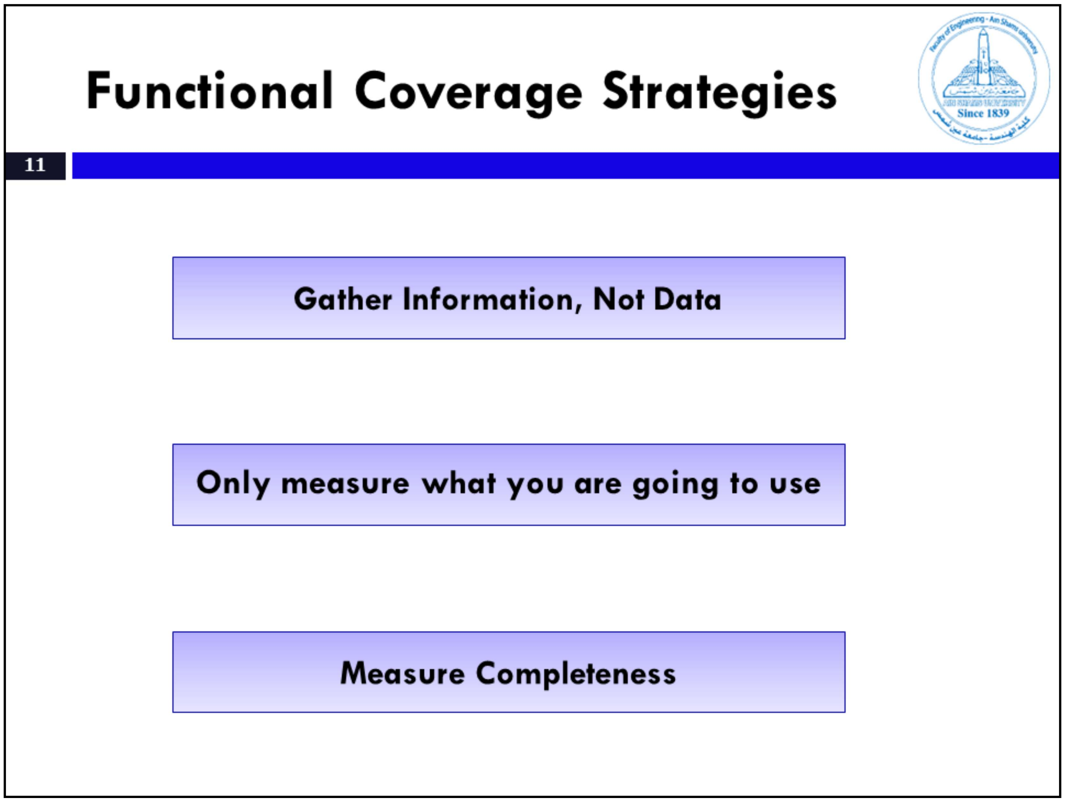


Functional Coverage Strategies

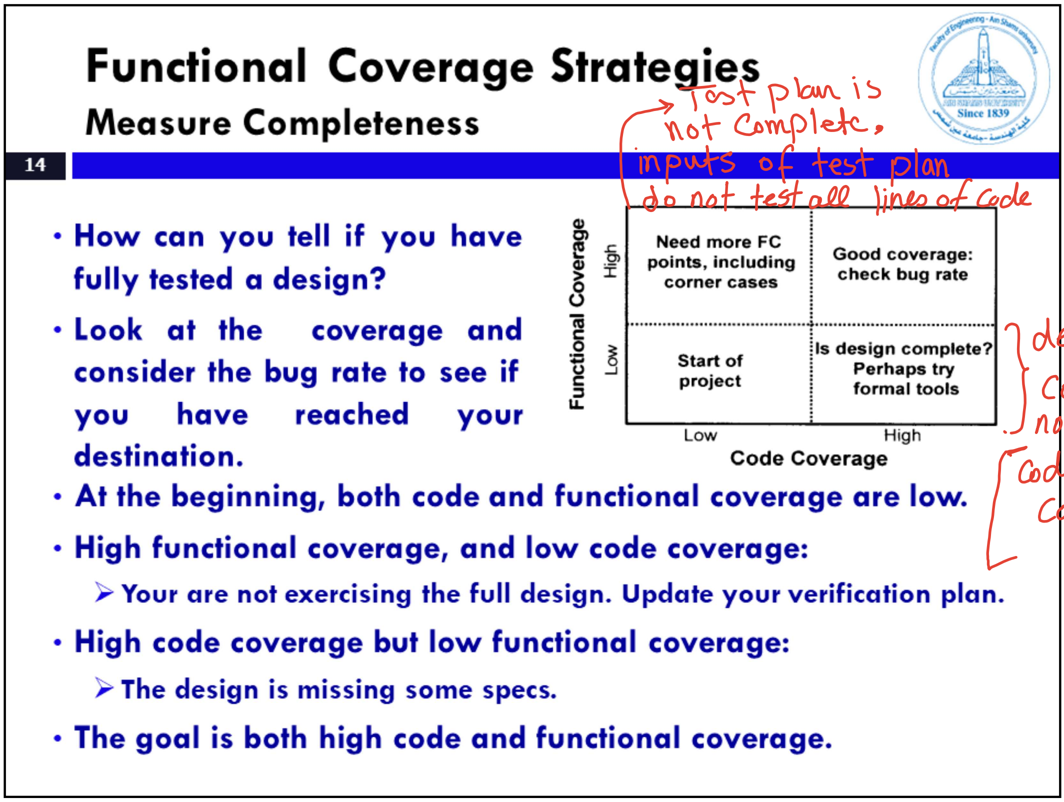
11

Gather Information, Not Data

Only measure what you are going to use

Measure Completeness

- Before you write the first line of test code, you need to anticipate:
  - ➢ The key design features
  - ➢ The comer cases
  - ➢ The possible failure modes.
- This is how you write your verification plan. Don't think in terms of data values only; instead, think about what information is encoded in the design.



Functional Coverage Strategies

Measure Completeness

14

- How can you tell if you have fully tested a design?
- Look at the coverage and consider the bug rate to see if you have reached your destination.

