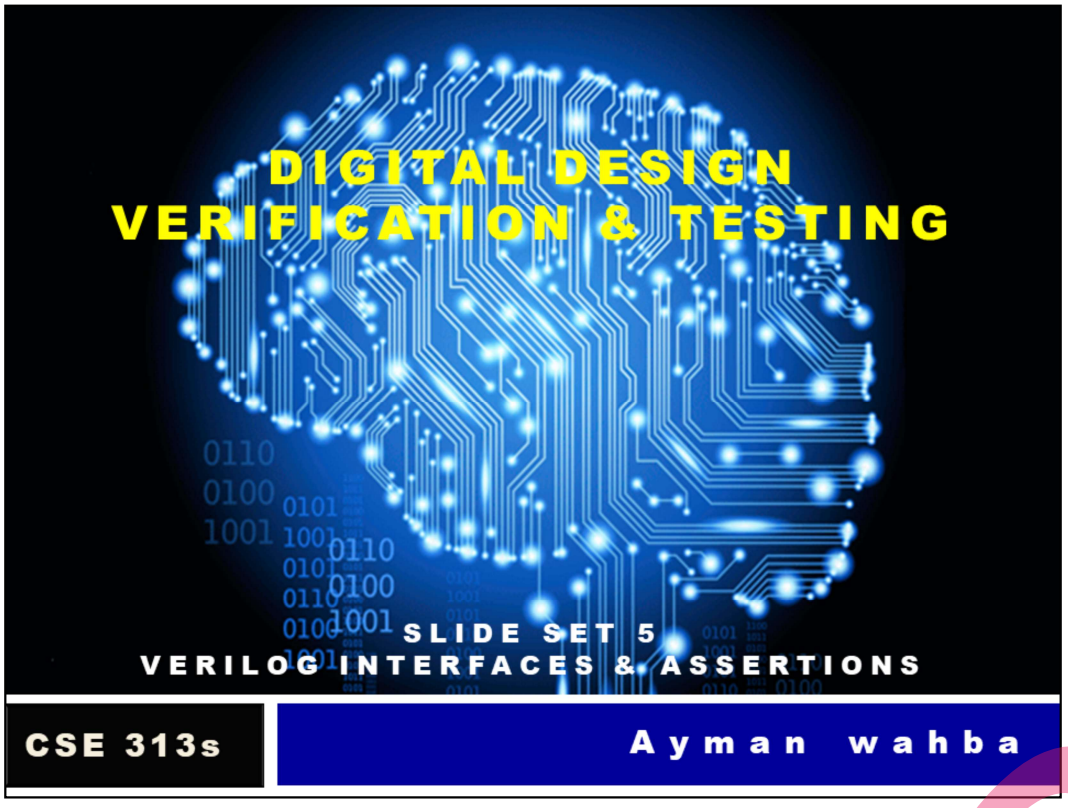
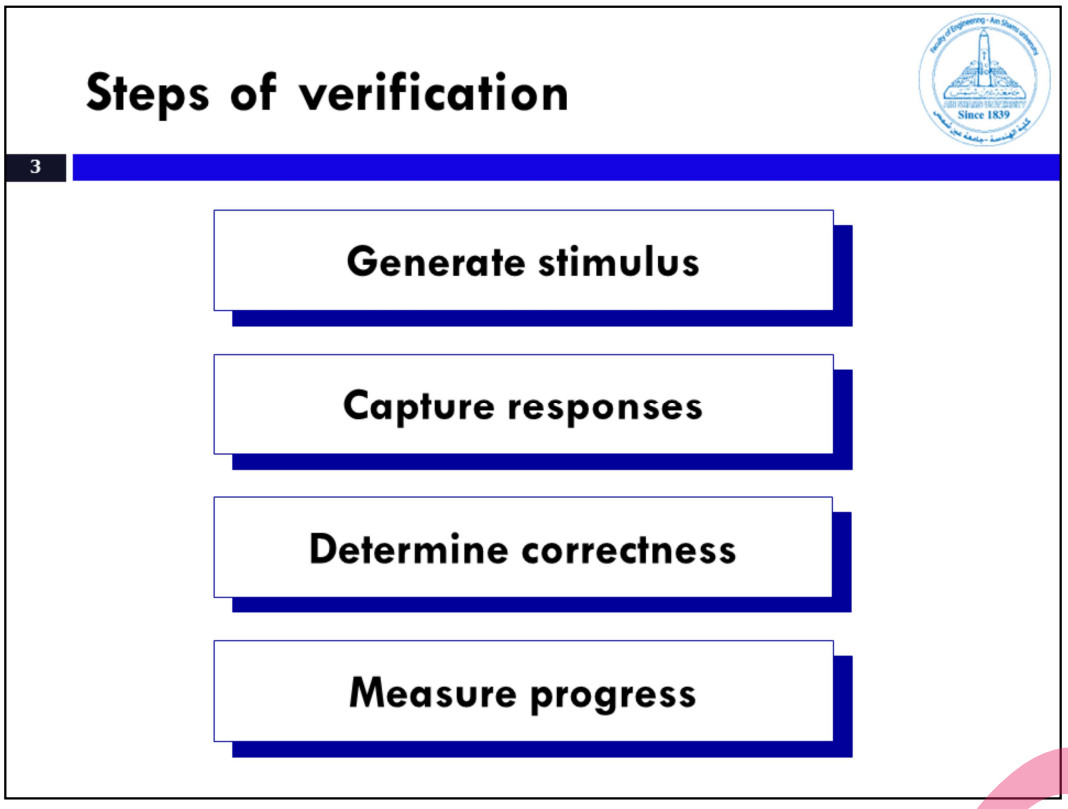

DIGITAL DESIGN
VERIFICATION & TESTING
SLIDE SET 5
VERILOG INTERFACES & ASSERTIONS
CSE 313s
Ayman wahba

Connecting the Testbench and the Design

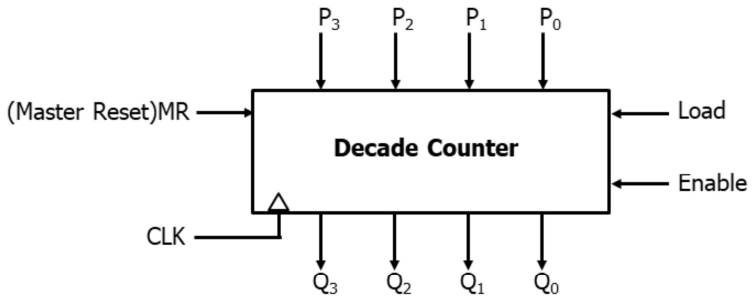


- There are several steps needed to verify a design:
  1. Generate stimulus,
  2. Capture responses,
  3. Determine correctness, and
  4. Measure progress.

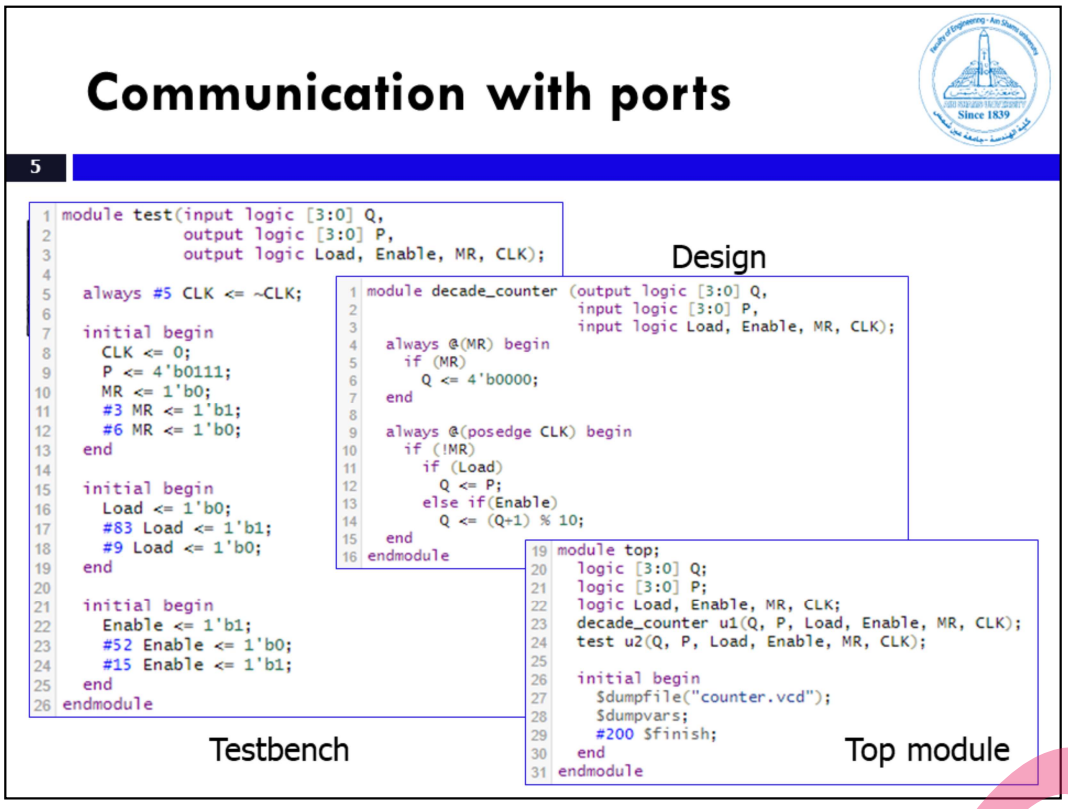
Simple example (decade counter)

