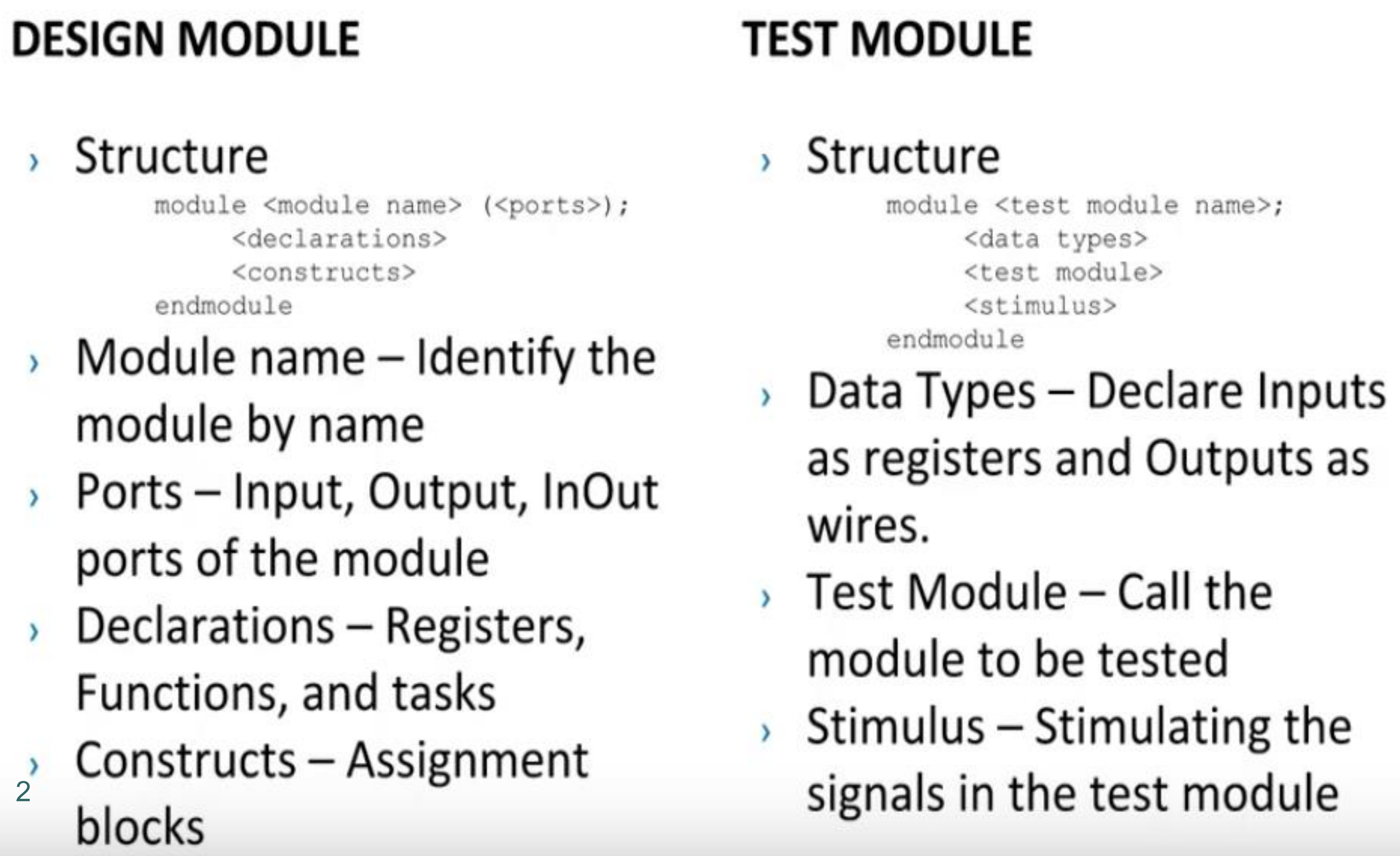
שפת תכנון חומרה
Verilog - ורילוג

Verilog – Testbench

Dr. Avihai Aharon

Testbench



DESIGN MODULE

- Structure

```
module <module name> (<ports>);
     <declarations>
     <constructs>
endmodule
```

- Module name – Identify the module by name
- Ports – Input, Output, InOut ports of the module
- Declarations – Registers, Functions, and tasks
- Constructs – Assignment blocks

