EECS 470 Project #1

Note:

- This is an individual assignment. While you may discuss the specification and help one another with the SystemVerilog language, your solution – particularly the designs you submit – must be your own.
- This project is due by **Tuesday, January 23rd**
- Project 1 is not graded for synthesis, but you should still implement your designs to produce correct output in both simulation and synthesis.

1 Introduction

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. As the names imply, the selector chooses one of the asserted request lines and asserts its corresponding grant line. In the most general case, the selector can assert k grant lines, where $k \leq n$, though for this project, $k = 1$.

Priority selectors are heavily used in computer architecture, where we often have limited resources that need to be assigned optimally for best performance. The modules you write for this assignment will both remind you about the concepts of digital design you first learned in EECS 270 and begin to prepare you for the final project, where they can be reused.

For this assignment you will need to design and test the following types of priority selectors:

- You will design combinational priority selectors in 3 styles:
  - A 4-bit selector using basic `assign` statements.
  - A 4-bit selector using `always` blocks with if-else statements.
  - An 8-bit and 4-bit selector using a hierarchy of smaller selectors as Verilog modules.
- You will write a testbench for the 8-bit hierarchical selector.
- You will design a sequential 4-bit rotating priority selector using a hierarchy of 2-bit selectors.

A set of example modules for the two hierarchical designs are the `and2` and `and4` modules in `and4.sv`. The Makefile starts ready to test these modules with `and4_test.sv`. You will need to update the `TESTBENCH` and `SOURCES` variables to run other tests, either by editing the Makefile itself or running with the updated files on the commandline.

You can set the variables at the commandline with: `make TESTBENCH=mod_test.sv SOURCES=mod.sv`

2 4-bit Combinational Priority Selector

For this section, you will design two different 4-bit combinational priority selectors. Both are to be declared as in figure 1

```
module ps4 (
    input        [3:0] req,  // lines being requested
    input              en,
    output logic [3:0] gnt  // lines to grant
);
```

Figure 1: ps4 module definition

The signal `req[3]` is the highest priority request, and priority goes down 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. For example, if `en=1` and `req=4'b0101`, then `gnt=4'b0100`.

You will make two designs of this module.

In `ps4-assign.sv`: The first design should use four assign statements and Verilog logic operators to implement the selector.

In `ps4-if_else.sv`: The second design should use a single if-else chain in an `always_comb` block to implement the selector.

A testbench is provided in ps4_test.sv.

The Makefile for this project starts set-up to run the example in `and4.sv`. Update the `TESTBENCH` and `SOURCES` variables and run `make sim` to test your modules.

3 Hierarchical Priority Selectors

For this section, you will design 4-bit and 8-bit selectors using a hierarchy of modules. Both designs should go in `ps8.sv`. You will also need to write a testbench for the 8-bit module in `ps8_test.sv`. A blank file has been created for you.

Consider the Verilog you wrote in `ps4-assign.sv` and `ps4-if_else.sv`. If you were going to make a 128-bit priority selector, your design would be quite long and unreadable. 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-bit `and` gates to create a single 4-bit `and` gate. Think through how you would go about building this. Figure 2 shows how this is done in the case of the `and`. How would you go about doing this with priority selectors? It's not easy...

```
module and2 (
    input [1:0]  in,
    output logic out
);
    // a 2-bit AND operation
    assign out = in[0] & in[1];
endmodule

module and4 (
    input [3:0]  in,
    output logic out
);
    // use a variable to connect the left/right outputs to top
    logic [1:0] tmp;

    // reuse the logic inside the and2 module
    and2 left (.in(in[1:0]), .out(tmp[0]));
    and2 right(.in(in[3:2]), .out(tmp[1]));

    // combine the left and right to produce the final out signal
    and2 top(.in(tmp), .out(out));
endmodule
```