4

- 4-bit up/down counter
- Asynchronous reset (MR)
- Synchronous load ($Q_{3:0} \leftarrow P_{3:0}$)
- Enable count



- As a starting example, let's model and test a decade counter
- It counts from 0 to 9, in the normal cases, when Enable is active
- It has an asynchronous reset signal MR (Master Reset).
- It ban be loaded by any value P, using the synchronous Load line.




```
module test(input logic [3:0] Q,
            output logic [3:0] P,
            output logic Load, Enable, MR, CLK);

  always #5 CLK <= ~CLK;

  initial begin
    CLK <= 0;
    P <= 4'b0111;
    MR <= 1'b0;
    #3 MR <= 1'b1;
    #6 MR <= 1'b0;
  end

  initial begin
    Load <= 1'b0;
    #83 Load <= 1'b1;
    #9 Load <= 1'b0;
  end

  initial begin
    Enable <= 1'b1;
    #52 Enable <= 1'b0;
    #15 Enable <= 1'b1;
  end
endmodule
```

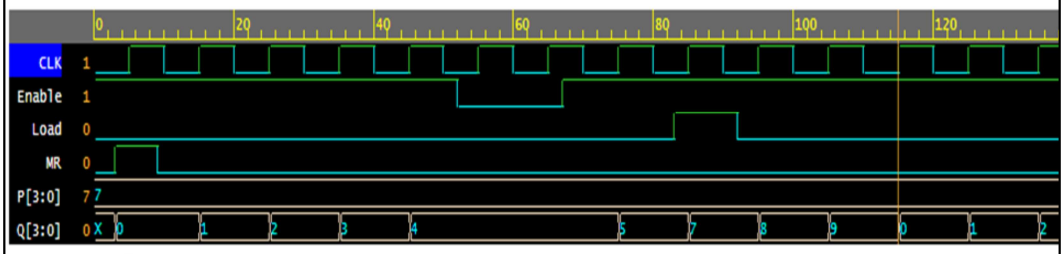
```
module decade_counter (output logic [3:0] Q,
                       input logic [3:0] P,
                       input logic Load, Enable, MR, CLK);

  always @(MR) begin
    if (MR)
      Q <= 4'b0000;
  end

  always @(posedge CLK) begin
    if (!MR)
      if (Load)
        Q <= P;
      else if(Enable)
        Q <= (Q+1) % 10;
  end
endmodule
```

```
module top;
  logic [3:0] Q;
  logic [3:0] P;
  logic Load, Enable, MR, CLK;
  decade_counter u1(Q, P, Load, Enable, MR, CLK);
  test u2(Q, P, Load, Enable, MR, CLK);

  initial begin
    $dumpfile("counter.vcd");
    $dumpvars;
    #200 $finish;
  end
endmodule
```

- In the shown code, the netlists are simple.
- In real designs, you may find hundreds of pins that require pages of signal and port declarations.
- All these connections can be error prone. As a signal moves through several layers of hierarchy, it has to be declared and connected over and over. (Here for example, the same signals are defined in the top module, the design, and the testbench).
- Worst of all, if you just want to add a new signal, it has to be declared and connected in multiple files.
- SystemVerilog ***interfaces*** can help in each of these cases.

Capture responses & determine correctness

6

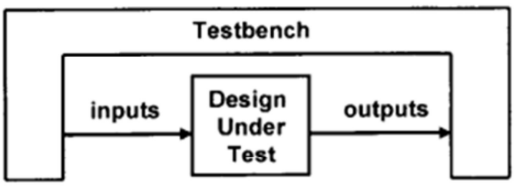


- Using the generated waveforms from simulation, you visually inspect the responses, and determine the correctness of your design.
- Again, this is possible for this simple design, but with large designs, we may need an automated process to make sure of the correctness of all the aspects of the design.
- In this example, we generated the stimulus manually to test each feature of the design. In big designs, we may need to use random inputs, but in the same time we may need to measure what is called the coverage, to make sure we tested all features of the design.

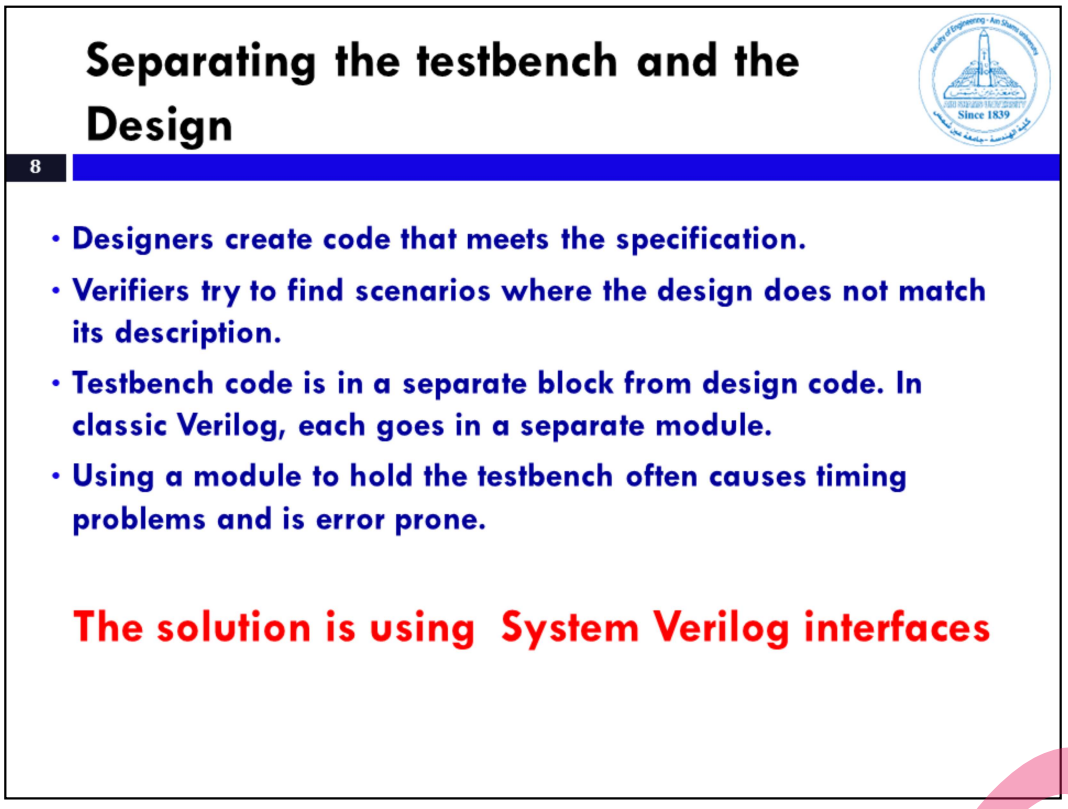
Connecting the testbench to the Design

7

- The testbench wraps around the design, sending in stimulus and capturing the design's response.
- The testbench forms the "real world" around the design, mimicking the entire environment.
- The key concept is that the testbench simulates everything *not* in the design under test.



