CIRCUIT DESIGN WITH HDL

Course outline

- Chapter 1: Introduction
- Chapter 2: Verilog Language Concepts
- Chapter 3: Structural modeling
- Chapter 4: Behavioral modeling
- Chapter 5: State machines
- Chapter 6: Tasks and Functions
- Chapter 7: Functional Simulation/Verification (Testbench)
- Chapter 8: Synthesis of Combinational and Sequential Logic
- Chapter 9: Post-synthesis design tasks
- Chapter 10: VHDL introduction

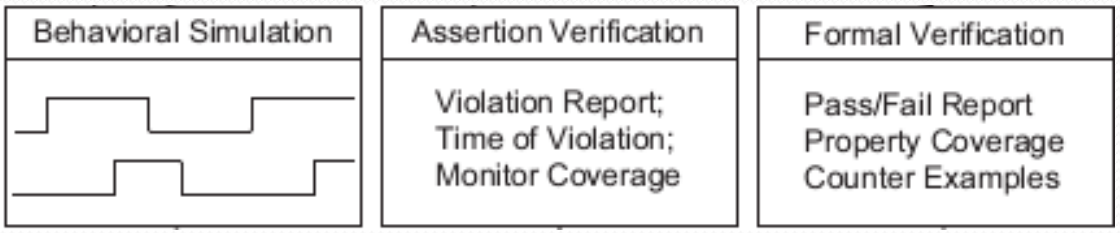
Digital design flow

Simulation & Function Verification

+ Design is simulated and tested for functionality before turning into hardware

+ Do not consider gate, propagation delay, hazards, glitches, race conditions, setup and hold violations, and other related timing issues.

+ Test data are generated by testbench or waveform editor

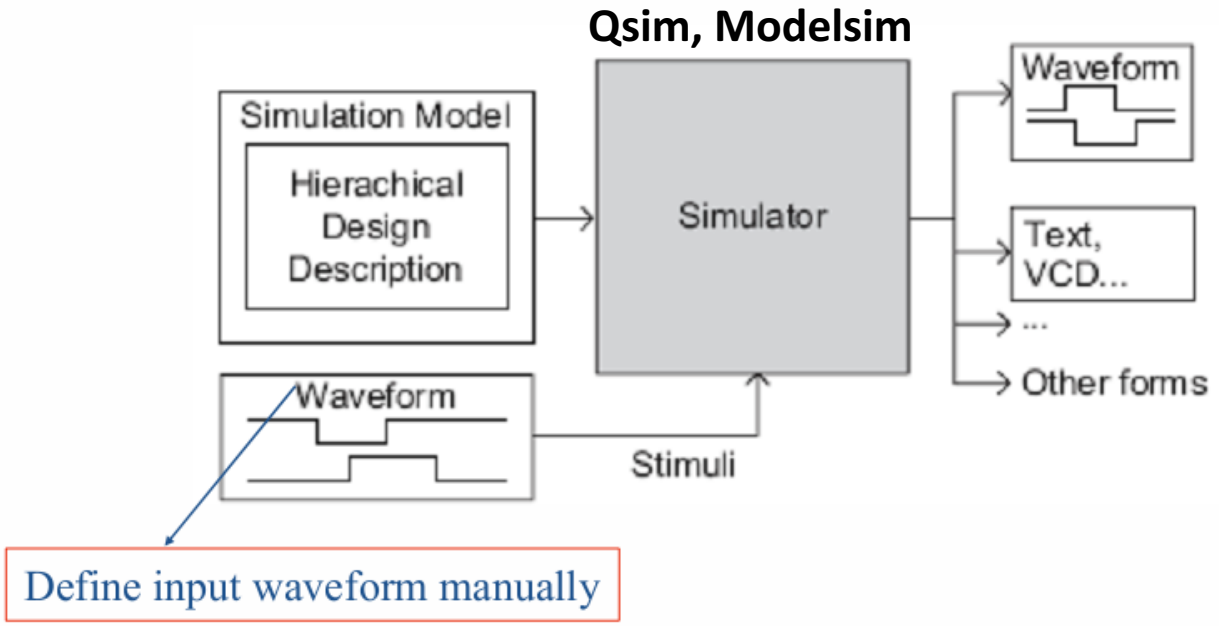


Ref. Verilog Digital System Design, Zainalabedin Navabi

- Design Specification
- Behavior Description
- Pre-synthesis verification
- Synthesis
- Routing and Placement
- Timing analyis
- Post-synthesis verification
- Physical layout
- Chip