Functional Coverage

	Low	High
High	Need more FC points, including corner cases	Good coverage: check bug rate
Low	Start of project	Is design complete? Perhaps try formal tools

Code Coverage

- At the beginning, both code and functional coverage are low.
- High functional coverage, and low code coverage:
  - ➤ Your are not exercising the full design. Update your verification plan.
- High code coverage but low functional coverage:
  - ➤ The design is missing some specs.
- The goal is both high code and functional coverage.

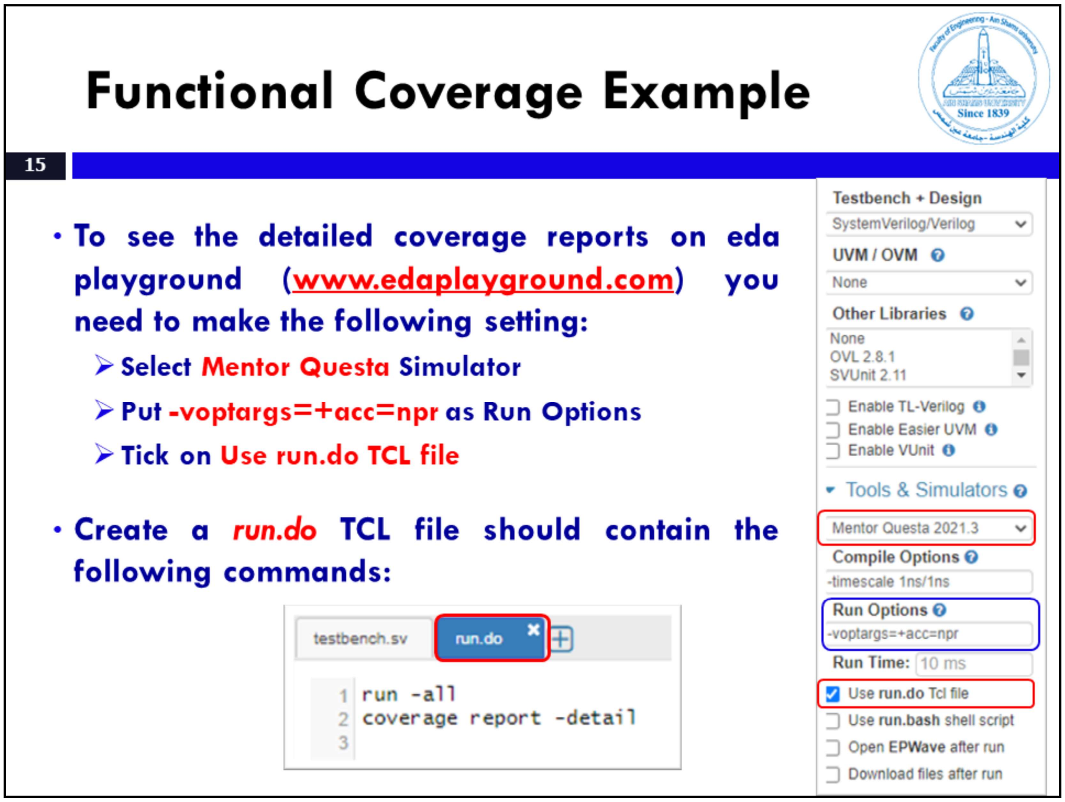
Test plan is not Complete, inputs of test plan do not test all lines of code

