Mesh lab #5 – Verifying somebody else's code

How to work as a group

This lab involves coding, simulating and writing a report. 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

In this lab, we'll work in a situation that is much more like real life — verify code that is difficult and that was written by somebody else.

Instructions

- Take brand new design files mesh_stop_challenge.sv and mesh_NxN_challenge.sv. These implement a capability of routing around dead links. While this capability is substantially simpler than a commercial implementation, it is also substantially more complex than what you have seen so far.
- Start by taking a brand-new verification testbench tb_mesh_challenge.sv. This testbench has an RCG as usual, but with the additional ability to randomly assign dead links.
- The test benches (hopefully!) do not have any bugs. However, the new hardware has multiple bugs. Each time you find one, you have two choices. You can report the bug to me, along with a description of what the bug is, and I will give you a new mesh_stop.sv with the bug fixed. Or, if you prefer (since you may be doing this at a time when I am not available), you can fix the hardware yourself.
- Feel free to ask me any questions you like about how the hardware works. I will probably answer them, but not always promptly or even fully. This is on purpose and is meant to be a feature of this lab — in real life, you will not always be the design team's top priority. (Sorry!)

What to turn in:

- Turn in your final *mesh_stop.sv*, as well as a .pdf with a description of each of the bugs and how you found it.
- As usual, just have one person per team turn in work; no need for every individual to turn in something separate.

The testbench:

The testbench has several knobs that you can play with. Alternatively, you can use your own testbench, and merely cut/paste the dead-link code (which is marked by the text NEW CODE, or which you can find by hunting for "dead" in the testbench) from this testbench to your own. Or you can add your own tracker to this code if you like.

The testbench has several useful knobs:

- N_PACKETS_TO_SEND: How long the test will be
- MAX_PACKETS_PER_CYCLE: the maximum number of packets ever launched in any one cycle
- LAUNCH_FREQ: controls the actual number of packets launched per cycle. It should be in the range [0,10*MAX_PACKETS_PER_CYCLE]. For example, 20 would mean that we launch exactly 2 packets/cycle; 30 would mean that we launch exactly 3 packets/cycle; and 24 would mean that we launch 2 packets 40% of the time and 3 packets 60% of the time.

- HOW_OFTEN_TO_TARGET: a number in the range [0,10] that says what fraction of the packets target a single mesh stop (0 means that all destinations are random, and 10 means that every packet targets the same mesh stop).
- TAKE_RES_FREQ: a number in [0,10], where 10 means that the environment takes output packets every cycle that the mesh has one to give, and 0 means that the environment never takes output packets (which of course would make your simulation never finish!)
- DEAD_LINK_FREQ: a number in [0,10] that controls what fraction of links are set to be dead (0 means no dead links, and 9 means that 90% of the links are randomly set to be dead). However, we never set all four links in any row to be dead – that would prevent successful routing.

***The bugs*:**

A small hint to you: there is one bug that has nothing at all to do with dead links. Then two dead-link bugs that have relatively simple fixes. Finally, there is one dead-link bug that is essentially unfixable.

Functional description of routing around broken links

We have decided to only handle broken vertical links, rather than both horizontal and vertical links. This is not because of any physical reason that vertical links are more fragile than horizontal; it is merely to keep the lab reasonably simple and focus on your verification skills.

Consider the places where a broken vertical link can force us to change our routing strategy. Start with a packet traveling from MS00 to MS22. It would normally first travel vertically from MS00 through MS10 and then turn right at MS20, finally traveling horizontally to MS22. What if the vertical link from MS00 to MS10 were broken?

Obviously, the packet could then not travel vertically from MS00 to MS10. Instead, it must leave MS00 on the horizontal ring, and thus go to MS01. At this point, it could simply continue traveling horizontally to the destination column at MS02, then do a ring turn to travel vertically, finally going through MS01 to reach MS22.

In order to do this, we need several new features. First: a new packet, leaving the DrvF, must be able to adapt to a broken link. If (the usual case), it is supposed to leave on the vertical ring, it must be able to leave on the horizontal ring instead (i.e., if the outgoing vertical link is broken). Note that we do not need to handle the opposite case (a packet that is supposed to leave on the horizontal ring but must go vertically instead due to a broken link), since we assumed that horizontal links never break.