CAD flow

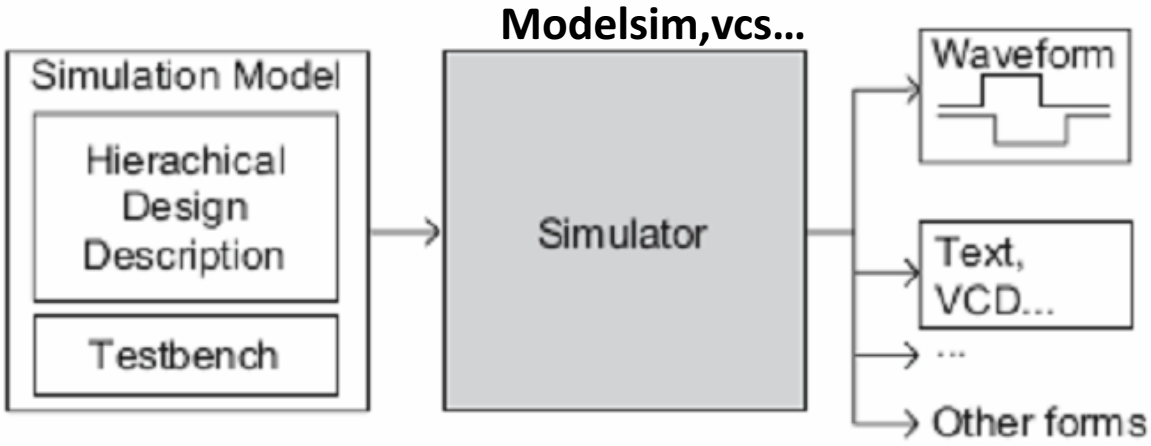
❖ Function verification with input waveform:



- Usually good for small design

CAD flow

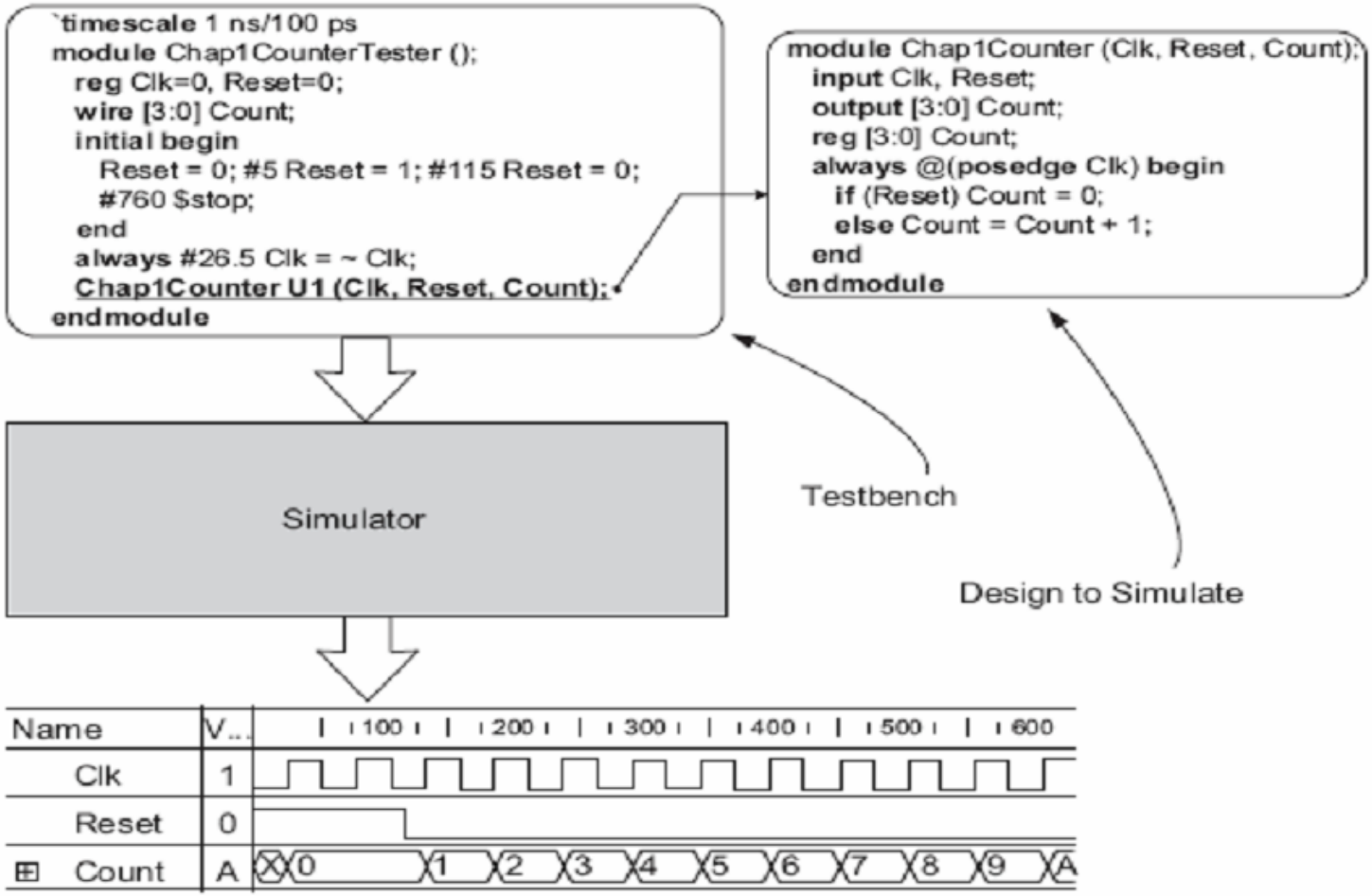
❖ Function verification with testbench:



- Using Verilog language for testing design module -> testbench
- Testbench is a code for test, not a part of final design.
- Timing & display procedures: important in designing testbench

Testbench

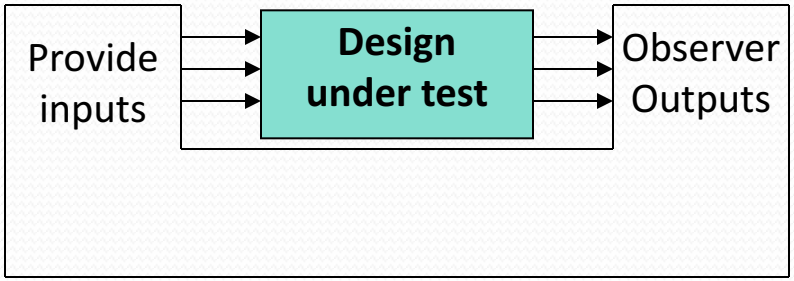
- Function verification with testbench:



Testbench

Structure of test bench

A test bench is a program which can give arbitrary inputs to Design Under Test (DUT) (or Module Under Test) and observe their outputs.



module test bench;
reg [7:0] dat, … *declaration of signals*

Connection of module under test …

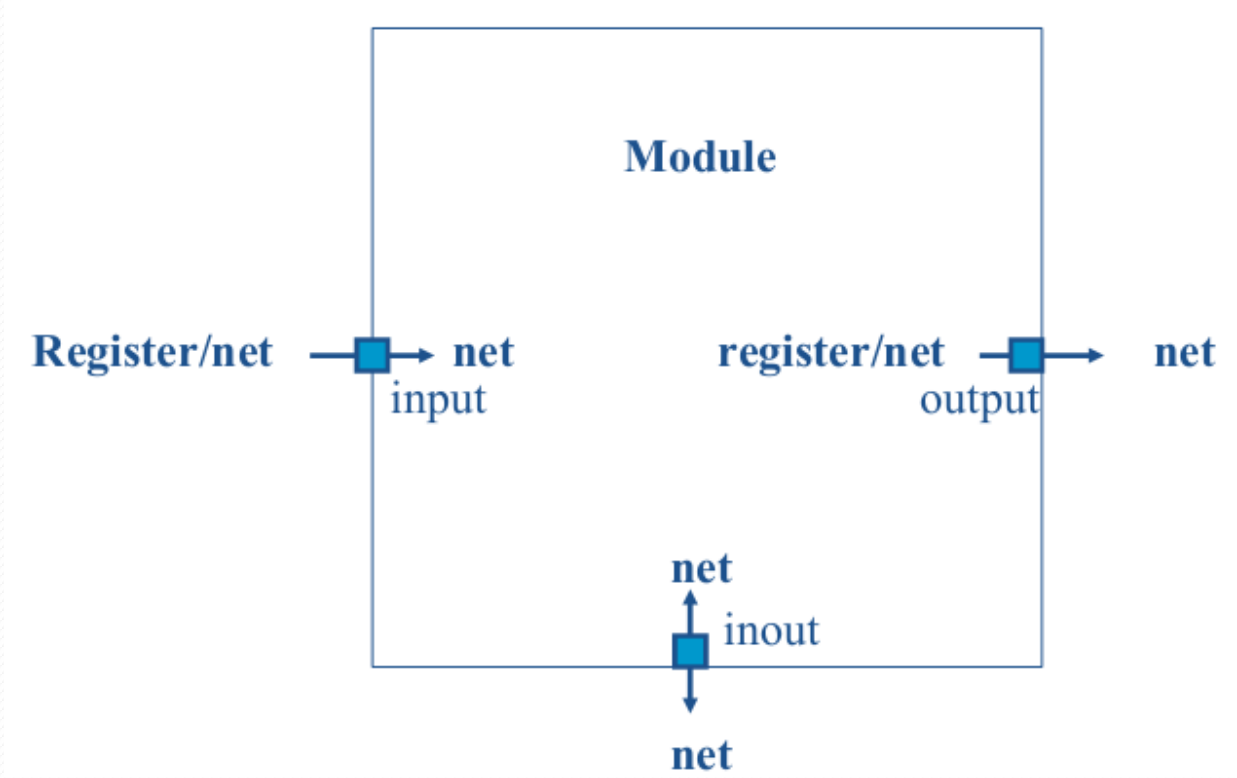
The description of the clock
initial begin …. end … *Test input declaration*
…..