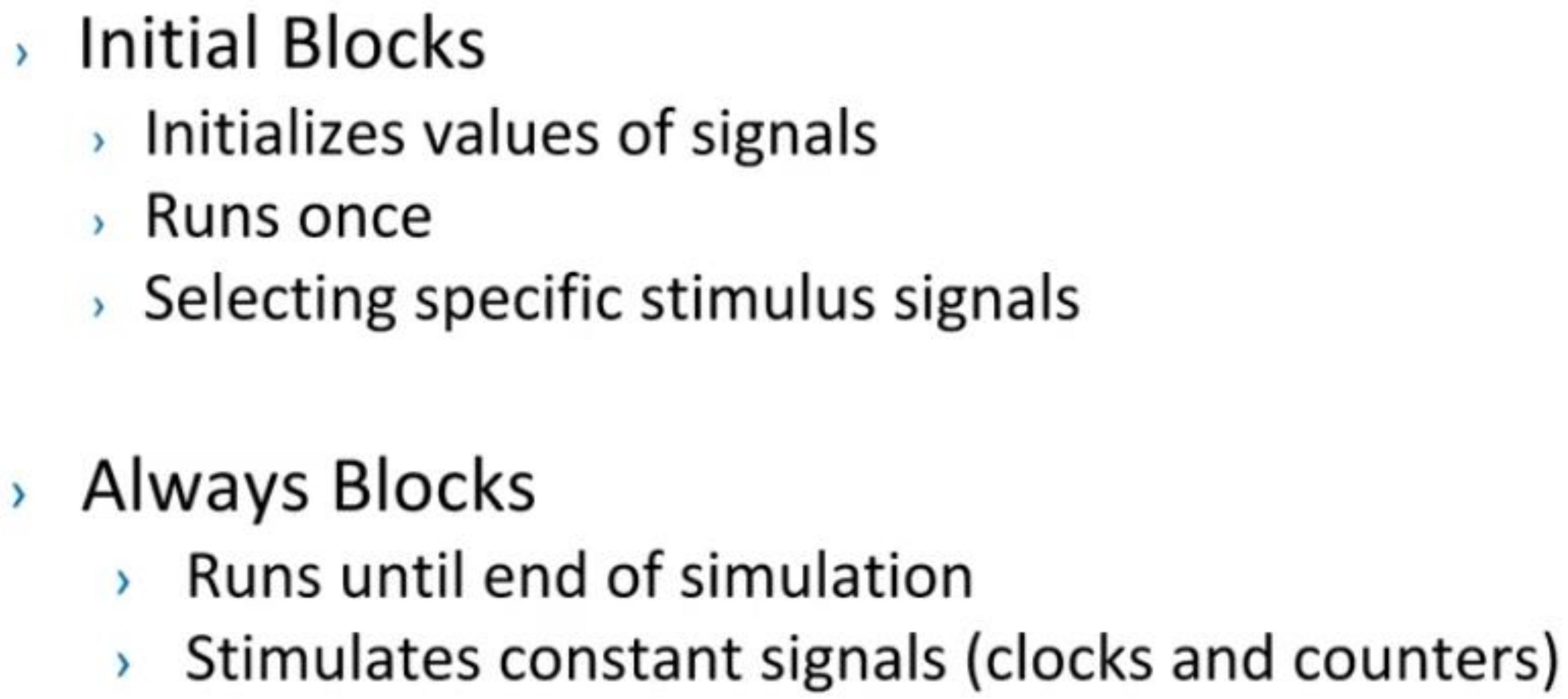
TEST MODULE

- Structure

```
module <test module name>;
     <data types>
     <test module>
     <stimulus>
endmodule
```

- Data Types – Declare Inputs as registers and Outputs as wires.
- Test Module – Call the module to be tested
- Stimulus – Stimulating the signals in the test module

