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.
- Get our first experience with SystemVerilog classes. We'll use a class object to generate our stimulus.

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

- Create a *tracker* to help us debug the mesh.

Big-picture instructions

- 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 *Sim_control* 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 Sim_control class

You have a skeleton for the *Sim_control* class. In the previous lab, this class was written so as to slowly cycle through a simple set of test vectors. This lab, you will completely rewrite *Sim_control* and thus create an RCG. The testbench creates one instance of *Sim_control*. It then uses the 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** *Ring_slot RS_array*[*MAX_PACKETS_PER_CYCLE*]). Our previous lab gave the mesh one new packet every ten cycles. Now we will 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 *RS_array* with pseudo-random packets. It also returns *n*, so the user 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 this function every time the mesh presents data to us. By returning 0 sometimes, this function can stress the mesh FIFOs more.

You may decide to write one or more small “helper” methods in the *Sim_control* 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 *Sim_control* class *new* () method. The *new*() method exists for all classes, and is the way that we create a new Sim_control object. Your Sim_control 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 *Sim_control* object with

```
Sim_control SC = new (.packets_per_cycle(3));
```

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

```
Sim_control SC = new ();
```

to work, which makes it easy for me to mix and match code for the upcoming debug derbies.

How the testbench works

Other than you rewriting the *Sim_control* class, the testbench has only minor changes compared to the previous lab:

- supports leaving packets at a mesh-stop output for several cycles under the direction of *Sim_control*
- checks the packets that are delivered more stringently

Debugging

Debugging your *mesh_stop.sv* with large amounts of randomly-generated packets, using nothing but waveform viewing, can be tedious! In our next lab, we’ll put together tracking software to help speed up the debug process.

We don’t have that code yet for this lab. You may want to put some thought into how to do your debugging, as well as talk to your instructor(s) before you spend *too* many hours at it.

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.
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?

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.