endmodule

A description of test bench in Verilog-HDL

Testbench

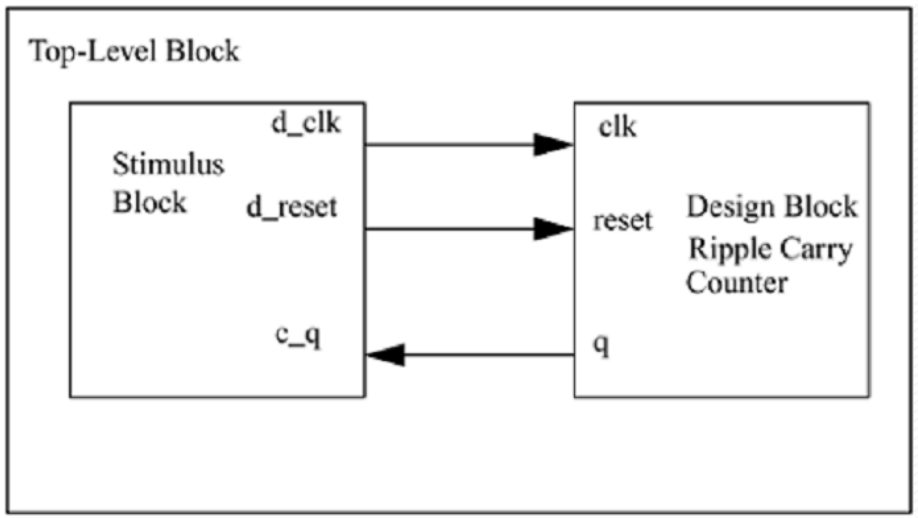
- Remember correct data types for ports



Structure of testbench

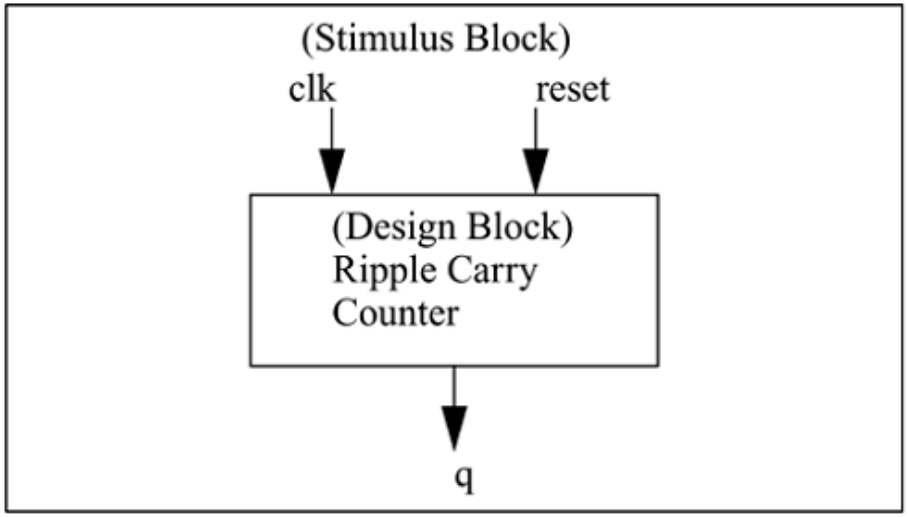
- 2 styles

Stimulus and Design block instantiated in a Dummy top-level module



Style 1

Stimulus block instantiates Design block



Style 2

Structure of testbench

- Style 1
  - Develop your hierarchical system within a module that has input and output ports (called "design" here)
  - Develop a separate module to generate tests for the module ("test")
  - Connect these together within another module ("testbench")

```
module testbench ();
    wire      l, m, n;

    design  d (l, m, n);
    test      t (l, m);

    initial begin
        //monitor and display
        ...
```

```
module design (a, b, c);
    input    a, b;
    output  c;
    ...
```

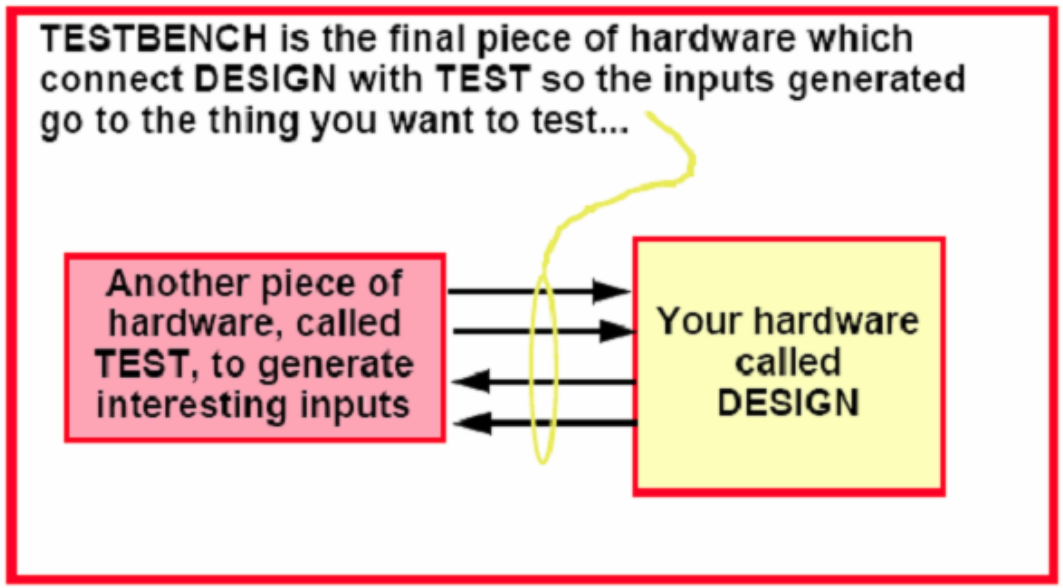
```
module test (q, r);
    output  q, r;

    initial begin
        //drive the outputs with signals
        ...
```