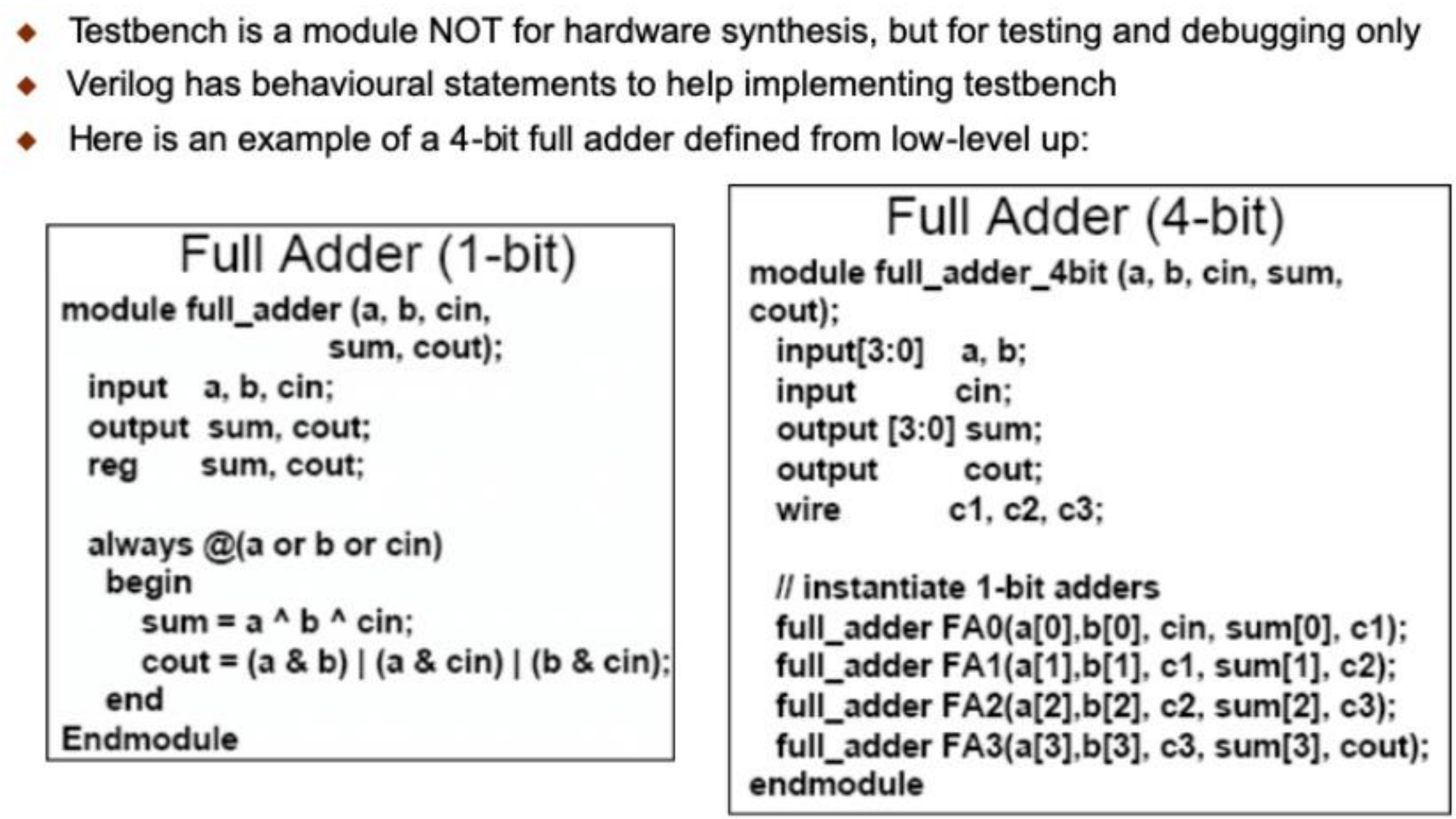
Testbench



- Initial Blocks
  - Initializes values of signals
  - Runs once
  - Selecting specific stimulus signals

- Always Blocks
  - Runs until end of simulation
  - Stimulates constant signals (clocks and counters)

Testbench – 4 bits Full adder



- Testbench is a module NOT for hardware synthesis, but for testing and debugging only
- Verilog has behavioural statements to help implementing testbench
- Here is an example of a 4-bit full adder defined from low-level up:

Full Adder (1-bit)

```
module full_adder (a, b, cin,
                   sum, cout);
  input    a, b, cin;
  output  sum, cout;
  reg       sum, cout;

  always @(a or b or cin)
   begin
     sum = a ^ b ^ cin;
     cout = (a & b) | (a & cin) | (b & cin);
   end
Endmodule
```

