Mesh lab #2 – better testing

How to work as a group

This lab involves coding and simulating. You can work as a group and only turn in one set of files.

The goal of working as a team is to mirror the way you will most likely work in a real job. However, my expectation is that everyone in the group learns the code and runs the simulations. The goal is *not* to have only one person learn the material 😊.

Goals of the lab

What we'll do this time:

- Mesh lab #1 did only very rudimentary testing, and probably left you with bugs you don't yet know about. In this lab, we'll improve our test and hopefully find some of those bugs. To help you, we'll add a few short (but hopefully useful) lines of debug code.
- Get our first experience with SystemVerilog classes. Almost our entire testbench will now be based on classes.

What we won't do for now (but will do soon enough):

- Create a *tracker* to improve our ability to debug the mesh.

Big-picture instructions

- The first part of this test is to make your testing more powerful by adding an RCG. For this part, you can reuse *mesh_defs.sv*, *mesh_stop.sv* and *mesh_NxN.sv* from the previous lab. Take the new *tb_mesh_2.sv*. Flesh out the code in the *Generator* class as described below.
- Run your code and try to get it to pass these more stringent tests.
- Turn in your final *tb_mesh_2.sv* and *mesh_stop.sv* (which will likely have bug fixes from the improved testing).

The Generator class

You have a skeleton for the *Generator* class. In the previous lab, this functionality was in the *Sim_control* class, and was written so as to slowly cycle through a simple set of test vectors. However *Sim_control* combined the functionality of both a generator and driver – and we know better than that now 😊.

This lab, you will create your *Generator* class. As with any good generator, it will create *abstract* packets. It will create them pseudo-randomly, thus creating an RCG.

The testbench creates one instance of *Generator*. It then uses this instance to create new packets to give to the mesh every cycle, and to decide how long to wait before accepting a packet that the mesh has routed to its destination. The class thus has two main user-visible methods:

- **int** *make_packets* (**ref** *Abstract_packet AP_array*[*MAX_PACKETS_PER_CYCLE*]). Our previous lab gave the mesh one new packet every ten cycles. Now we will be much more flexible. The *make_packets* () method takes an array of packets. It then

decides how how many packets (call it *n*) to give to the mesh this cycle and fills in the first *n* entries of *AP_array* with pseudo-random abstract packets. It also returns *n*, so the caller knows how many of the entries in *RS_array* are valid.

- **bit** *take_results*(). The mesh-stop output *data_to_venv* is driven by FIFOs. If the verification environment always takes outgoing data as soon as it is available, we never give those FIFOs a chance to fill up (and potentially overflow). Thus, we call *take_results*() every time the mesh presents data to us. By returning 0 sometimes, this function can stress the mesh FIFOs.

You may decide to write one or more small "helper" methods in the *Generator* class if it makes your job easier, but the only mandatory methods are the two above.

Simulation-control knobs

Most random-code generators give you *knobs* to adjust their functionality. For example, do we average giving one packet to the mesh every 10 cycles, ten packets every one cycle, or somewhere in between? Will the source and destination address of the packets be totally random, or will they (e.g.,) target one particular mesh stop? If so, which one?

We implement knobs with the *Generator* class *new* () method. The *new*() method exists for all classes, and is the way that we create a new Generator object. Your Generator object should take a parameter for each knob that you use, and which sets the value of that knob. So if, e.g., we only wanted the one knob *packets_per_cycle*, we might define

```
function new (float packets_per_cycle=5);
```

We might then create a *Generator* object with

```
Generator GG = new (.packets_per_cycle(3));
```

Note that your **new** method should provide a reasonable default value for each knob. This allows the code

```
Generator GG = new ();
```

to work, which makes it easy for me to mix and match code from different groups.

The new class-based testbench

Even after all of our discussion in class about using objects and separating the generator from the driver and the monitor from the checker, last week's testbench doesn't support that. Time to fix that shortcoming! We'll do a short tour of the new code in class.

Extra features of the testbench for this assignment

In addition to being object oriented, the testbench also has additional functionality compared to the previous lab:

- supports leaving packets at a mesh-stop output for several cycles under the direction of *Generator*
- checks the packets that are delivered more stringently
- adds a few lines of code to help you debug

Debugging

Debugging your *mesh_stop.sv* with large amounts of randomly-generated packets, using nothing but waveform viewing, can be tedious! We've added a few lines of debug code in

this lab. In our next lab, we'll put together more powerful tracking software to help speed up the debug process.

If you look at *tb_mesh_2.sv*, you'll notice an *initial begin* block labeled *debugger*. It has code to look at the vertical and horizontal rings every cycle (on the falling clock edge), and to print out any valid packets. This is a pretty simple debugging facility – but may prove easier to use than looking at lots of waveforms.

You should feel free to modify this code as you see fit. You could change the printing format, add "if" statement to reduce how often messages are printed, or whatever you think might be helpful.

Questions

1. Describe the knobs that you have implemented.
2. A bit of reverse engineering for you – describe just how we now check that the mesh has correctly delivered all packets. Is it now as stringent as it can be, so as to catch as many bugs as possible?
3. Your RCG delivers random or constrained-random packets to the mesh. One common problem with random code is that it may wind up being illegal. Are there any combinations of packets that are illegal in our case? I.e., any that we simply cannot present to the mesh? If so, how does your RCG avoid them?
4. You had a rudimentary (but hopefully still useful) debug facility to print out packets in flight. What do you think might have helped your debugging still more?

What to turn in:

- Turn in your final *tb_mesh_2*.sv, as well as your latest *mesh_stop.sv* and a .pdf with answers to the questions.
- Just have one person per team turn in work; no need for every individual to turn in something separate.