational logic
  - Using always blocks for the combinational logic
- A 4-bit and 8-bit fixed-priority selector using a hierarchy of 2-bit selectors.
- A 4-bit rotating priority-selector using a hierarchy of 2-bit selectors.

Before you start
In your account create a directory named “470” and change directories into it. Go to the website and download P1.tar.gz into that directory. From a Unix shell prompt type “tar xzvf P1.tar.gz”. Now change directory into the (just created) P1 directory.

4-bit fixed-priority selectors

![4-bit Priority Selector Diagram]

For this section you will design two different 4-bit fixed-priority selectors. Both are to be declared as follows:

```verilog
module ps4(req, en, gnt);
    input [3:0] req;
    input en;
    output [3:0] gnt;

    // Logic implementation
endmodule
```

req[3] is the highest-priority request, and priority goes downward to req[0] which is the lowest priority request. In all cases no more than one of the grant lines should be asserted at any given time. If en is low then no grant lines should be asserted. So for example, if en=1 and req=4'b1111, gnt should be 4'b1000.

Create a file named P1a.v. Using assign statements (and no always blocks) implement this device. You may want to use a K-map to find the logic equations needed for the grant lines or you might be able to solve it in an ad hoc method.

The Makefile is already set to use this file along with a provided testbench called test.v. Once you think you have the code correct, simply type “make”. Odds are very good this won’t work the first time and you
will have to figure out the errors. Compile errors will point you to a line number in your code, with some kind of hint as to what is wrong. If things compile it is still possible (perhaps likely) that the testbench will find an error. If that happens, you are strongly encouraged to look at the testbench and try to understand what it is doing.

Once you have completed the above task, you are to do it again, but this time using if/else statements inside of an always block. Name this file P1b.v (being sure not to change P1a.v). Make the needed changes to the Makefile so that it uses P1b.v this time.

4-bit and 8-bit fixed-priority selectors using a hierarchy of 2-bit selectors
Consider your code written in P1a.v and P1b.v. If you were going to make a 128-bit priority selector, your code would be quite long and probably pretty buggy. One way to avoid this problem is to build your module in a way that it can be combined with itself to make a larger version. For example, it is fairly easy to use three 2-input AND gates to create a single 4-input AND gate (think about how you would do this if it isn’t obvious). So how could we do this with the priority selector? (Again, think about this for a while, it isn’t easy at all...)

Let’s consider a 2-bit priority selector as a building block. In order to make a 4-bit device you’d need three of these in a tree structure as shown below. The right might get the two lowest priority request and grant lines while the left got the two highest priority grant and request lines. Then the left and the right would ask the top to choose which of them got the grant. In our previous 4-bit module we had a way to be told if we could grant to anyone, the enable line. But we didn’t have a way of saying “hey, I have a request that would like a grant”. So we will add that functionality and call it “req_up” (requesting something from the device above me). Req_up should be asserted if either request is asserted no matter the value of the enable.

So now we need to make two devices, the 2-bit priority selector, and the 4-bit priority selector built out of three of the 2-bit devices. Build the 2-bit device using either P1a or P1b as a model (your choice). Don’t forget to include “req_up”. Then you need to build the 4-bit device. To give you a bit of a start on the 4-bit device we’ve provided a simple bit of Verilog code which creates a 4-bit AND gate from three 2-bit AND gates. Be sure you understand this code before you proceed.
Use the following module declarations:

```verilog
module ps2(req, en, gnt, req_up);
  input [1:0] req;
  input en;
  output [1:0] gnt;
  output req_up;
endmodule

module ps4(req, en, gnt, req_up);
  input [3:0] req;
  input en;
  output [3:0] gnt;
  output req_up;
endmodule
```

Your code for this part should be put into P1c.v. Modify the testbench as needed to deal with the additional port on ps4. Make the needed changes to the Makefile so that you can test your code. Now in the same file use ps2 and ps4 to build an 8-bit priority selector named “ps8”. The module declarations should follow from ps4. Your testbench to test your 8-bit devices should be called testC.v. It will be turned in.

4-bit rotating priority selector

It is often the case that you’d like to build a device which uses a different priority scheme each time. This is mainly to avoid live-lock (that is, one device never gets to go because a higher priority request is always requesting. This can also be called “starvation.”) One trick is to change priorities every clock tick. Clearly we will need to introduce some sequential logic here.

In this case, the “sel” input chooses if req[0] or req[1] is to be the higher priority. Let’s say that if sel=1 req[1] has the higher priority.

In the above picture, you can see that each level of the tree uses the same select bit. You are to create a file called P1d.v. You should use P1c.v as a starting point adding a select like to ps2 and renaming ps4 to rps4. Additionally rps4 will need a 1 bit clock and reset input and a 2 bit counter output. It is to be declared as:
module rps4(clock, reset, req, en, gnt, count);
    input clock, reset;
    input [3:0] req;
    input en;
    output [3:0] gnt;
    output [1:0] count;

A testbench, named testD.v has been provided in the tar file. Be sure you’ve followed the directions above about which of sel to route where and exactly what the “sel” bit does. In all cases, the testbench needs to be passed…

Bored?
Here are some things you should look at. No points associated with any of this.

- Try to understand the three testbenches we provided.
  - What is the “correct” signal doing?
  - In testD.v there is a #6. What happens if you replace that with a #10? Why?
  - Using a counter you could do exhaustive testing (testing every case) for the fixed-priority devices. You could do it for the variable also, but it would be harder. In general, when is exhaustive testing NOT a good idea.
  - There is a $rand function which can be used to generate random test vectors. Using the web try to understand how that works. Why would you want to do this?

Turning in everything
After you believe your solution is correct make sure you have the following files in your directory:
P1a.v, P1b.v, P1c.v, P1d.v, testC.v

Now go to the higher level directory (type "cd ..") and run:
/afs/umd.edu/user/b/r/brehob/Public/470submit -p1 <directory>

Note that the script cannot deal with absolute paths, so your submission command line should be "/afs/umd.edu/user/b/r/brehob/Public/470submit -p1 P1" and be run from the directory containing P1.

At that point you should see something similar to the following:

Submitting files in P1_sol/
  --- submitting P1/
  --- submitting P1/Makefile
  --- submitting P1/testD.v
  --- submitting P1/P1b.v
  --- submitting P1/testAND.v
  --- submitting P1/test.v
  --- submitting P1/P1c.v
  --- submitting P1/P1a.v
  --- submitting P1/testC.v
  --- submitting P1/P1d.v
  --- submitting P1/And.v
Submitted.

If there is a problem it will be printed to the screen. Shortly after submission you should receive an e-mail telling you if you submission passed the basic tests. This is not an autograder! It simply means your code compiled and had module declarations that looked somewhat right. If your submission didn't pass some idea as to the problem with be provided.