Full Adder (4-bit)

```
module full_adder_4bit (a, b, cin, sum,
cout);
  input[3:0]    a, b;
  input           cin;
  output [3:0] sum;
  output          cout;
  wire            c1, c2, c3;

  // instantiate 1-bit adders
  full_adder FA0(a[0],b[0], cin, sum[0], c1);
  full_adder FA1(a[1],b[1], c1, sum[1], c2);
  full_adder FA2(a[2],b[2], c2, sum[2], c3);
  full_adder FA3(a[3],b[3], c3, sum[3], cout);
endmodule
```

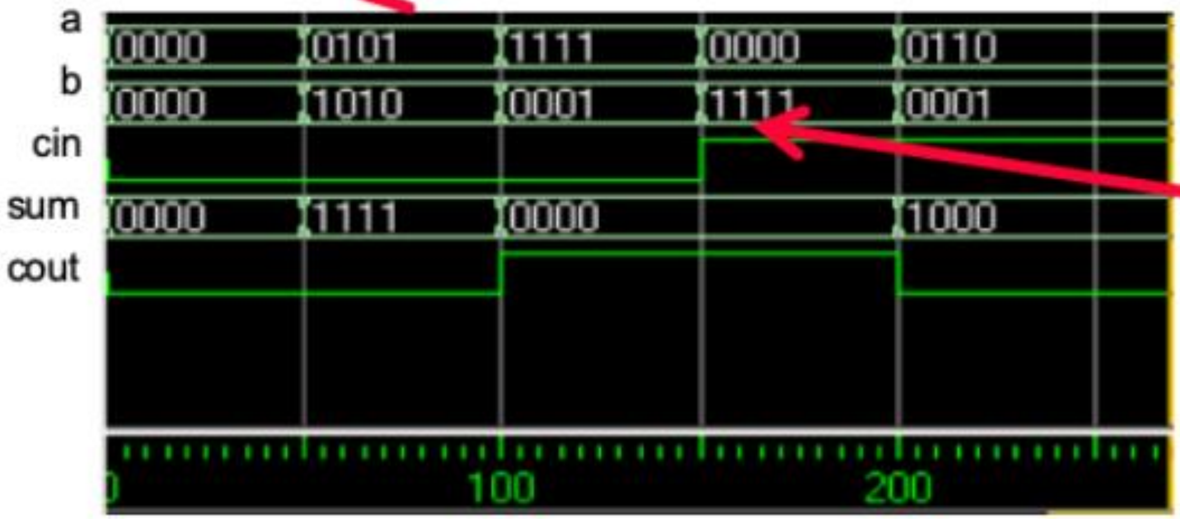
Testbench – 4 bits Full adder

- **Initial** block together with **#<time>** define input vectors at different times to test circuit:

```
module test_adder;
  reg [3:0]  a, b;
  reg        cin;
  wire [3:0] sum;
  wire       cout;

  full_adder_4bit dut(a, b, cin,
                        sum, cout);
```

```
initial
   begin
      a = 4'b0000;
      b = 4'b0000;
      cin = 1'b0;
      #50;
      a = 4'b0101;
      b = 4'b1010;
      // sum = 1111, cout = 0
      #50;
      a = 4'b1111;
      b = 4'b0001;
      // sum = 0000, cout = 1
#50;
      a = 4'b0000;
      b = 4'b1111;
      cin = 1'b1;
      // sum = 0000, cout = 1
      #50;
      a = 4'b0110;
      b = 4'b0001;
      // sum = 1000, cout = 0
   end // initial begin
endmodule // test_adder
```