- Your testbench wraps around the design, sending in stimulus and capturing the design's response.
- The testbench forms the "real world" around the design, mimicking the entire environment. For example:
  - A processor model needs to connect to various buses and devices, which are modeled in the testbench.
  - A networking device connects to multiple input and output data streams that are modeled based on standard protocols in the testbench.
  - A video chip connects to buses that send in commands, and then forms images that are written into memory models.
- The key concept is that the testbench simulates everything not in the design under test.
- Your testbench needs a higher-level way to communicate with the design than verilog's ports.
  - You need a robust way to describe the timing so that synchronous signals are always driven and sampled at the correct time and all interactions are free of the race conditions, that are so common in Verilog models.

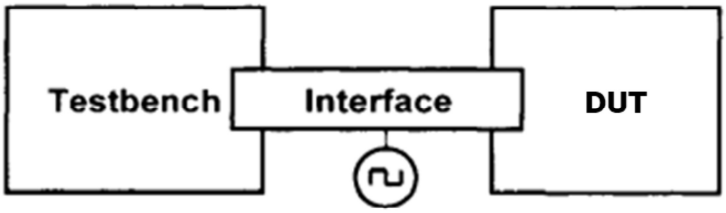


- In an ideal world. all projects have two separate groups: one to create the design and one to verify it, but in the real world, limited budgets may require you to wear both hats.
- The designer has to create code that meets that specification, and he owns a set of specialized skills such as creating synthesizable RTL code.
- Whereas the verification engineer has to create scenarios where the design does not match its description. He must have skills in finding bugs in the design.
- Your testbench code is in a separate block from design code. In classic Verilog, each goes in a separate module.
- Using a module to hold the testbench often causes timing problems.
- As designs grow in complexity, the connections between the blocks increase. Two RTL blocks may share dozens of signals, which must be listed in the correct order for them to communicate properly. One mismatched or misplaced connection and the design will not work. You can reduce errors by using the connect-by-name syntax, but this more than doubles your typing burden.
- Again, one wrong connection at any level and the design stops working.
- The solution is the ***interface***, the System Verilog construct that represents a bundle of wires, with intelligence such as synchronization, and functional code.
- In the next, we will explain interfaces.

The *interface* construct

9

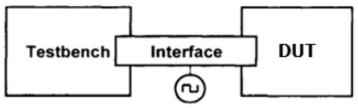
- Interfaces can be thought of as an intelligent bundle of wires.
- They contain the *connectivity*, *synchronization*, and optionally, the *functionality of the communication* between two or more blocks.
- They connect design blocks and testbenches.



- Designs have become so complex that even the communication between blocks may need to be separated out into separate entities.
- To model this, SystemVerilog uses the *interface* construct that you can think of as an *intelligent bundle* of wires.
- They contain the *connectivity*, *synchronization*, and optionally, *the functionality of the communication* between two or more blocks.
- They connect design blocks and testbenches.
- The first improvement to the counter example is to bundle the wires together into an interface. The figure shows the testbench and the counter, communicating using an interface. Note how the interface extends into the two blocks, representing the drivers and receivers that are functionally part of both the testbench and the DUT. The clock can be part of the interface or a separate port.

Communication with interfaces

10




```
1  interface count_ifc (input bit CLK);
2    logic [3:0] Q,P;
3    logic Load, Enable, MR;
4  endinterface
```

Interface

```
6  module test(count_ifc x);
7    initial begin
8      x.P <= 4'b0111;
9      x.MR <= 1'b0;
10     x.Enable <= 1'b1;
11     x.Load <= 1'b0;
12     #3 x.MR <= 1'b1;
13     #6 x.MR <= 1'b0;
14     #43 x.Enable <= 1'b0;
15     #15 x.Enable <= 1'b1;
16     #16 x.Load <= 1'b1;
17     #9 x.Load <= 1'b0;
18   end
19 endmodule
```

Testbench

```
2  module decade_counter (count_ifc y);
3    always @(y.MR) begin
4      if (y.MR)
5        y.Q <= 4'b0000;
6    end
7
8    always @(posedge y.CLK) begin
9      if (!y.MR)
10       if (y.Load)
11         y.Q <= y.P;
12     else if(y.Enable)
13         y.Q <= (y.Q+1) % 10;
14   end
15 endmodule
```

Design

```
18 module top;
19   bit clk;
20   always #5 clk <= ~clk;
21   count_ifc ifc(clk);
22   decade_counter u1(ifc);
23   test u2(ifc);
24
25   initial begin
26     $dumpfile("counter.vcd");
27     $dumpvars;
28     #200 $finish;
29   end
30 endmodule
```

Top module

- The first improvement to the counter example is to bundle the wires together into an interface.
- Here, the testbench and DUT, communicate using an interface.
- The clock can be part of the interface or a separate port.
- The interface instance name, (e.g *ifc* in the above code) should be kept as short as possible as you are going to type it a lot in the design and testbench.
- You might even consider using a single character, as long as this is not ambiguous.
- You can see an immediate benefit, even on this small device: the connections become cleaner and less prone to mistakes.
- If you wanted to put a new signal in an interface. you would just have to add it to the interface definition and the modules that actually used it.
- You would not have to change any module such as top that just pass the interface through.
- This language feature greatly reduces the chance for wiring errors.
- Make sure you declare your interfaces outside of modules and program blocks. If you forget, expect all sorts of trouble.

Modports

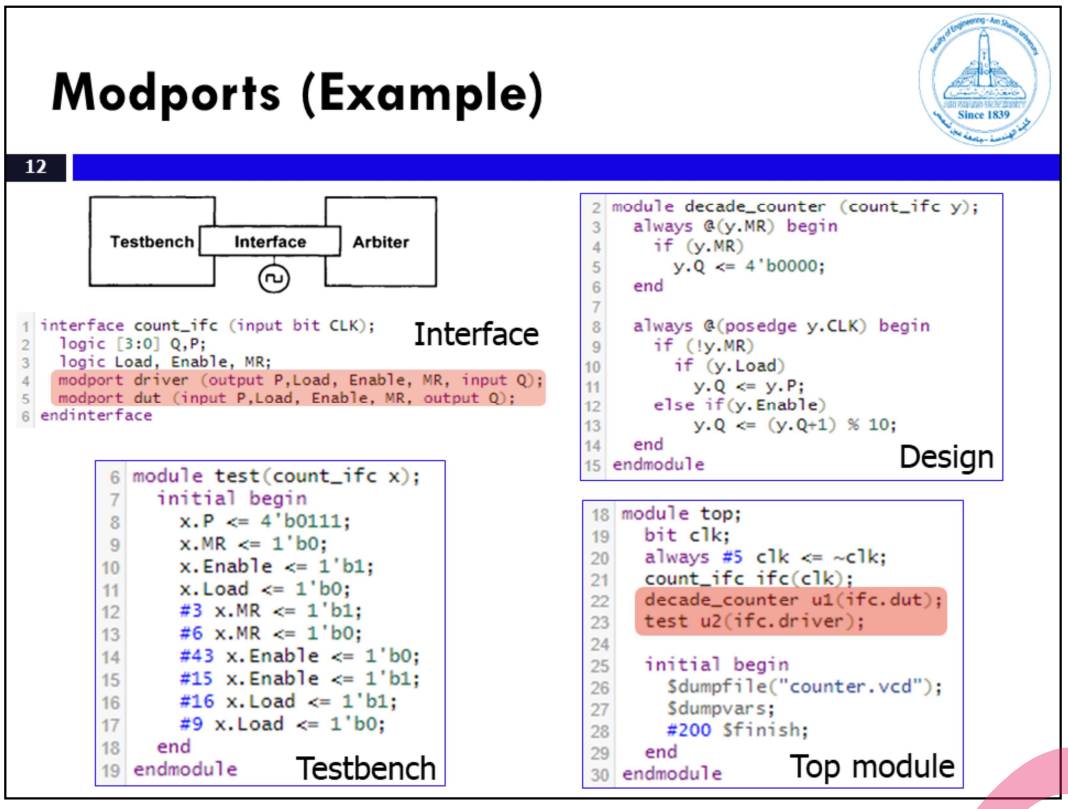
11

```
interface count_ifc (input bit CLK);
  logic [3:0] Q,P;
  logic Load, Enable, MR;

  // modport declaration
  modport driver (output P,Load, Enable, MR, input Q);
  modport dut (input P,Load, Enable, MR, output Q);

endinterface
```

- **The *modport* groups and specifies the *port directions* to the signals declared within the interface.**
- **modports are declared inside the interface with the keyword modport.**
- **The Interface can have any number of modports, the signals declared in the interface can be grouped in many modports**
- **Modpports can have: input, inout, output, and ref**

