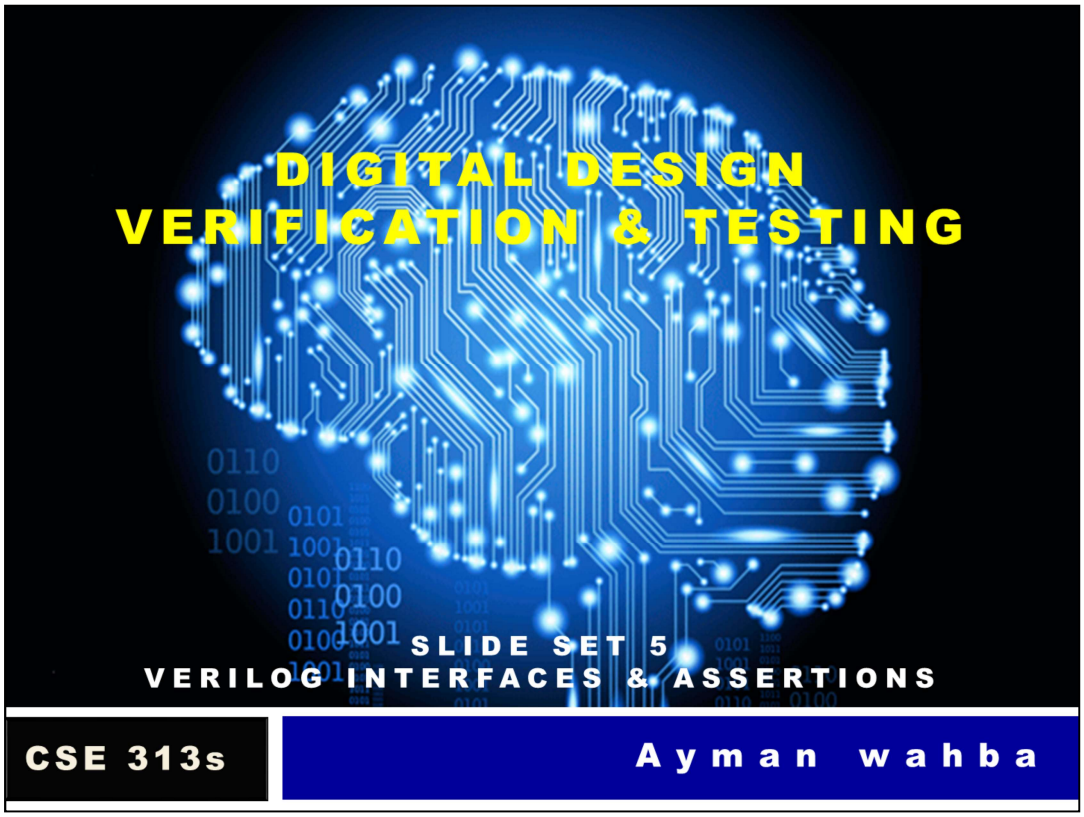
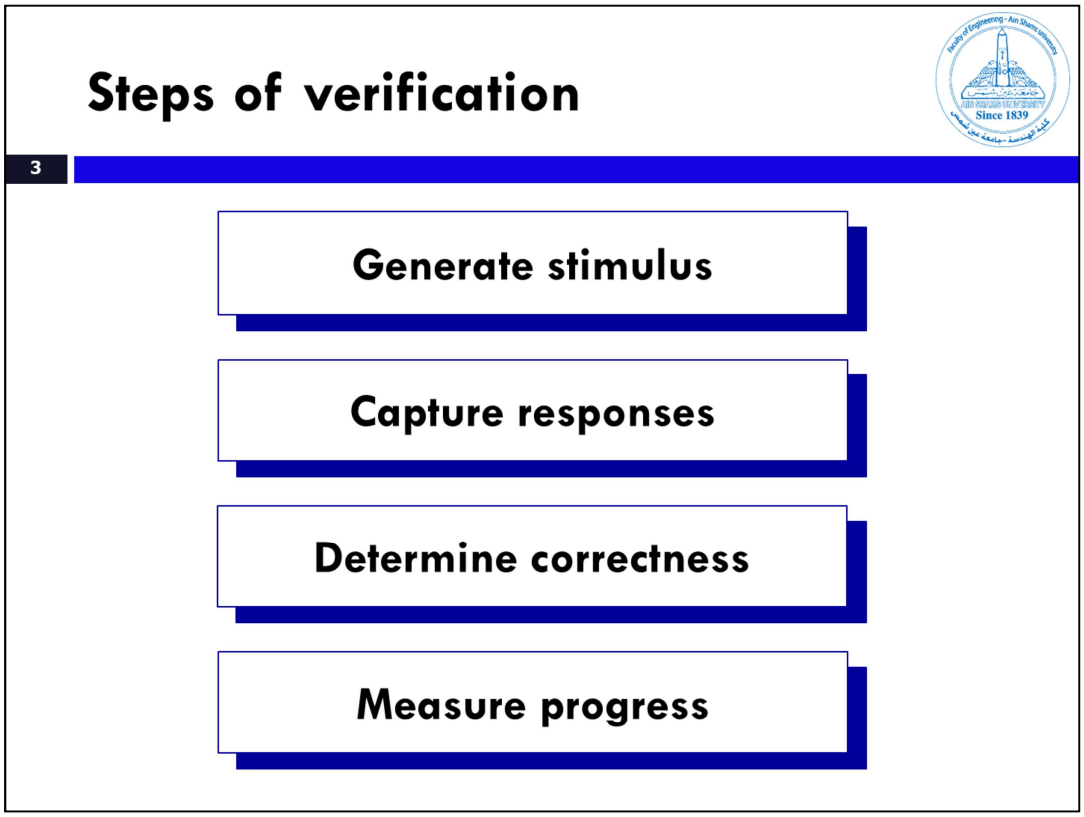

