# FIFO lab #3 – checking and improving coverage

***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

- Again, not touch our FIFO itself at all.
- Improve our testbench by checking *functional coverage* to measure how well it tests the FIFO, including functional *cross-product* coverage.
- Hopefully increase the coverage by improving our test.

***Big-picture instructions***

- Reuse your two files from the previous lab. Don't touch *fifo.sv* unless you want to, since this lab only changes the testbench code and not the DUT. Our improved version of *tb_fifo_2.sv* will now be called *tb_fifo_3.sv*.
- Add functional-coverage checking to your code as detailed below, and print out the level of coverage you're achieving.
- Make the recommended changes to improve your coverage, rerun and again print out the level of coverage you're achieving.
- Turn in your final *tb_fifo_3.sv*, as well as a .pdf with answers to the questions below.

***What is functional coverage?***

We'll talk in class about different types of test coverage. Functional coverage is one of the most important types of coverage. The nice thing about it is that it kind of means whatever you want it to mean 😊. Perhaps a bit more precisely, achieving good functional coverage means that you have tested the signals that you believe are most important to the functionality of the design.

The wonderful thing about functional coverage is that we get to decide for ourselves what is important! Of course, there are other types of coverage to check also.

What are the important signals for our FIFO? We'll start with *empty* and *full*. We would like to make sure that *empty* and *full* each take values of both 0 and 1 at some point in the test; if they don't, then arguably we've not sufficiently tested the FIFO.

Next, we might want to check that both the *rd_ptr* and *wr_ptr* go through all of their legal values. Again, if not, then arguably we're not done testing. In fact, we can take this one step further and decide that we want to cover the *cross-product* of all (in our case) 8 values of *rd_ptr* against all 8 values of *wr_ptr*. I.e., we want the FIFO to go through all 64 of those combinations at some point in the test.

Why is getting good functional coverage so important? Because it's one more factor leading to you knowing when your design "probably works."

***Using covergroups to check functional coverage***

The tool we'll use to check coverage is called a *covergroup*. You can find more detail in Spear chapter 9, or in chapter 19 of the SystemVerilog LRM.

You could add a covergroup to your testbench with code like:

```
covergroup fifo_cov @(posedge clk);
      rp: coverpoint F.rd_ptr;
      wp: coverpoint F.wr_ptr;
      empty: coverpoint F.empty;
endgroup
fifo_cov cov = new;
```

Note that this code would only check coverage on *rd_ptr*, *wr_ptr* and *empty*. You still need more code to check the cross product of *rd_ptr* and *wr_ptr* and to check *full*.

Once the simulation is done, you can check coverage on (e.g.,) the read pointer by using

```
int perc, n_bins_cov, n_bins;
perc = cov.rp.get_inst_coverage(n_bins_cov, n_bins);
$display ("Rd_ptr coverage = %0d%%: %0d / %0d bins", perc, n_bins_cov,
n_bins);
```

Once you've measured your coverage, save your printout to answer question #1 below.

***Improving functional coverage***

Our current strategy is to both read and write the FIFO every cycle. Let's try adding randomness. Instead of writing the FIFO every cycle, let's write it on average every other cycle (being sure not to write it when it's full). Instead of reading the FIFO every cycle, let's read it on average every other cycle (being sure not to read it when it's empty).

We can do this using **if** statements and using the function *$urandom_range*(*max*), which returns a random integer in the range [0,max]. E.g., we might get a random integer that is either 0 or 1 and then write the FIFO if we've gotten a "1".

Use this strategy to randomize the reading and writing, and then see how this affects your coverage numbers.

***Questions to answer:***

1. Show your original functional coverage. For each category (*empty, full, rd_ptr, wr_ptr, rd_ptr x wr_ptr*), first explain why it has the given number of bins. Then explain the the coverage fraction. Why was the cross-product coverage fraction so low?
2. Given our original strategy of writing and reading the FIFO every cycle, how would you expect the coverage to change if we simulated 100x more cycles? Why?
3. Show your functional coverage after changing to a random-writing-and-reading strategy. Can you explain any changes in the coverage?
4. Given our new strategy of writing and reading the FIFO more randomly, how would you expect the coverage to change if we simulated 100x more cycles? Why?
5. You have hopefully ensured that you do not try to write a full FIFO or read an empty one. Is there any *legal* corner case that you've missed by following this strategy?

***What to turn in:***

- Turn in your final *fifo.sv* (which is probably the same as in the first lab) and *tb_fifo_3.sv*, as well as a .pdf with your answers to the questions.
- Just have one person per team turn in work; no need for every individual to turn in something separate.