design Code is not Complete

Code does not Cover all functionality

- Your manager constantly asks you, “Are we there yet?” How can you tell if you have fully tested a design? You need to look at all coverage measurements and consider the bug rate to see if you have reached your destination.
- At the start of a project, both code and functional coverage are low.
- As you develop tests, run them over and over with different random seeds until you no longer see increasing values of functional coverage.
- Create additional constraints and tests to explore new areas. Save test/seed combinations that give high coverage, so that you can use them in regression testing.

- **What if the functional coverage is high but the code coverage is low?** Your tests are not exercising the full design, perhaps from an inadequate verification plan. It may be time to go back to the hardware specifications and update your verification plan. Then you need to add more functional coverage points to locate untested functionality.
- **A more difficult situation is high code coverage but low functional coverage.** Even though your testbench is giving the design a good workout, you are unable to put it in all the interesting states. First, see if the design implements all the specified functionality. If the functionality is there, but your tests can't reach it, you might need a formal veritication tool that can extract the design's states and create appropriate stimulus.
- The goal is both high code and functional coverage. However, don't plan your vacation yet. What is the trend of the bug rate? Are significant bugs still popping up?
- On the other hand, a low bug rate may mean that your existing strategies have run out of steam, and you should look into different approaches.



Functional Coverage Example

- To see the detailed coverage reports on eda playground (www.edaplayground.com) you need to make the following setting:
  - Select Mentor Questa Simulator
  - Put -voptargs=+acc=npr as Run Options
  - Tick on Use run.do TCL file
- Create a *run.do* TCL file should contain the following commands:

testbench.sv | run.do

```
run -all
coverage report -detail
```

Testbench + Design
SystemVerilog/Verilog

UVM / OVM
None

Other Libraries
None
OVL 2.8.1
SVUnit 2.11

□ Enable TL-Verilog
□ Enable Easier UVM
□ Enable VUnit

Tools & Simulators
Mentor Questa 2021.3

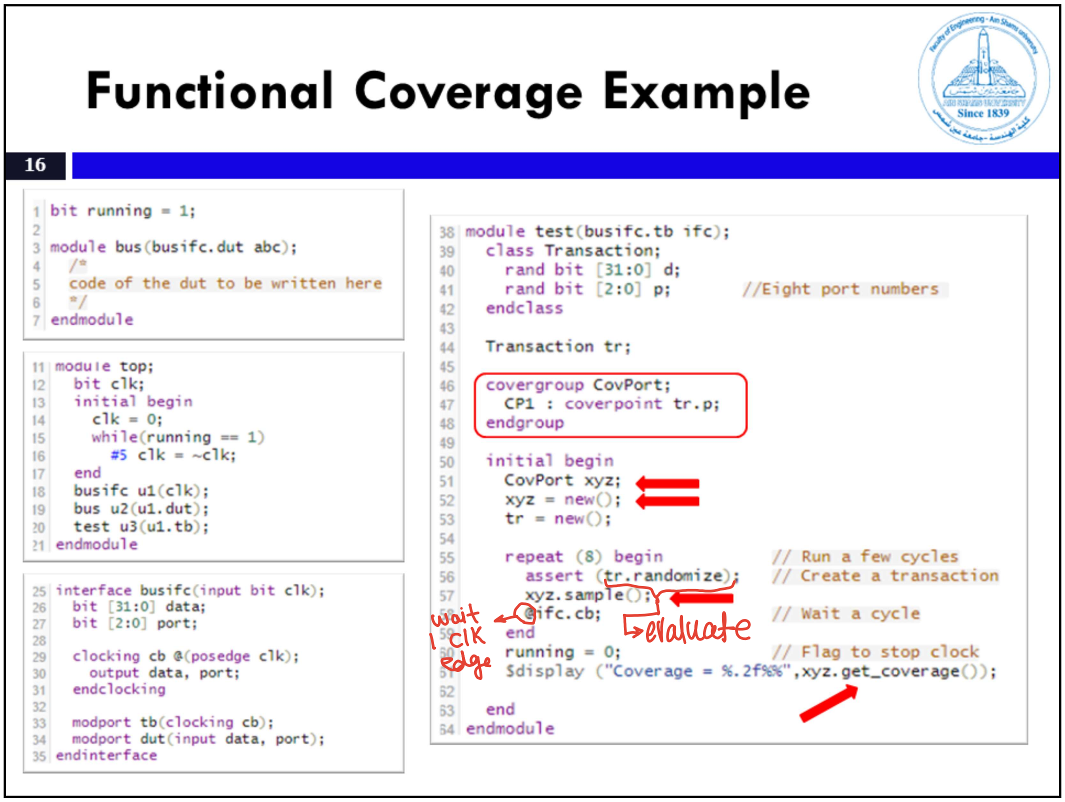
Compile Options
-timescale 1ns/1ns

Run Options
-voptargs=+acc=npr

Run Time: 10 ms

☑ Use run.do Tcl file
□ Use run.bash shell script
□ Open EPWave after run
□ Download files after run

- T



Functional Coverage Example

```
bit running = 1;

module bus(busifc.dut abc);
  /*
  code of the dut to be written here
  */
endmodule
```

```
module top;
  bit clk;
  initial begin
    clk = 0;
    while(running == 1)
      #5 clk = ~clk;
  end
  busifc u1(clk);
  bus u2(u1.dut);
  test u3(u1.tb);
endmodule
```

```
interface busifc(input bit clk);
  bit [31:0] data;
  bit [2:0] port;

  clocking cb @(posedge clk);
    output data, port;
  endclocking

  modport tb(clocking cb);
  modport dut(input data, port);
endinterface
```

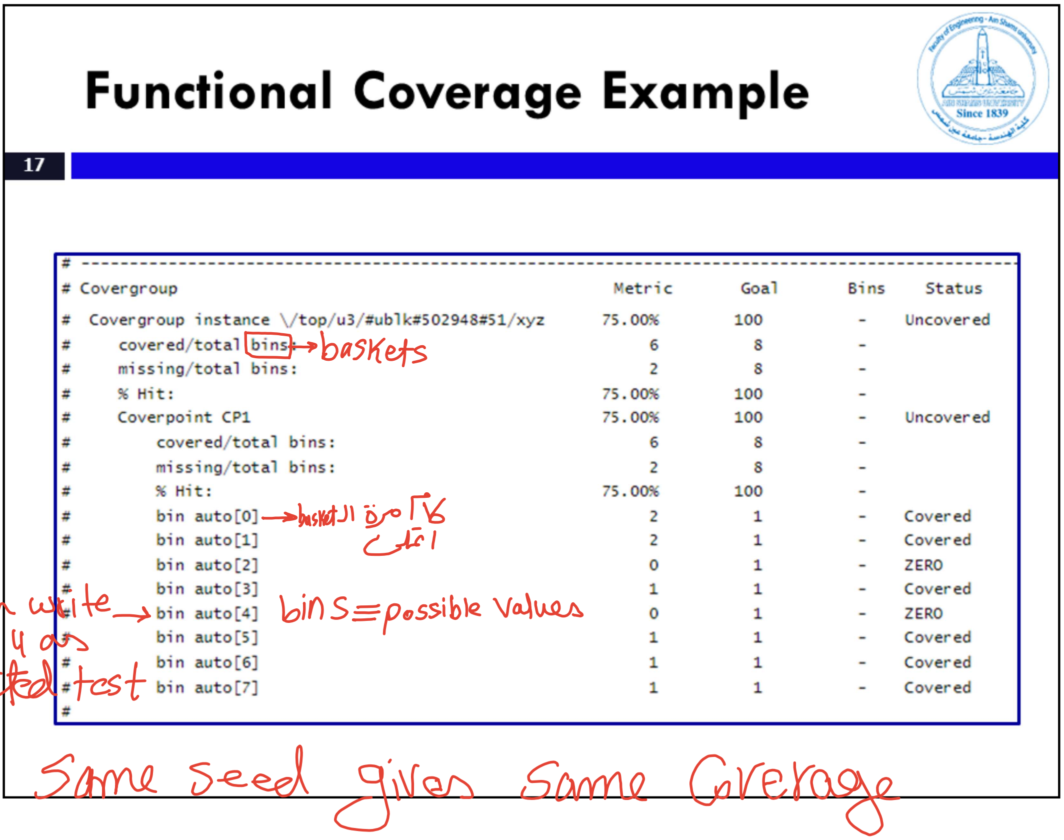
```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [31:0] d;
    rand bit [2:0] p;        //Eight port numbers
  endclass

  Transaction tr;

  covergroup CovPort;
    CP1 : coverpoint tr.p;
  endgroup

  initial begin
    CovPort xyz;
    xyz = new();
    tr = new();

    repeat (8) begin             // Run a few cycles
      assert (tr.randomize);     // Create a transaction
      xyz.sample();
      @ifc.cb;                   // Wait a cycle
    end
    running = 0;                 // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());

  end
endmodule
```