DIGITAL DESIGN
VERIFICATION & TESTING
SLIDE SET 5
VERILOG INTERFACES & ASSERTIONS
CSE 313s
Ayman wahba

Connecting the Testbench and the Design

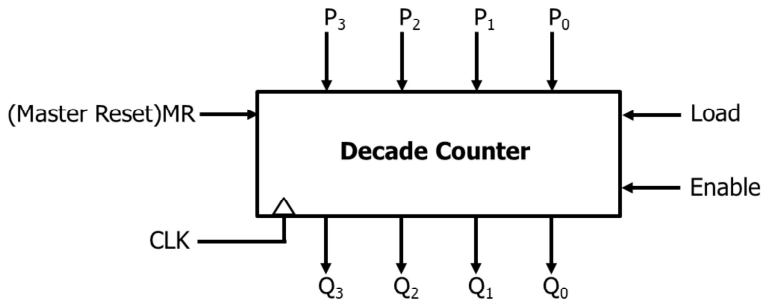


- There are several steps needed to verify a design:
  1. Generate stimulus,
  2. Capture responses,
  3. Determine correctness, and
  4. Measure progress.

Simple example (decade counter)

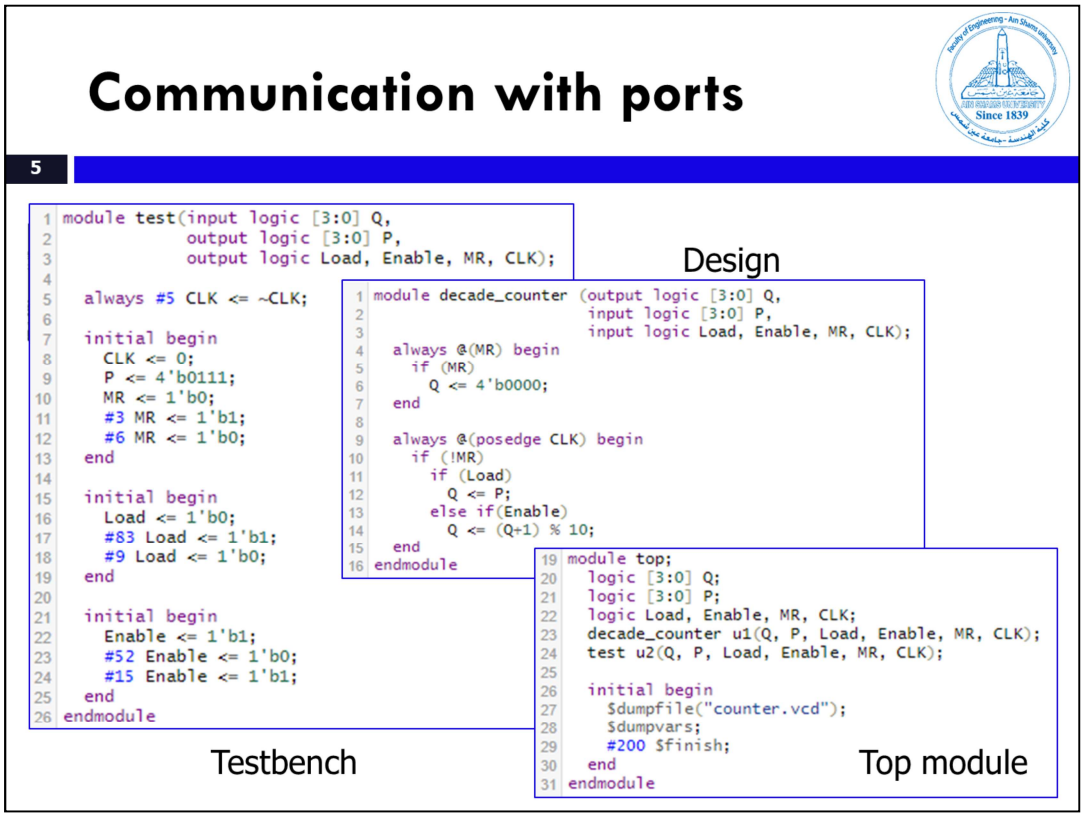
4

- 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 can be loaded by any value P, using the synchronous Load line.

Communication with ports




```
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
```

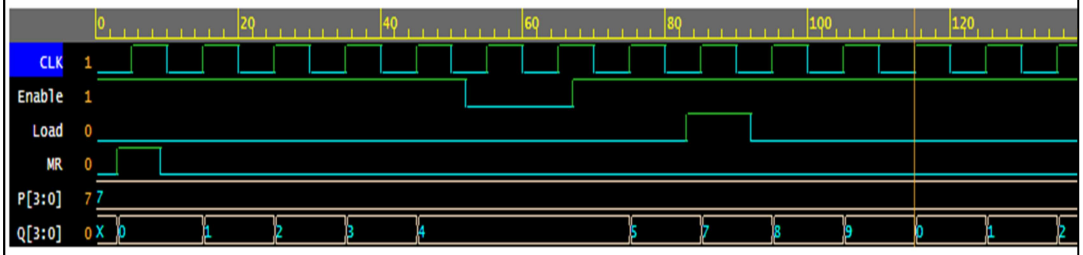
```
module top;
  logic [3:0] Q;
  logic [3:0] P;
  logic Load, Enable, MR, CLK;
  decade_counter u1(Q, P, Load, Enable, MR, CLK);
  test u2(Q, P, Load, Enable, MR, CLK);

  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

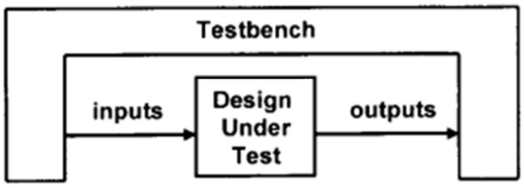


- Using the generated waveforms from simulation, you visually inspect the responses, and determine the correctness of your design.
- Again, this is possible for this simple design, but with large designs, we may need an automated process to make sure of the correctness of all the aspects of the design.
- 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.

Separating the testbench and the Design

8

- Designers create code that meets the specification.
- Verifiers try to find scenarios where the design does not match its description.
- 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 and is error prone.

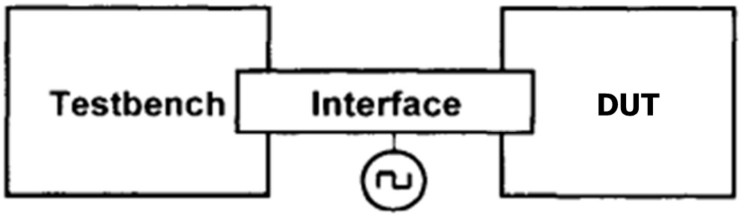
The solution is using System Verilog interfaces

- 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

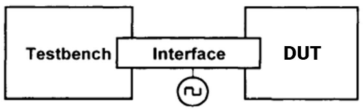
- Interfaces can be thought of as an intelligent bundle of wires.
- They contain the *connectivity*, *synchronization*, and optionally, the *functionality of the communication* between two or more blocks.
- They connect design blocks and testbenches.



- 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.