Structure of testbench

- Style 1

  Another view of this: 3 chunks of Verilog, one for each of:



Structure of testbench

- Style 1

Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the *design*

```
module tBench;
    wire     su, co, a, b;

    halfAdd      ad(su, co, a, b);
    testAdd      tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
    output   sum, cOut;
    input    a, b;

    xor #2   (sum, a, b);
    and #2   (cOut, a, b);
endmodule
```

```
module testAdd(a, b, sum, cOut);
    input    sum, cOut;
    output   a, b;
    reg      a, b;

    initial begin
        $monitor ($time,,
          "a=%b, b=%b, sum=%b, cOut=%b",
           a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

Structure of testbench

- Structure1 (Cont'd)

- It's the test generator
- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current simulation time
- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

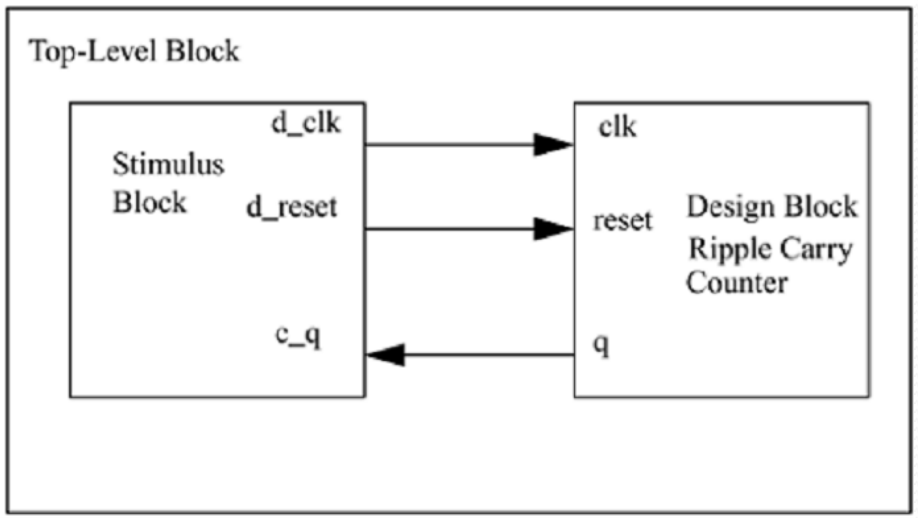
```
module testAdd(a, b, sum, cOut);
    input   sum, cOut;
    output  a, b;
    reg     a, b;