$finish – ends the simulation

Testbench – sequential

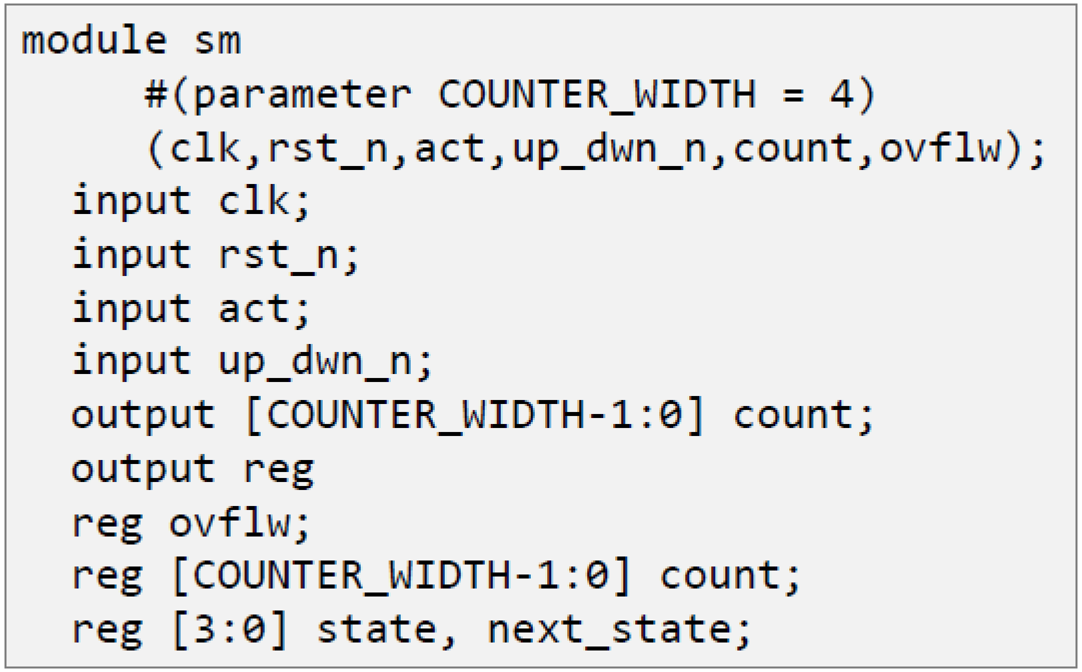
- Create a clock:

```
`define CLK_PER 10

initial
  begin  //begins executing at time 0
    clk = 0;
  end

always   //begins executing at time 0 and never stops
    #(CLK_PER/2) clk = ~clk;
```

Testbench – FSM example



```
module sm
     #(parameter COUNTER_WIDTH = 4)
     (clk,rst_n,act,up_dwn_n,count,ovflw);
  input clk;
  input rst_n;
  input act;
  input up_dwn_n;
  output [COUNTER_WIDTH-1:0] count;
  output reg
  reg ovflw;
  reg [COUNTER_WIDTH-1:0] count;
  reg [3:0] state, next_state;
```

Testbench – FSM example

```
module sm
      #(parameter COUNTER_WIDTH = 4)
      (clk,rst_n,act,up_dwn_n,count,ovflw);
  input clk;
  input rst_n;
  input act;
  input up_dwn_n;
  output [COUNTER_WIDTH-1:0] count;
  output reg
  reg ovflw;
  reg [COUNTER_WIDTH-1:0] count;
  reg [3:0] state, next_state;
```