- The *modport* groups and specifies the *port directions* to the signals declared within the interface.
- **modports** are declared inside the interface with the keyword **modport**.
- By specifying the port directions, **modport** provides access restrictions.
- The keyword **modport** indicates that the directions are declared (as if inside the module).
- **modport** item declared with input is not allowed to be driven or being assigned. Any attempt to drive leads to a compilation error.
- The Interface can have any number of modports, the signals declared in the interface can be grouped in many modports
- Modports can have, input, inout, output, and ref

Modports (Example)




```
1  interface count_ifc (input bit CLK);
2    logic [3:0] Q,P;
3    logic Load, Enable, MR;
4    modport driver (output P,Load, Enable, MR, input Q);
5    modport dut (input P,Load, Enable, MR, output Q);
6  endinterface
```

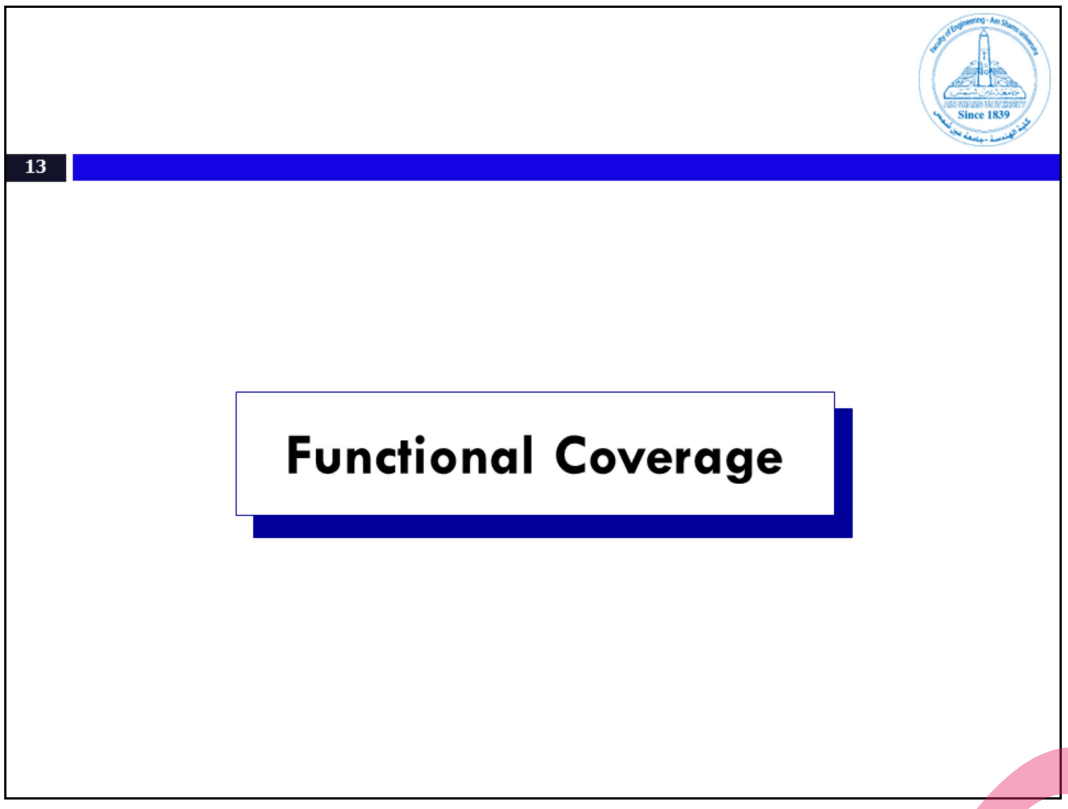
```
2  module decade_counter (count_ifc y);
3    always @(y.MR) begin
4      if (y.MR)
5        y.Q <= 4'b0000;
6    end
7
8    always @(posedge y.CLK) begin
9      if (!y.MR)
10       if (y.Load)
11         y.Q <= y.P;
12     else if(y.Enable)
13         y.Q <= (y.Q+1) % 10;
14   end
15 endmodule
```

```
6  module test(count_ifc x);
7    initial begin
8      x.P <= 4'b0111;
9      x.MR <= 1'b0;
10     x.Enable <= 1'b1;
11     x.Load <= 1'b0;
12     #3 x.MR <= 1'b1;
13     #6 x.MR <= 1'b0;
14     #43 x.Enable <= 1'b0;
15     #15 x.Enable <= 1'b1;
16     #16 x.Load <= 1'b1;
17     #9 x.Load <= 1'b0;
18   end
19 endmodule
```

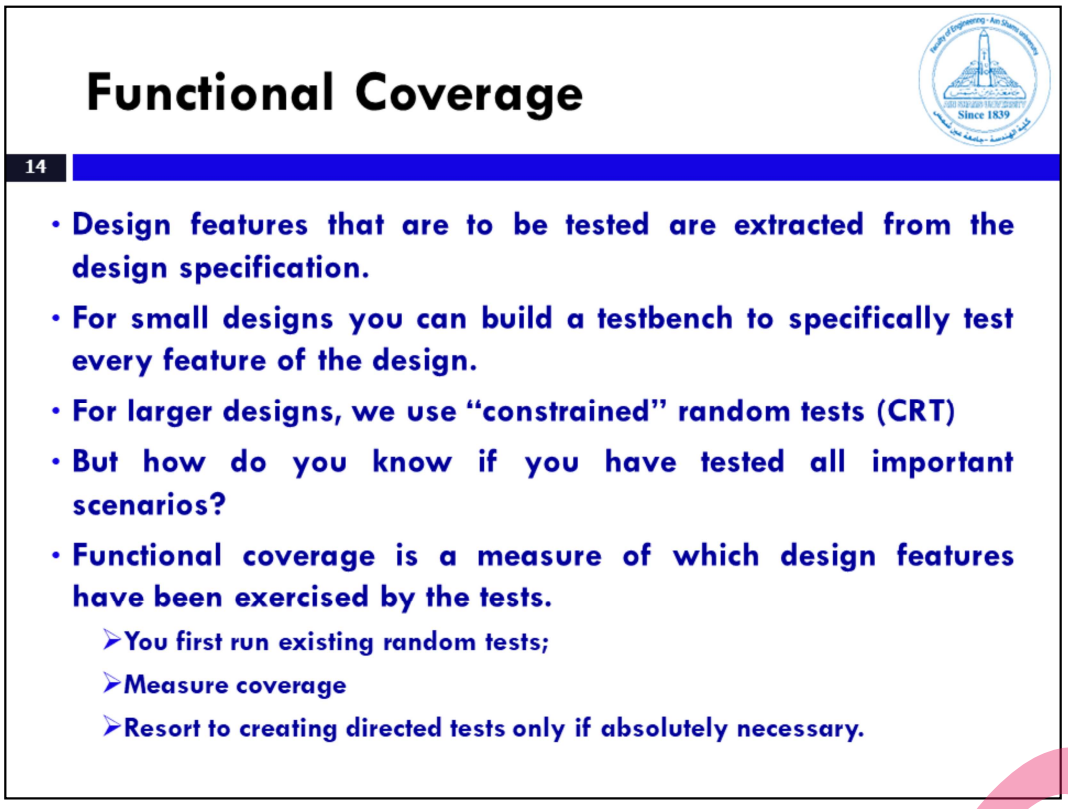
```
18 module top;
19   bit clk;
20   always #5 clk <= ~clk;
21   count_ifc ifc(clk);
22   decade_counter u1(ifc.dut);
23   test u2(ifc.driver);
24
25   initial begin
26     $dumpfile("counter.vcd");
27     $dumpvars;
28     #200 $finish;
29   end
30 endmodule
```

- Note that when you make an instance from the DUT or the testbench, you select the convenient modport, by the following syntax:

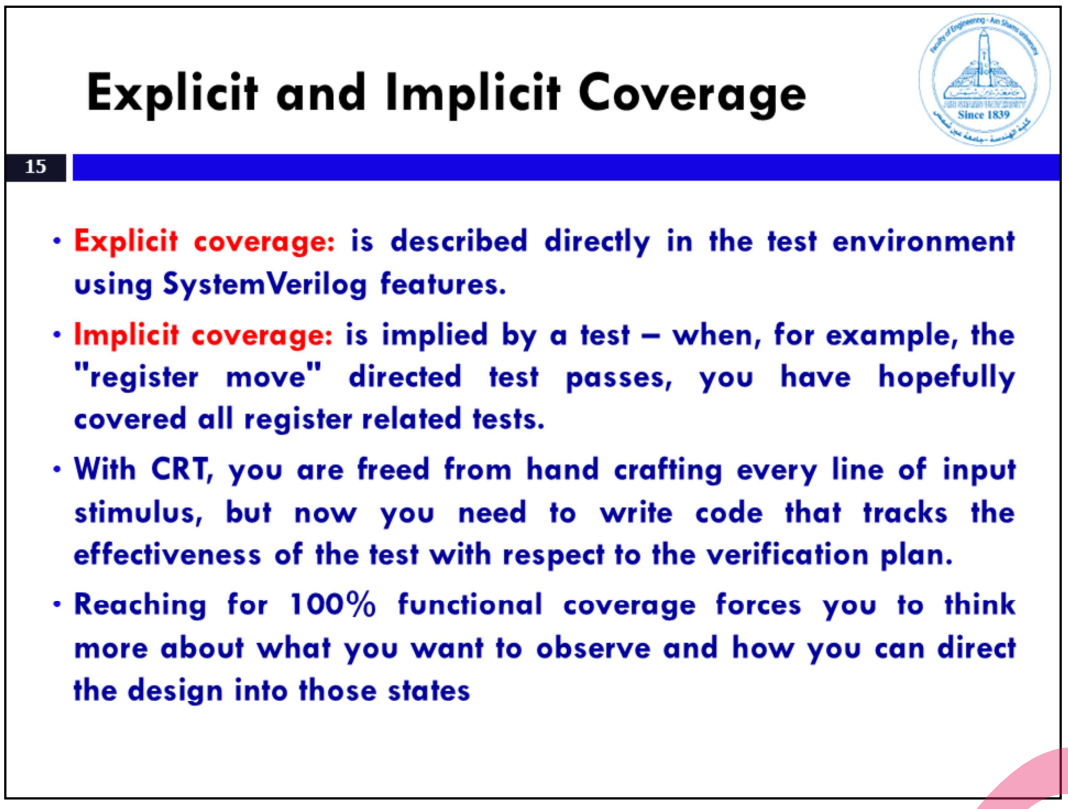
  interface_name.modport_name



- Ver



- Start with the design specification and create a verification plan with a detailed list of what to test and how.
- For small designs you can build a testbench to specifically test every aspect of the design.
- As designs become more complex, the only effective way to verify them thoroughly is with constrained-random testing (CRT). This approach elevates you above the tedium of writing individual directed tests, one for each feature in the design.
- However, if your testbench is taking a random walk through the space of all design states, how do you know if you have tested all important scenarios?
- Whether you are using random or directed stimulus, you can measure progress using *coverage.*
- Functional coverage is a measure of which design features have been exercised by the tests.
- Analyze the coverage results and decide on which actions to take in order to converge on 100% coverage.
  - ➢ Your first choice is to run existing tests with more seeds;
  - ➢ The second choice is to build new constraints.
  - ➢ Resort to creating directed tests only if absolutely necessary.

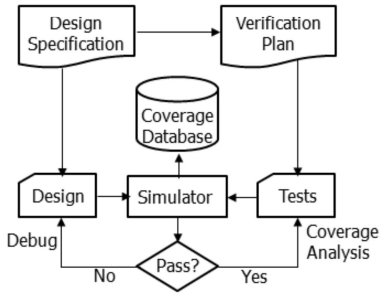


- When you exclusively write directed tests, the verification planning is limited. If the design specification has 100 features, all you have to do is to write 100 tests.
- In this case, coverage is implicit in the tests. Measuring progress is easy: if you complete 50 tests, you are halfway done.
- **Explicit coverage:** is described directly in the test environment using SystemVerilog features.
- **Implicit coverage:** is implied by a test – when, for example, the "register move" directed test passes, you have *hopefully* covered all register related tests.
- With CRT, you are freed from hand crafting every line of input stimulus, but now you need to write code that tracks the effectiveness of the test with respect to the verification plan.
- You are still more productive, as you are working at a higher level of abstraction. You have moved from tweaking individual bits to describing the interesting design states.
- Reaching for 100% functional coverage forces you to think more about what you want to observe and how you can direct the design into those states

Gathering Coverage Data

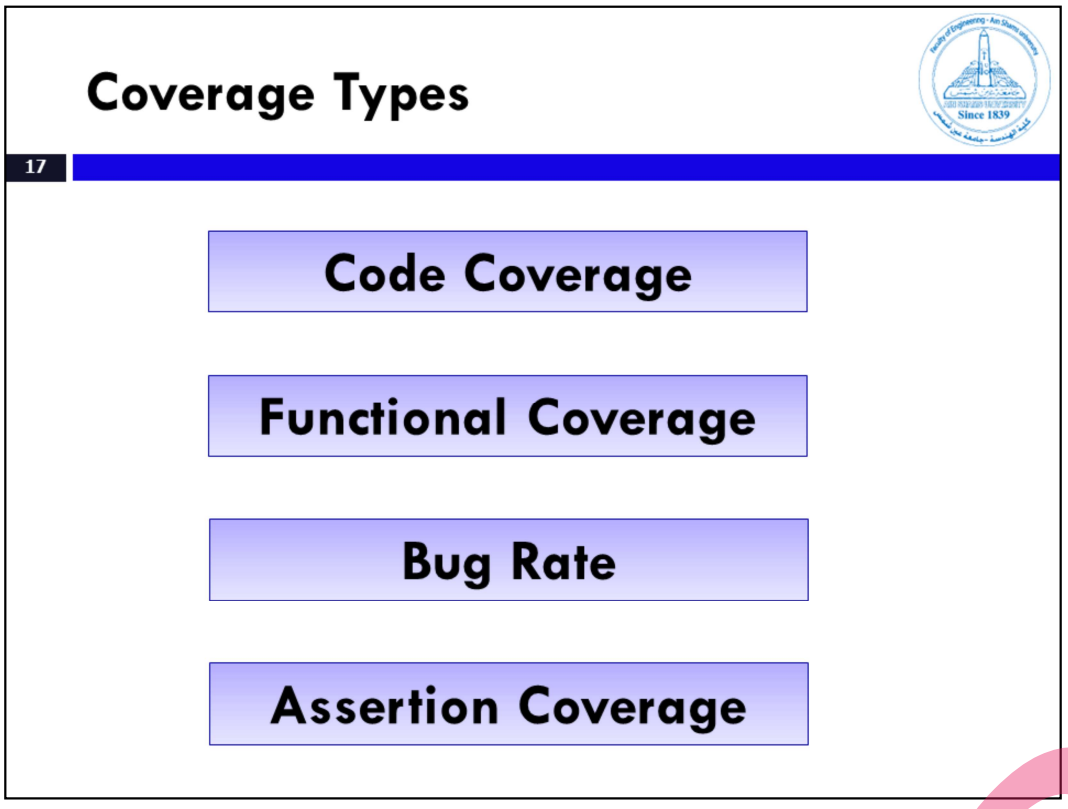
16

- Run the same random testbench over and over, with different seeds, to generate new stimulus.
- Gather the coverage information from all the runs in a coverage database to measure the overall progress.



- If the coverage levels are growing, you need to run the testbench with new seeds, or just run longer tests.
- If the coverage growth slows, add additional constraints to generate more "interesting" stimuli.
- When the coverage is not increasing, some parts of the design are not exercised, so you need to create more directed tests.

- You can run the same random testbench over and over, simply by changing the random seed, to generate new stimulus.
- Each individual simulation generates a database of functional coverage information. You can then merge all this information together to measure your overall progress using functional coverage. You then analyze the coverage data to decide how to modify your tests.
- If the coverage levels are steadily growing. you may just need to run existing tests with new random seeds, or even just run longer tests.
- If the coverage growth starts to slow, you can add additional constraints to generate more "interesting" stimuli.
- When the coverage is not increasing any more, some parts of the design are not being exercised, and so you need to create more directed tests.