    initial begin
        $monitor ($time,,
          "a=%b, b=%b, sum=%b, cOut=%b",
          a, b, sum, cOut);
        a = 0; b = 0;
        #10 b = 1;
        #10 a = 1;
        #10 b = 0;
        #10 $finish;
    end
endmodule
```

Structure of testbench

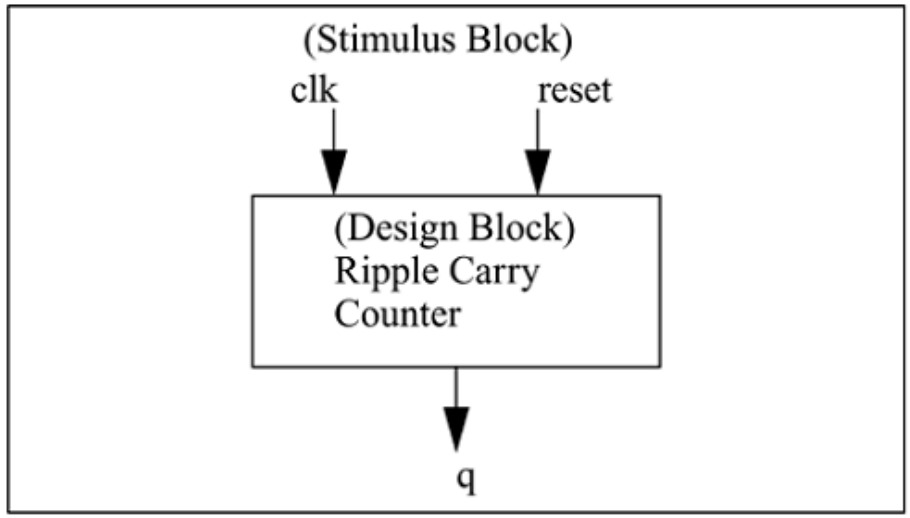
- 2 styles

Stimulus and Design block instantiated in a Dummy top-level module



Style 1

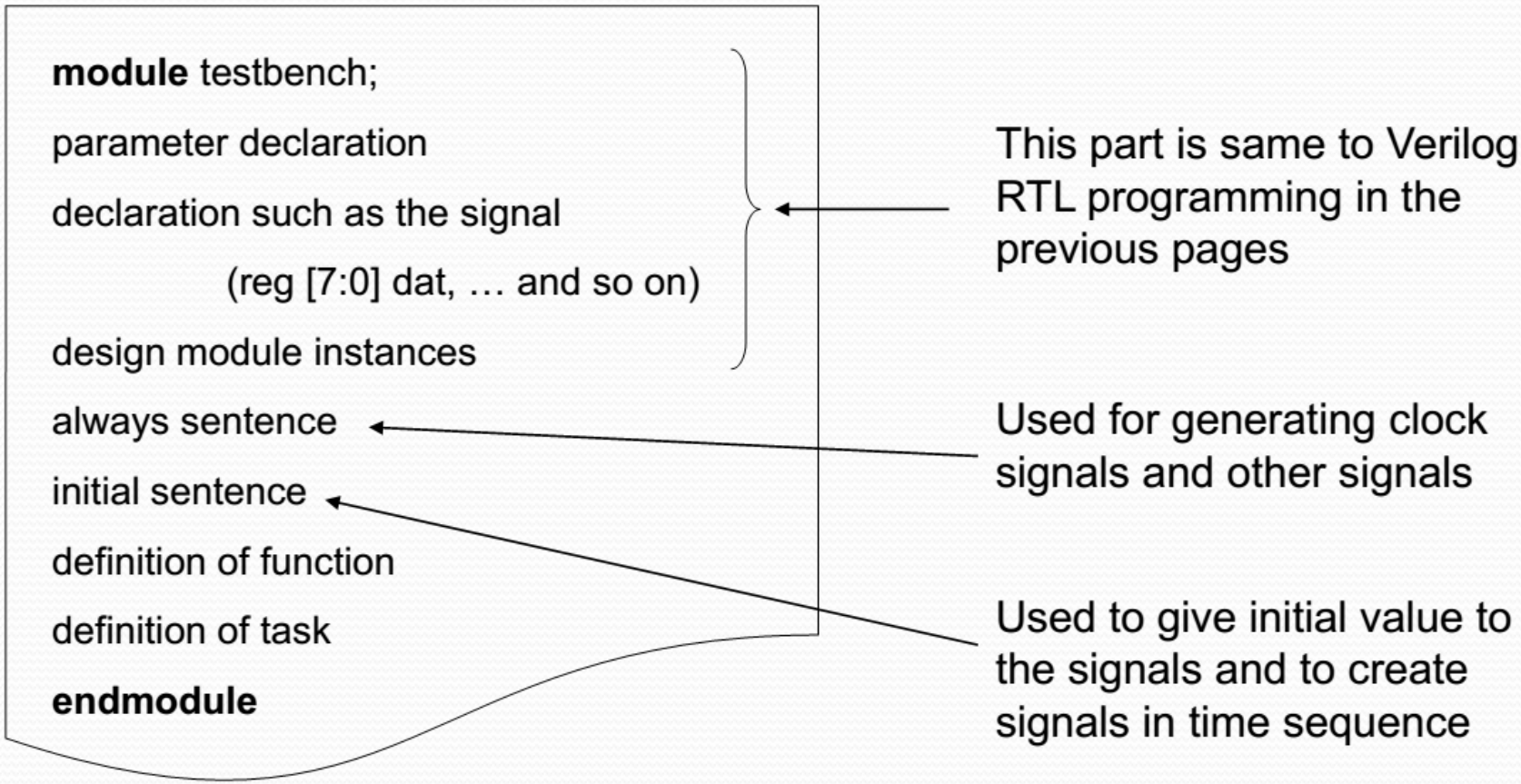
Stimulus block instantiates Design block



Style 2

Structure of testbench

- Style 2



Structure of testbench

- Style 2 (cont'd)

D_FF example:

```
module dff (d ,clk, q );
input d, clk;
output q;
reg q;
always @(posedge clk)
        q <= d;
endmodule
```

Structure of testbench

- Style 2 (cont'd)

```
`timescale 1ns/1ps
module stimulus;
reg din, clock;
reg exp;
wire dout;
parameter clk_cycle = 20;

// Instance DFF
dff instance_1 (.d(din), .clk(clock), .q(dout));