Testbench Example

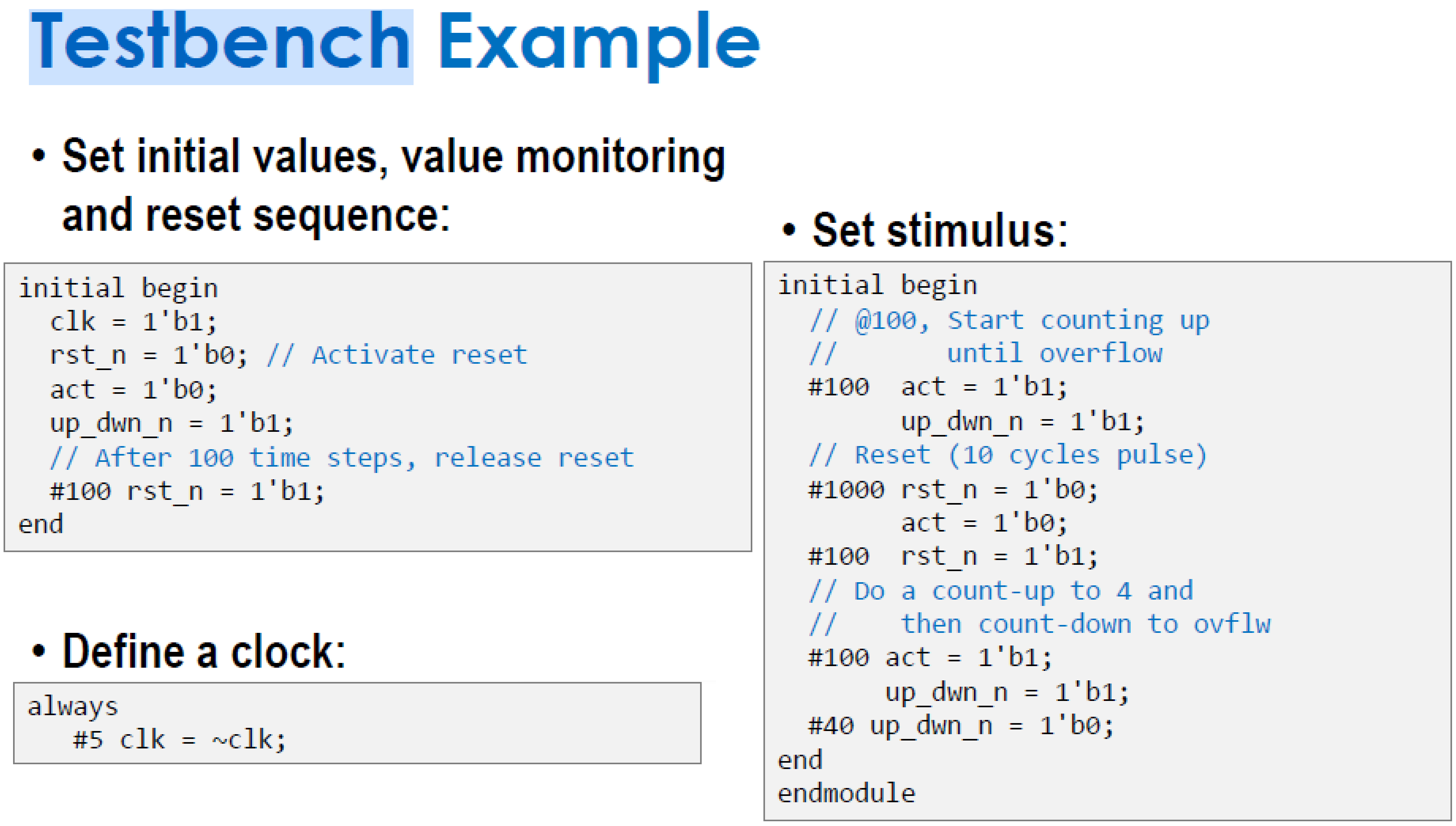
- Definition of signals and parameters

```
module sm_tb;
  parameter WIDTH = 5;
  reg clk;
  reg rst_n;
  reg act;
  reg up_dwn_n;
  wire [WIDTH-1:0] count;
  wire ovflw;
```

- Instantiate the state machine:

```
sm #(WIDTH) DUT (.clk(clk),.rst_n(rst_n),
        .act(act),.up_dwn_n(up_dwn_n),
        .count( count),.ovflw(ovflw));
```

Testbench – FSM example



Testbench Example

- Set initial values, value monitoring and reset sequence:

```
initial begin
  clk = 1'b1;
  rst_n = 1'b0; // Activate reset
  act = 1'b0;
  up_dwn_n = 1'b1;
  // After 100 time steps, release reset
  #100 rst_n = 1'b1;
end
```

- Define a clock:

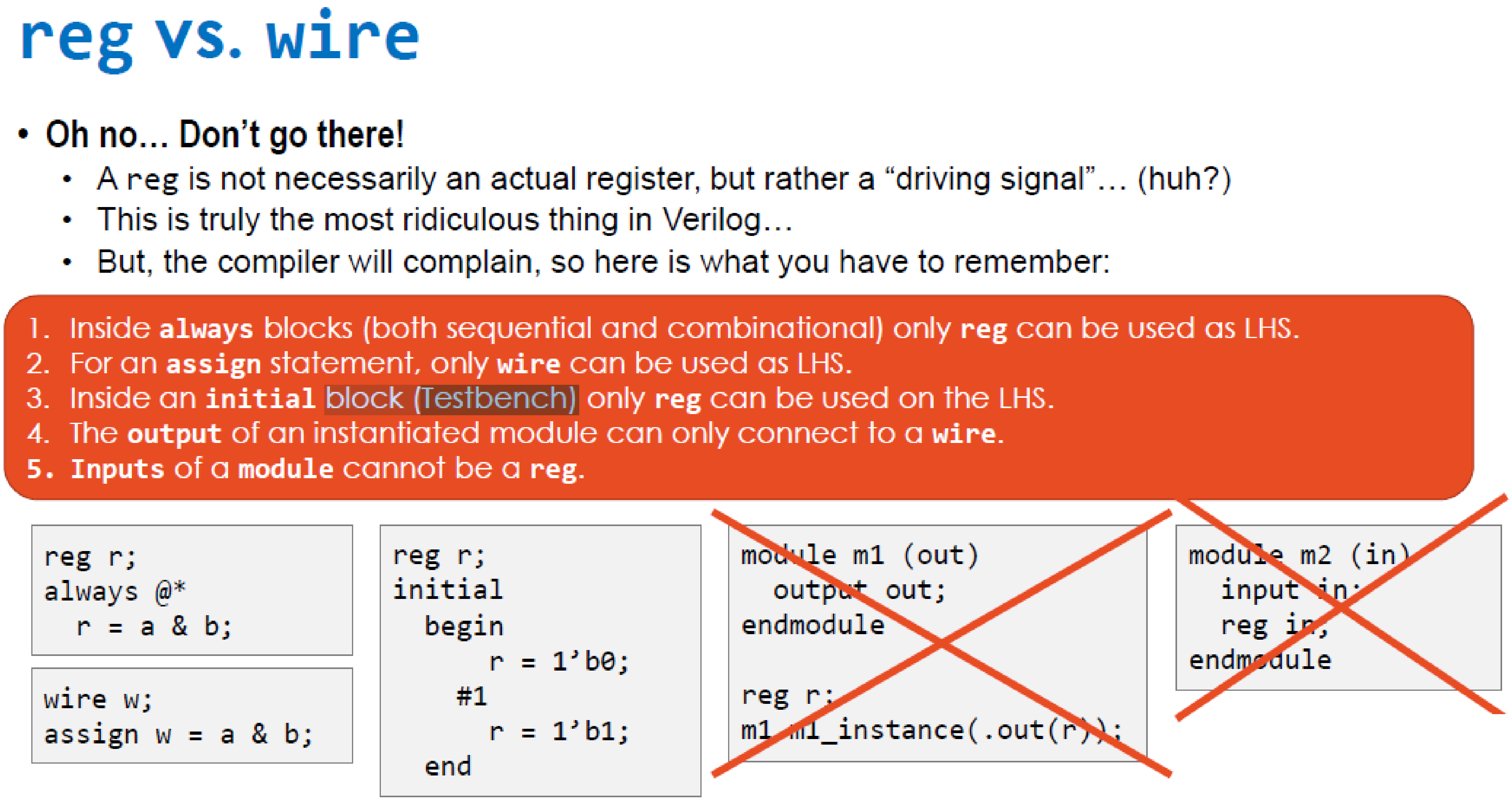
```
always
   #5 clk = ~clk;
```

- Set stimulus:

```
initial begin
  // @100, Start counting up
  //       until overflow
  #100  act = 1'b1;
        up_dwn_n = 1'b1;
  // Reset (10 cycles pulse)
  #1000 rst_n = 1'b0;
        act = 1'b0;
  #100  rst_n = 1'b1;
  // Do a count-up to 4 and
  //    then count-down to ovflw
  #100 act = 1'b1;
       up_dwn_n = 1'b1;
  #40 up_dwn_n = 1'b0;
end
endmodule
```

Reminder:

Testbench – Net types



reg vs. wire

- **Oh no... Don’t go there!**
  - A reg is not necessarily an actual register, but rather a “driving signal”… (huh?)
  - This is truly the most ridiculous thing in Verilog…
  - But, the compiler will complain, so here is what you have to remember:

1. Inside **always** blocks (both sequential and combinational) only **reg** can be used as LHS.
2. For an **assign** statement, only **wire** can be used as LHS.
3. Inside an **initial** block (Testbench) only **reg** can be used on the LHS.
4. The **output** of an instantiated module can only connect to a **wire**.
5. **Inputs** of a **module** cannot be a **reg**.

```
reg r;
always @*
  r = a & b;
```

```
wire w;
assign w = a & b;
```

```
reg r;
initial
  begin
      r = 1’b0;
    #1
      r = 1’b1;
  end
```

```
module m1 (out)
  output out;
endmodule

reg r;
m1 m1_instance(.out(r));
```

```
module m2 (in)
  input in;
  reg in;
endmodule
```