- Coverage is a generic term for measuring progress to complete design verification.
- Your simulations slowly paint the canvas of the design, as you try to cover all of the legal combinations.
- The coverage tools gather information during a simulation and then postprocess it to produce a coverage report.
- You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes.
- This iterative process continues until you are satisfied with the coverage level.
- There are 4 things to consider when studying the coverage:
    1. Code Coverage
    2. Functional Coverage
    3. Bug Rate
    4. Assertion Coverage

Code Coverage

18

- **In code coverage you are measuring:**
  - **Line coverage: How many lines of code have been executed.**
  - **Path coverage: Which paths through the code have been executed**
  - **Toggle coverage: Which single bit variables have had the values 0 or 1.**
  - **FSM coverage: Which states and transitions in a FSM have been visited**
- **Simulators include a code coverage tool. You don't have to write extra HDL code.**
- **You are concerned with analyzing the design, not the testbench.**
- **Just because your tests have reached 100% code coverage, your job is not done.**

We have 100% coverage but there is a mistake. The reset logic was accidently left out.

```
module dff(output logic q, q_l,
           input logic clk, d, reset);

  always @(posedge clk or negedge reset) begin
    q <= d;
    q_l <= !d;
  end
endmodule
```

- The easiest way to measure verification progress is with code coverage.
- Here you are measuring:
  - How many lines of code (of the design) have been executed (line coverage).
  - Which paths through the code and expressions have been executed (path coverage).
  - Which single bit variables have had the values 0 or 1 (toggle coverage).
  - Which states and transitions in a state machine have been visited (FSM coverage).
- You don't have to write any extra HDL code. Most simulators include a code coverage tool that does that automatically by analyzing the source code to gather statistics. You then run all your tests, and the code coverage tool creates a database.
- A postprocessing tool converts the database into a readable form. The end result is a measure of how much your tests exercise the design code.
- Note that you are primarily concerned with analyzing *the design code, not the testbench.*
- Untested design code could conceal a hardware bug, or may be just a redundant code.
- Code coverage measures how thoroughly your tests exercised the "implementation" of the design specification, and not the verification plan.
- Just because your tests have reached 100% code coverage, your job is not done. What if you made a mistake that your test didn't catch? Worse yet, what if your implementation is missing a feature?
- The shown module is for a D-flip flop. Can you see the mistake?
- The reset logic was accidently left out. A code coverage tool would report that every line had been exercised, yet the model was not implemented correctly.

Functional Coverage

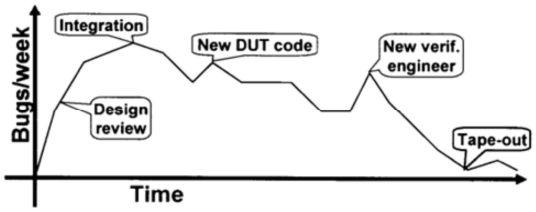
19

- The verification plan lists how the functionality of a design is to be stimulated, verified, and measured.
- When you gather measurements on what functions were covered, you are performing “functional" coverage.
- Functional coverage is tied to the design intent and is sometimes called specification coverage, While code coverage measures the design implementation.
- If a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

- The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, cell phone, or anything else.
- The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured.
- When you gather measurements on what *functionalities* were covered, you are performing “functional" coverage.
- For example, the verification plan for a D-flip flop would mention not only its data storage but also how it resets to a known state. Until your test checks both these design features, you will not have 100% functional coverage.
- Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation. Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

Bug Rate

20



- You should keep track of how many bugs you found each week, over the life of a project.
- The bug rate approaches zero, as the design nears tape-out.
- However, you are not yet done. Every time the rate goes down. it is time to find different ways to create corner cases.
- The bug rate can vary per week based on many factors such as project phases, recent design changes, blocks being integrated, personnel changes, and even vacation schedules.

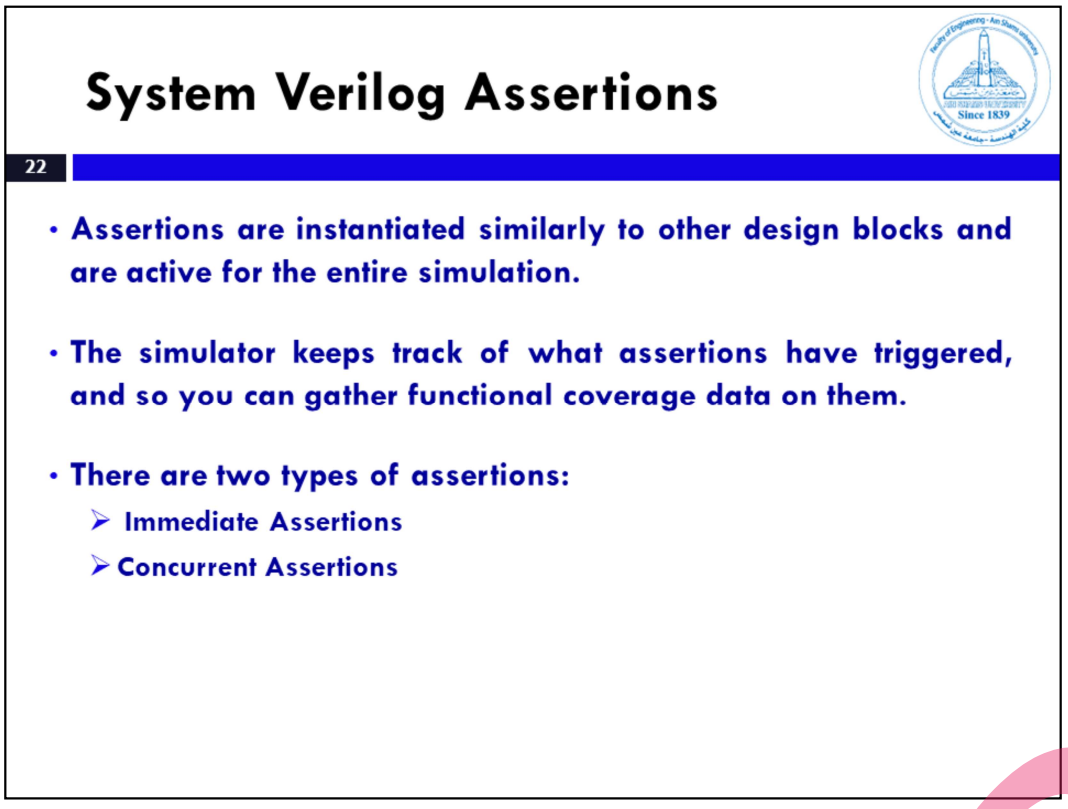
- An indirect way to measure coverage is to look at the rate at which *fresh bugs* are found.
- You should keep track of how many bugs you found each week, over the life of a project.
  - At the start, you may find many bugs through inspection as you create the testbench.
  - As you read the design specs, you may find inconsistencies, which hopefully are fixed before the RTL is written.
  - Once the testbench is up and running, a torrent of bugs is found as you check each module in the system.
  - The bug rate drops, hopefully to zero, as the design nears tape-out. However, you are not yet done. Every time the rate goes down. it is time to find different ways to create *corner cases*. (by corner case we mean values at, and around, the extreme values intended by the design)
- The bug rate can vary per week based on many factors such as *project phases, recent design changes*, *blocks being integrated*, *personnel changes*, and even *vacation schedules*. Unexpected changes in the rate could signal a potential problem. As shown in the figure, it is not uncommon to keep finding bugs even after tape-out and even after the design ships to customers.

Assertion Coverage

21

- The Assertions are pieces of declarative code that check the relationships between design signals, either *once* or *over a period of time*.
- The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant.
- Some assertions might look for interesting signal values or design states.
- You can measure how often these assertions are triggered during a test by using assertion coverage.

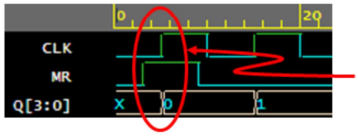
- Assertions are pieces of declarative code that check the relationships between design signals, either *once* or over *a period of time*.
- The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem.
- Assertions can also check arbitration algorithms, FIFOs, and other hardware. These are coded with the *assert property* statement.
- Some assertions might look for interesting signal values or design states, such as a successful bus transaction. These are coded with the *cover property* statement.
- You can measure how often these assertions are triggered during a test by using *assertion coverage*.
- A *cover property* observes sequences of signals, whereas a *cover group* (explained later) samples data values and transactions during the simulation.
- These two constructs overlap in that a cover group can trigger when a sequence completes. Additionally, a sequence can collect information that can be used by a cover group.