A similar issue can happen if (keeping our example of a packet from MS00 to MS22) the vertical link from MS10 to MS20 is broken. The packet must now do unexpected ring turn, leaving MS10 on the horizontal ring towards MS11.

With either of the two issues above, we now have a new situation to deal with. Previously, packets only entered a horizontal ring when they reached their final destination row; so any packet on a horizontal ring wire, by the nature of our algorithm, could simply stay on that ring, in that row, until it reached the destination column – at which point it was done.

Now, however, we must consider the possibility that a packet is traveling on a horizontal ring not because that is the correct destination row, but because the packet was unfortunately diverted to that row due to a broken vertical link. In this situation, we will have the packet continue on that row until it reaches its destination column. At that point, we will have the packet do a brand new trick – a horizontal to vertical ring turn – and travel on this vertical ring until it reaches its destination mesh stop (MS22 in our example).

These are the new high level functional capabilities we have an added to support routing around a broken link. Next, let's take a slightly deeper dive into exactly how we implemented these features.

Start with the driver FIFO. It already has the capability of driving packets to either the vertical or the horizontal ring, so we will not need any new datapath. However, we do need extra control logic to force a packet leaving the FIFO onto the horizontal ring if the outgoing vertical link is broken.

Next consider the logic that handles packets arriving from the vertical ring. It already knows how to send an incoming packet from the vertical ring to the horizontal ring with a ring turn for the case of a packet that has reached its correct destination row. It now will need a little bit more control logic, so as to also force a packet to do a ring turn if the outgoing vertical link is broken. I.e., the vert_sel_pass case must be converted to vert_sel_turn when a broken vertical leg prevents the vert_sel_pass case from being possible. That's it for this section of logic!

Now it's time to let the fun begin with our deep dive into the final, trickiest logic – handling a packet traveling horizontally but not on its destination row. When a horizontal-ring packet reaches its

destination column, we already have logic to pull the packet off of the horizontal ring and place it into the HRxF. All good so far; when the packet leaves the HRxF, we now have two choices. First, if the packet is a “normal” packet traveling on its correct destination row, it should go to the verification environment just as before. However, if it is on the “wrong” row, it must now take a ring turn onto the vertical ring. This requires new datapath to be added, as well as control logic to tell these two cases apart from each other. It also requires that the vertical ring-driver mux, which has the two legs for vert_sel_pass and vert_sel_me, now get a new vert_sel_turn (appropriately prioritized) that receives data from the HRxF and drives it onto the outgoing vertical ring.

And that’s about it – just be thankful that we did not double the complexity by allowing broken horizontal links 😊.

How to break a link

We've talked about how a mesh stop must change to route around broken vertical links now let's talk about how a mesh stop knows which links are broken, and how we actually break a link(s).

How much information does a mesh stop need about which vertical links in the mesh, if any, are broken? The algorithms discussed above are *local*, in the sense that they only require that a mesh stop know if its outgoing vertical link is broken; in particular, a mesh stop does *not* need to know the status of any vertical link other than that one. We thus add one more signal *vert_link_dead* to module mesh_stop. In a real system, the hardware itself would be responsible for detecting broken links and driving *vert_link_dead* correctly to each mesh stop. For us, however, the verification environment will drive it.

What mechanism should the verification environment use to decide which link or links to break? We will add one more function *set_dead_links*() to our RCG class Sim_control. It uses a parameter to decide how often to randomly break links. At simulation start time when The environment instantiates Sim_control, it calls *set_dead_links*() and gets a 4x4 array of bits, of which it drives one bit into the *vert_link_dead* input of each mesh_stop. Of course, you are free to ignore this 4x4 array and, especially for your initial debug, simply drive one or two inputs of your choice to be dead.

Driving *vert_link_dead* inputs high "should" get the mesh stops to correctly reroute packets. However, it does not physically break the actual mesh connections. Thus, a mesh could actually route a packet over a "broken" link without any consequences. To close this loophole, the verification environment adds one more trick. In the middle of every cycle, it checks the links that it has decided to declare is broken. If any of those links ever holds a valid packet, the environment immediately flags an assertion error to stop the simulation.