// Clock generator
always
        begin
        #(clk_cycle/2) clock = ~clock;
        $display( "Time at %5f. DIN: [%b]  DO=[%b]", $realtime, din, dout);
        end
```

Structure of testbench

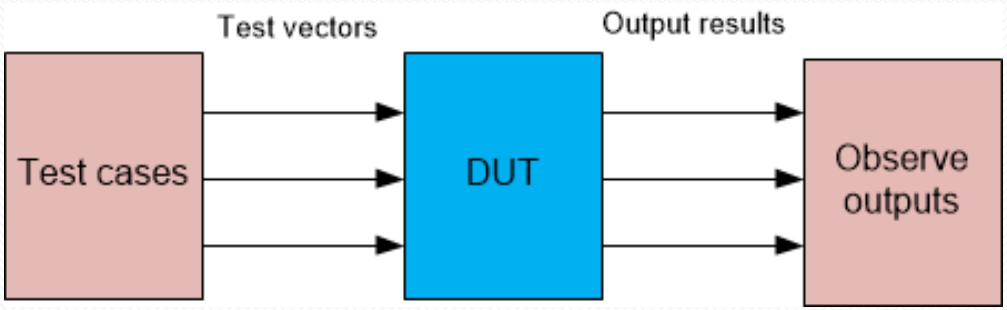
- Style 2 (cont'd)

```
// Input waveform and Expected output
    initial
    begin
            clock  = 1'b0; din    = 1'b0;
            fork
                    #6 din = 1'b1;
                    #11 exp = 1'b1;
                    #26 din = 1'b0;
                    #31 exp = 1'b0;
                    #46 din = 1'b1;
                    #51 exp = 1'b1;
                    #66 din = 1'b0;
                    #71 exp = 1'b0;
                    #100 $finish;
            join
    end
```

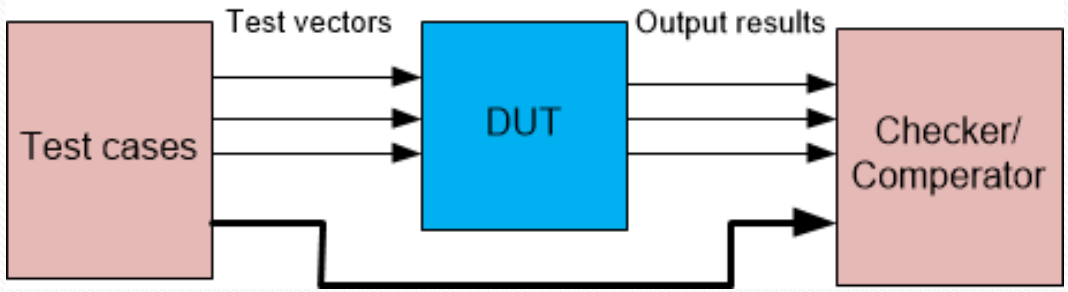
```
//Compare output
        always @(negedge clock)
        begin
            if (dout !== exp)
                $display( "FAIL: %5f - output=[%b]
                  Exp=[%b]", $realtime, dout, exp);
            else
                $display( "PASS: %5f - output=[%b]
                  Exp=[%b]", $realtime, dout, exp);
        end
endmodule
```

Test plan

- *Test plan for writing testbench*



Observe result model



Self-checking model

Testbench techniques

- **Write testbench for 101 Moore detector**

```
module moore_detector (input x, rst, clk, output z );

   parameter [1:0] a=0, b=1, c=2, d=3;
   reg [1:0] current;

   always @( posedge clk )
      if ( rst )  current = a;
      else case ( current )
         a : current = x ? b : a ;
         b : current = x ? b : c ;
         c : current = x ? d : a ;
         d : current = x ? b : c ;
         default : current = a ;
      endcase

   assign z = (current==d) ? 1'b1 : 1'b0;

endmodule
```

Ref: Verilog Digital System Design – Zainalabedin Navabi

Testbench techniques

Simulation control

- $stop, $finish: stop and finish a simulation. A stop simulation can be resumed, finish one can not.

```
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

    moore_detector MUT ( x, reset, clock, z );

    initial #24 reset=1'b0;
    always #5 clock=~clock;
    always #7 x=~x;
    initial #189 $stop;

endmodule
```

Testbench techniques

Limiting data set

- Instead of setting simulation time limit, a testbench can put a limit on the number of data put on inputs of a MUT
- **$random**: generate random data

```
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

    moore_detector MUT ( x, reset, clock, z );

    initial #24 reset=1'b0;
    initial repeat(13) #5 clock=~clock;
    initial repeat(10) #7 x=$random;

endmodule
```

Testbench techniques

Applying synchronized data

- Synchronize data with clock

```
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

    moore_detector MUT ( x, reset, clock, z );

    initial #24 reset=0;
    initial repeat(13) #5 clock=~clock;
    initial forever @(posedge clock) #3 x=$random;

endmodule
```

  - *The stable data after the positive edge of the clock will be used by moore_detector on the next leading edge of the clock.*

Testbench techniques

Synchronized display of result