- You can create temporal assertions about signals in your design using SystemVerilog Assertions (SVA).
- Assertions are instantiated similarly to other design blocks and are active for the entire simulation.
- The simulator keeps track of what assertions have triggered, and so you can gather functional coverage data on them.
- There are two types of assertions: Immediate Assertions, and Concurrent Assertions. These two types will be explained in the following slides.

Counter with an Error

23



Synchronous not Asynchronous

```
1  interface count_ifc (input bit CLK);
2    logic [3:0] Q,P;
3    logic Load, Enable, MR;
4    modport driver (output P,Load, Enable, MR, input Q);
5    modport dut (input P,Load, Enable, MR, output Q);
6  endinterface
```

Interface

```
6  module test(count_ifc x);
7    initial begin
8      x.P <= 4'b0111;
9      x.MR <= 1'b0;
10     x.Enable <= 1'b1;
11     x.Load <= 1'b0;
12     #3 x.MR <= 1'b1;
13     #6 x.MR <= 1'b0;
14     #43 x.Enable <= 1'b0;
15     #15 x.Enable <= 1'b1;
16     #16 x.Load <= 1'b1;
17     #9 x.Load <= 1'b0;
18   end
19 endmodule
```

Testbench

```
1  module decade_counter (count_ifc y);
2  //  always @(y.MR) begin
3  //    if (y.MR)
4  //      y.Q <= 4'b0000;
5  //  end
6
7    always @(posedge y.CLK) begin
8      if (y.MR)
9        y.Q <= 4'b0000;
10     else if(y.Load)
11       y.Q <= y.P;
12     else if(y.Enable)
13         y.Q <= (y.Q+1) % 10;
14   end
15 endmodule
```

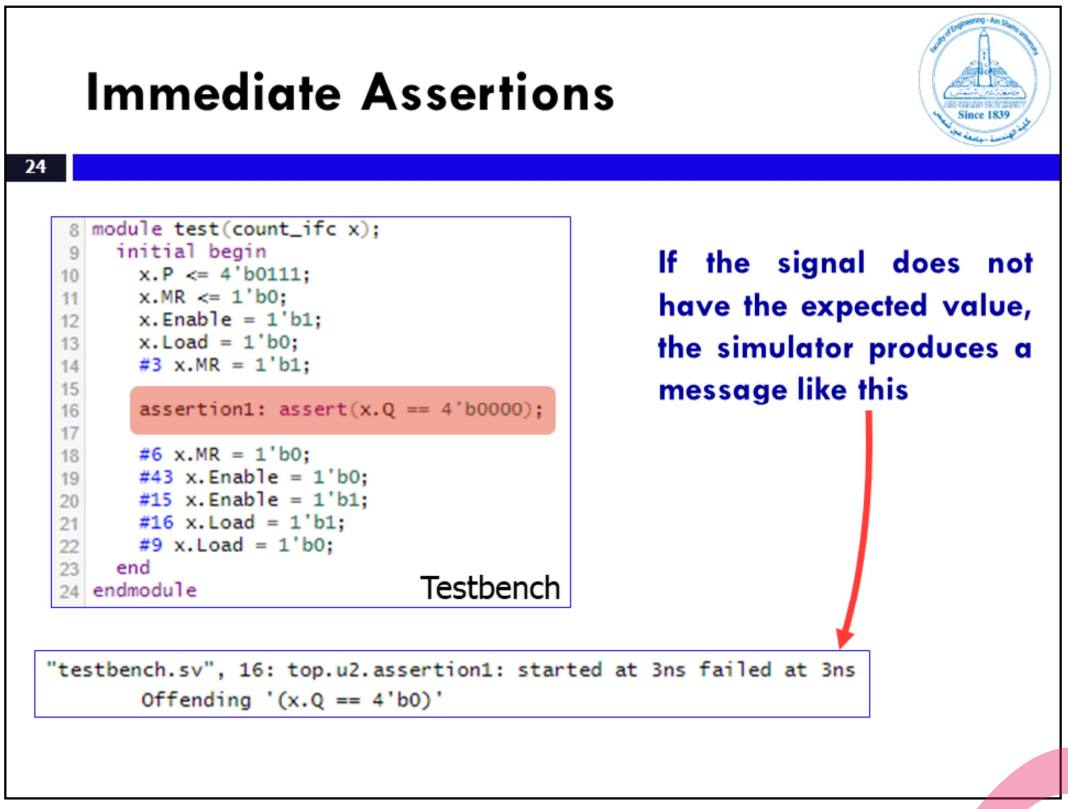
Design

```
18 module top;
19   bit clk;
20   always #5 clk <= ~clk;
21   count_ifc ifc(clk);
22   decade_counter u1(ifc.dut);
23   test u2(ifc.driver);
24
25   initial begin
26     $dumpfile("counter.vcd");
27     $dumpvars;
28     #200 $finish;
29   end
30 endmodule
```

Top module

- Here, the designer mistakenly made the Master Reset Synchronous.
- As shown in the waveform, resetting the output is synchronized with the edge of the clock. The specifications states that the reset is an asynchronous reset.
- How can assertions help revealing this error?

Immediate Assertions



- The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code.
- Immediate assertions are simple non-temporal domain assertions that are executed like statements in a procedural block.
- The assertion condition is non-temporal, which means its execution computes and reports the assertion results at the same time.
- If the expression evaluates to X, Z or 0, then it is interpreted as being false and the assertion is said to fail.
- Otherwise, the expression is interpreted as being true and the assertion is said to pass.
- In the above code we are using the assertion to test whether the output Q is reset immediately after MR is activated.
- The message says: on line 16 of the file testbench.sv, the assertion *top.u2.assertion1* started at 3 ns to check the signal *x.Q*, failed.

Customizing the Assertion Actions

25

```
8  module test(count_ifc x);
9    initial begin
10     x.P <= 4'b0111;
11     x.MR <= 1'b0;
12     x.Enable = 1'b1;
13     x.Load = 1'b0;
14     #3 x.MR = 1'b1;
15
16     assertion1: assert(x.Q == 4'b0000)
17       $display("The output value is correct");
18     else
19       $fatal("The reset is not synchronous");
20
21     #6 x.MR = 1'b0;
22     #43 x.Enable = 1'b0;
23     #15 x.Enable = 1'b1;
24     #16 x.Load = 1'b1;
25     #9 x.Load = 1'b0;
26   end
27 endmodule
```

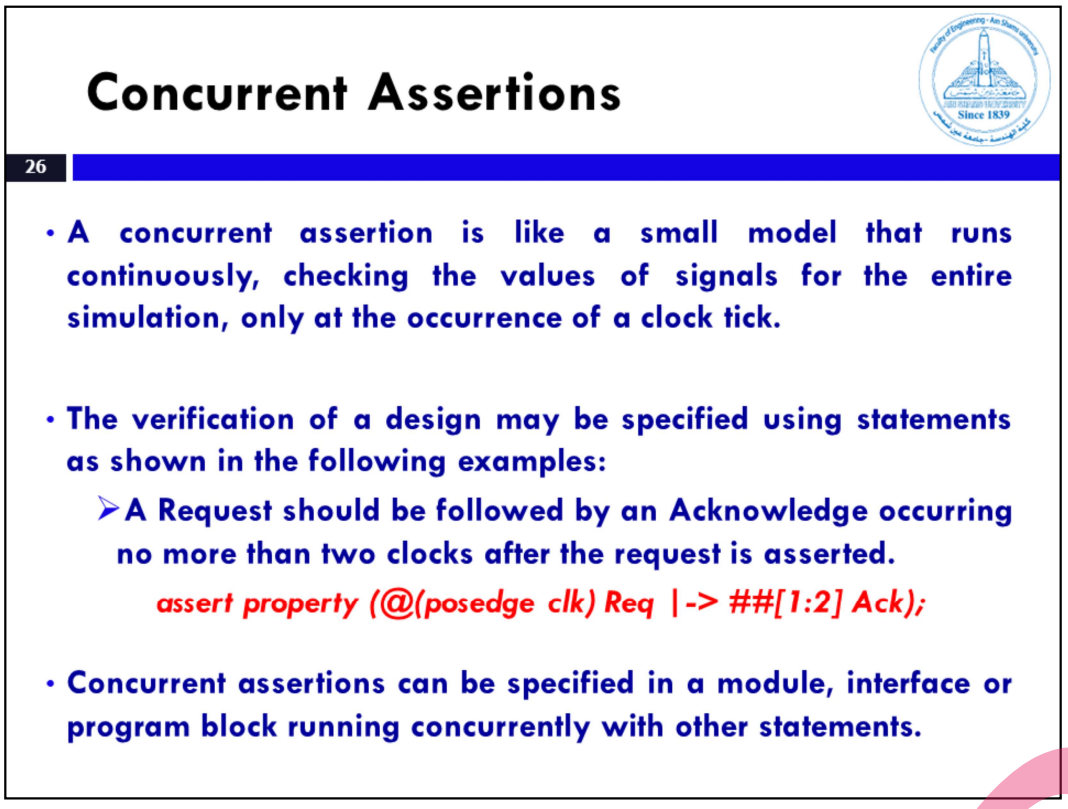
If the assertion fails:

```
"testbench.sv", 16: top.u2.assertion1: started at 3ns failed at 3ns
        Offending '(x.Q == 4'b0)'
Fatal: "testbench.sv", 16: top.u2.assertion1: at time 3 ns
The reset is not synchronous
```