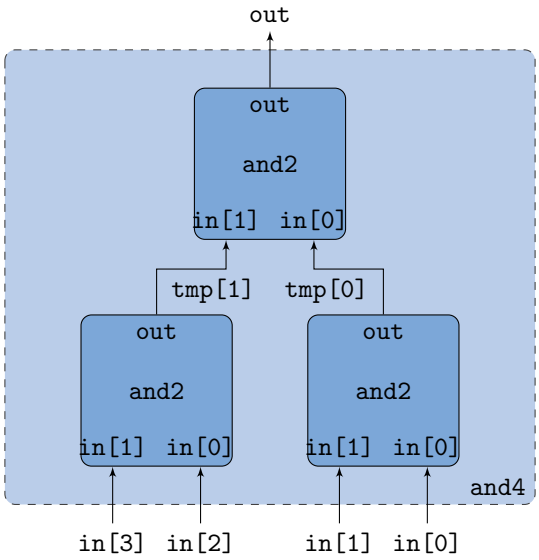

Figure 2: Hierarchical Module Example: a 4-bit `and` built with three 2-bit `and` modules.

Let's consider a 2-bit priority selector as a building block. In order to make a 4-bit selector, you can use three 2-bit selectors in a tree structure, similar to the `and` modules above. The right selector might get the two lowest priority request and grant lines, while the left gets the two highest priority request and grant lines. Then the left and right would ask the top to choose which of them is enabled to output the grant.

In our previous 4-bit module we had a way to be told if we *could* grant to anyone, the enable line (`en`). But we didn't have a way of saying "Hey, I have a request that would like a grant." We will add that functionality, as the `req_up` output for "requesting something from the device above me." This signal should be asserted if either request is asserted no matter the value of the enable.

You will need three modules, one for the base 2-bit priority selector and others for the the 4-bit and 8-bit selectors.

These modules should be declared as follows:

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

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

module ps8 (
    input        [7:0] req,
    input              en,
    output logic [7:0] gnt,
    output logic       req_up
);
```

The 2-bit selector should be implemented like either `ps4-assign.sv` or `ps4-if_else.sv`, but with extra logic for implementing the `req_up` signal (high if anything is requesting).

Once you've made the 2-bit selector, implement the 4-bit and 8-bit selectors by using fig. 2 as an example. You must use the minimum number of child modules for each design.

Your designs should output the final `gnt` lines directly from the module instantiations using the outputs of those modules. They should not be the result of an assign statement or assigned to in a combinational block. You may use an assign statement to output the `req_up` signal.

You will also need to write a testbench for the ps8 module. Write this in `ps8_test.sv` using `ps4_test.sv` as a reference (Hint: make sure to test all outputs of the module, including any new ones).

Your testbench must print "`@@@ Incorrect`" for an incorrect ps8 implementation and must print "`@@@ Passed`" for a correct implementation.

Update the `TESTBENCH` and `SOURCES` variables and run `make sim` to test your modules.

3.1 Hierarchical Rotating Priority Selectors

For this section, you will design a 4-bit rotating priority selector using a hierarchy of modules. Your design should go in `rps4.sv`.

It is often the case that you would like to evenly service all devices connected to some common resource, instead of prioritizing one over the other. This avoids *livelock* where a lower priority device is never selected because something with higher priority is constantly requesting. One way to build such a selector is to change priority every clock tick. To do this, we will need to introduce some sequential logic.

Our sequential logic comes in the form of a counter that we add to the `rps4` module. Our counter has two bits (it can count to 3!) and repeats the pattern 0, 1, 2, 3, 0... We call these our *selector* bits. The idea is that our counter decides the priority at each cycle, but also changes each cycle. This *sequential* logic enables us to evenly share the priority between different lines! We set the highest priority based on the counter's number. If the counter is at 0, requester 0 has the priority. If the counter is at 2, requester 2 has the priority. For this project, if the highest priority requester isn't requesting, you can grant to any other line.

Now look at the `rps2` module (Figure 3) we'll be using for our `rps4`. `rps2` has a sel input that represents the selector value passed in by `rps4`. We use `sel` to define our priority. If it's 1, we grant to `req` 1 over `req` 0. If `sel` is 0, we grant to `req` 0 over `req` 1.

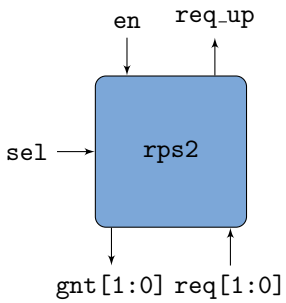

Figure 3: 2-bit rotating priority selector

Finally, Figure 4 shows how we build the `rps4` hierarchy and how the `count` bus should be used in the 4-bit selector. Each level of the tree uses one bit of the bus, the two lower level `rps2` modules share the same bit.

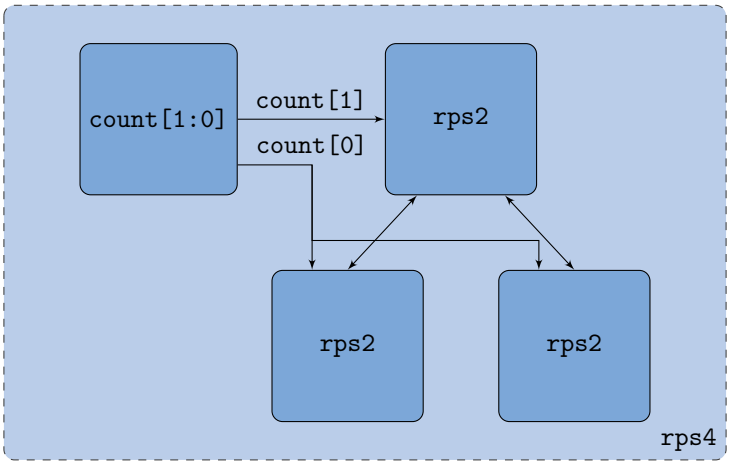

Figure 4: Hierarchy of the 4-bit rotating priority selector

The design for the `rps4` and `rps2` modules should go in the `rps4.sv` file. You should use start with `rps2` using `ps2` as a starting point and adding a select line. Additionally, `rps4` will need 1-bit `clock` and `reset` lines, and a 2-bit `count` output. The module declaration is as follows:

```
module rps4 (
    input              clock,
    input              reset,
    input        [3:0] req,
    input              en,
    output logic [3:0] gnt,
    output logic [1:0] count
);
```

A testbench has been provided in `rps4_test.sv`. Update the `TESTBENCH` and `SOURCES` variables in the Makefile and run `make sim` to test your modules.

3.2 Additional Thoughts

If you get extra time, here are some things you might want to look into. These will not be graded.

- Using a counter you could do exhaustive testing (test every possible case) for the combinational priority selectors. You could do it for the rotating ones too, but it would be harder. In general, when is exhaustive testing *not* a good idea?
- There is a `$random` function provided in Verilog for testing, which can be used to generate random test vectors. Read about it online and try to understand how it works. Why would you want to use this function?

4 Submission

To submit your project to the EECS 470 autograder, upload your solution files to the `main` branch of your GitHub repository and run the project submission script for project 1:

```
/afs/umich.edu/class/eecs470/Public/470submit 1
```

Soon after your submission you will receive an email with a summary of the public output of the autograder. This will contain the public results of correctness tests and whether the autograder encountered any errors and is not representative of your final grade.

Email the instructors or create a private Piazza post to debug any issues with the autograder.

The following files will be graded:

- `ps4-assign.sv`
- `ps4-if_else.sv`
- `ps8.sv`
- `rps4.sv`
- `ps8_test.sv`

Note: Project 1 is not graded for synthesis, but you should still implement your designs to produce correct output in both simulation and synthesis.