wait 1 clk edge

evaluate

- To measure functional coverage, you begin with the veritication plan and write an executable version of it for simulation.
- In your SystemVerilog testbench, sample the values of variables and expressions. These sampling locations are known as *cover points*.
- Multiple cover points that are sampled at the same time (such as when a transaction completes) are placed together in a *cover group*.
- The shown code has a transaction that comes in eight flavors (you have 8 possible values for the port). The testbench generates the port variable randomly, and the verification plan requires that every value be tried.
- The testbench samples the value of the port field using the CovPort cover group. The above code generates 8 random transactions. Did they cover all the possible values of the port? Did your testbench generate them all?
- To know the answer we have to look at the coverage report. Here is part of a coverage report from QuestaSim (In the next slide).

- **Practical note:**
- If $finish is called explicitly, or implicitly (when a *program* finishes execution), eda playground will not run the TCL file run.do, and you will not see the detailed report. You will see only the line reporting the coverage from the get_coverage() function.
- That is why we used a module for the testbench instead of a program.



Functional Coverage Example

17

```
# --------------------------------------------------------------------------------------
# Covergroup                                          Metric      Goal      Bins     Status
#  Covergroup instance \/top/u3/#ublk#502948#51/xyz   75.00%      100        -      Uncovered
#     covered/total bins:                                  6        8         -
#     missing/total bins:                                  2        8         -
#     % Hit:                                          75.00%      100        -
#     Coverpoint CP1                                  75.00%      100        -      Uncovered
#         covered/total bins:                              6        8         -
#         missing/total bins:                              2        8         -
#         % Hit:                                      75.00%      100        -
#         bin auto[0]                                      2        1         -      Covered
#         bin auto[1]                                      2        1         -      Covered
#         bin auto[2]                                      0        1         -      ZERO
#         bin auto[3]                                      1        1         -      Covered
#         bin auto[4]                                      0        1         -      ZERO
#         bin auto[5]                                      1        1         -      Covered
#         bin auto[6]                                      1        1         -      Covered
#         bin auto[7]                                      1        1         -      Covered
#
```

bins → baskets

bin auto[0] → كام مرة الbasket
اتملت

bins ≡ possible Values

we can write value 4 as directed test → bin auto[4]

Same Seed gives Same Coverage

- As you can see, the testbench generated the values 0,1,3,5,6, and 7, but never generated a port value of 2 nor 4.
- The "*Goal*" column specifies how many hits are needed before a bin is considered covered.
- To improve your functional coverage, the easiest strategy is to just run more simulation cycles, or to try new random seeds.
- Look at the coverage report for items with two or more hits. Chances are that you just need to make the simulation run longer or to try new seed values.
- If a cover point had zero or one hit, you probably have to try a new strategy, as the testbench is not creating the proper stimulus.

- Bins are the basic units of measurement for functional coverage. When you specify a variable or expression in a cover point, System Verilog(SV) creates a number of "bins" to record how many times each value has been seen.
- The bins can be explicitly defined by the user or automatically (implicitly) created by SV. The number of bins created implicitly can be controlled by *auto_bin_max* parameter as will be seen later.
- To calculate the coverage for a point that has 3-bit variable (has the domain 0:7) can be normally divided into 8 bins. So coverage is then measured as the number of non-empty bins divided by the total number of bins in the domain. If during simulation, values belonging to seven bins are sampled, the report will be 7/8 or 87.5% coverage for this point.

total Coverage = average of assertion and functional Coverage.
#