If the assertion succeeds:

```
The output value is correct
```

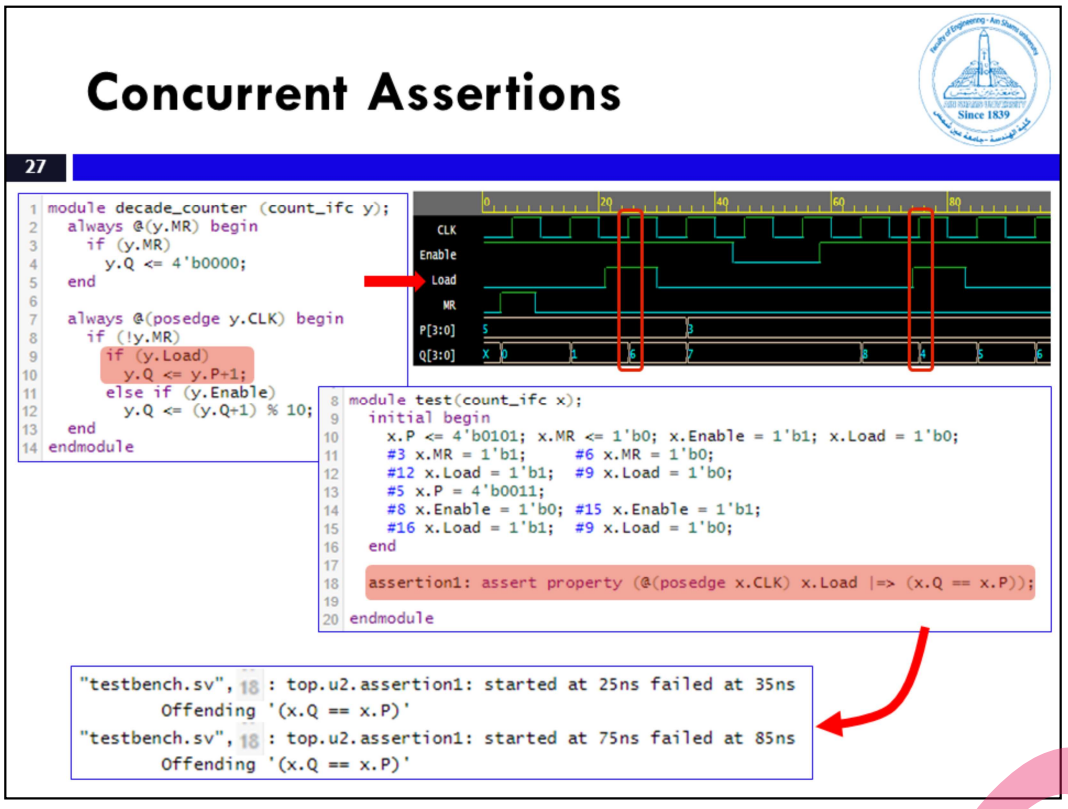
- An immediate assertion has optional *then* and *else* clauses.
- If you want to augment the default message, you can add your own.
- The above example shows how to create a custom error message in an immediate assertion.
- If Q does not have the expected value. you'll see an error message as shown above.
- SystemVerilog has four functions to print messages:
  - $info,
  - $warning,
  - $error
  - $fatal.
- These are allowed only inside an assertion, not in procedural code.
- You can use the then-clause to record when an assertion completed successfully.



- The other type of assertions is the *concurrent assertion* that you can think of as a small model that runs continuously, checking the values of signals for the entire simulation.
- Concurrent assertions are written as follows: *assert property*, unlike immediate assertions which are written as follows: *assert*
- The behaviour of a design may be specified using statements similar to these:
  - A Request should be followed by an Acknowledge occurring no more than two clocks after the request is asserted. You can check that using an assertion like this one:

    assert property (@(posedge clk) Req |-> ##[1:2] Ack);

    We use the implication operator |-> to express some event implies some other thing. **##[1:2]** means after one or two clock cycles



- Here is a simple assertion to check that whenever the load line is active, then at the very next rising edge of the clock the counter is loaded by the values found on the P inputs.
- It is clear that there is an error in the design, as the value loaded is P+1 not P, so the assertion fails.
- It is also clear that, as the assertion is a *concurrent* assertion, it is active during the *whole simulation time*, and whenever it fails an error message is issued.
- In the shown simulation the assertion fails twice. One time at 25 ns, and the second time at 75 ns.
- In the previous slide we used "|->" as an implication operator, whereas in the above example we used "|=>". So is there any difference between them ?

Types of Implication operators

28

- The left-hand side of the implication is called the *antecedent* and the right-hand side is called the *consequent*
- If the antecedent succeeds, then the consequent is evaluated.
- There are two types of implication operators:
  - Overlapped implication (|->)
  - Non-overlapped implication (|=>)
- The implication construct can be *used only with property definitions.*

- The implication is equivalent to an if-then structure.
- The left-hand side of the implication is called the “antecedent” and the right-hand side is called the “consequent.”
- The antecedent is the gating condition. If the antecedent succeeds, then the consequent is evaluated.
- The implication construct can be used only with property definitions. It cannot be used in sequences.
- There are 2 types of implication:
  - Overlapped implication
  - Non-overlapped implication

Overlapped Implication

29

- The overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the same clock cycle*.

```
1 property p;
2   @(posedge clk) a |-> b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should also be high *on the same clock edge*.

- The overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the same clock cycle*.
- The shown property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should also be high *on the same clock edge*.

Non-Overlapped Implication

30

- The non-overlapped implication is denoted by the symbol |=>.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the next clock cycle*.

```
1 property p;
2   @(posedge clk) a |=> b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should also be high *on the next clock edge*.

- The non-overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the next clock cycle*.
- The shown property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should also be high *on the next clock edge*.

Implication with a fixed delay on the consequent

31

- Delays for a certain number of clock cycles is expressed in implications by ## followed by the number of cycles.

```
1 property p;
2   @(posedge clk) a |-> ##2 b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should be high after *exactly 2 clock cycles*.

- If there is a match on the antecedent, then the consequent expression is evaluated *after certain number of cycles* expressed by ## followed by the number of cycles.

Timing windows in SVA checkers

32

- **Delays for a certain *range* of clock cycles is expressed in implications by ## followed by a range of cycles between square brackets.**

```
1 property p;
2   @(posedge clk) a |-> ##[1:4] b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then *within 1 to 4 clock cycles*, the signal *b* should be high. You can write the range as [0:4] if you want *b* to be high in the same cycle, or with 4 clock cycles

- If there is a match on the antecedent, then the consequent expression is evaluated *after certain range of cycles* expressed by ## followed by a range of cycles.
- In the above example, if you want b to be high in the same clock cycle, or withn 4 cycles, we write the range as ##[0:4]

Infinite timing window

33

- The upper limit of the timing window specified in the right-hand side can be defined with a $ sign which implies that there is no upper bound for timing.
- This is called the *eventuality operator*. The checker will keep checking for a match until the end of the simulation.

```
1 property p;
2   @(posedge clk) a |-> ##[1:$] b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* will be high *eventually* starting from the next clock cycle.

- The upper limit of the timing window specified in the right-hand side can be defined with a "$" sign which implies that there is no upper bound for timing.
- This is called the *eventuality* operator. The checker will keep checking for a match until the end of the simulation.
- The shown property checks that, if signal *a* is high on a given positive clock edge, then signal *b* will be high *eventually* starting from the next clock cycle.