```
module test_moore_detector;
    reg x=0, reset=1, clock=0;
    wire z;

    moore_detector MUT ( x, reset, clock, z );

    initial #24 reset=0;
    initial repeat(13) #5 clock=~clock;
    initial forever @(posedge clock) #3 x=$random;
    initial forever @(posedge clock) #1 $displayb(z);

endmodule
```

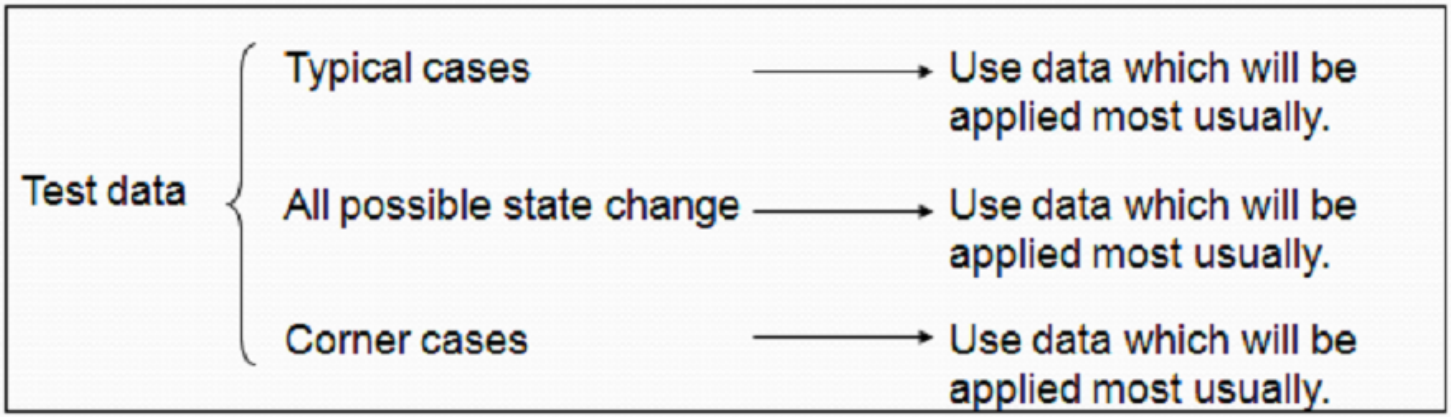
- *When being displayed using **$displayb** task, the circuit output is supposed to have stable values.*

Test vector definition

While programming RTL code, it is important to assume various data as input and make your code prepared for such data. Your code will not work properly for data which you did not expected to come. This means your imaginative power decide the quality of your program.

To verify design function correctly but less time comsuming, it is very important for an engineer to be able to select or determine proper data to test a module he/she designed.

Test data shall be selected so that all the possible paths of your code are covered.



Test vector definition

To do test, you must have the state transition matrix of your logic. You have to apply data which causes all the possible transition of the state.

state / input	INTL	ST1	ST2	….
in1	no-op	→ST2	→ST3	….
in2	→ST1	→ST3		
in3	→ST2			
….	….			

If you have not prepared a state transition matrix, write it.

A state transition matrix is needed to verify our logic.

Test vector definition

How to select typical and corner cases.

What data can check corner cases depend on the logic to handle them. However, we have to have a good sensitivity to tell what kind of data may be critical for various logic.

Example:

For 8-bit numerical input data,
8'h25, 8'h74, 8'h09, etc. may be typical input
8'h00, 8'hFF may be corner case input

For 1-bit control input data which is supposed to be 1 several times for certain period of time,
Input sequence: 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, … may be typical input,
1, 1, 1, 1, 1, 1, 1, 1, … may be corner case input.

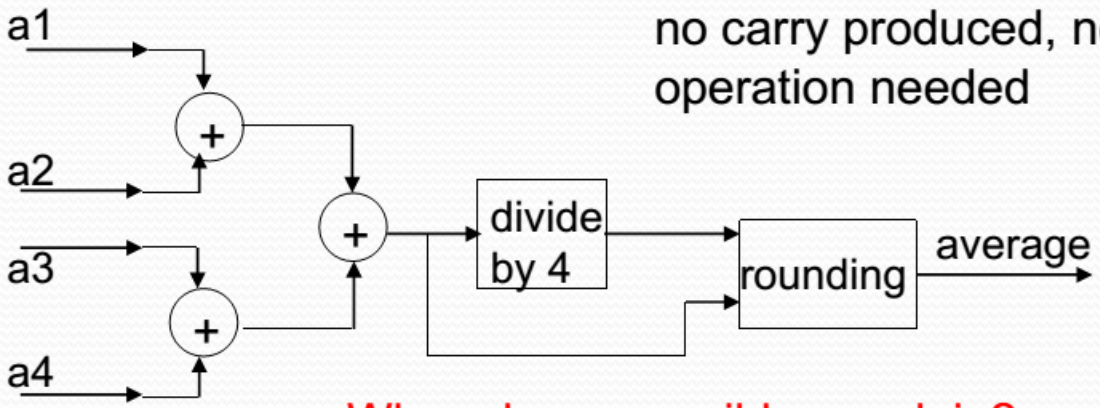
Test vector definition

Suppose testing logic which calculate average of four 4-bit positive integers. 4-bit output average must be rounded.

If we prepare the following data for testing, is it effective for finding bugs?

Prepared data: 4'b0010, 4'b0101, 4'b0100, and 4'b0001,

add result 12 is still 4-bit integer data, no carry produced, no round up operation needed



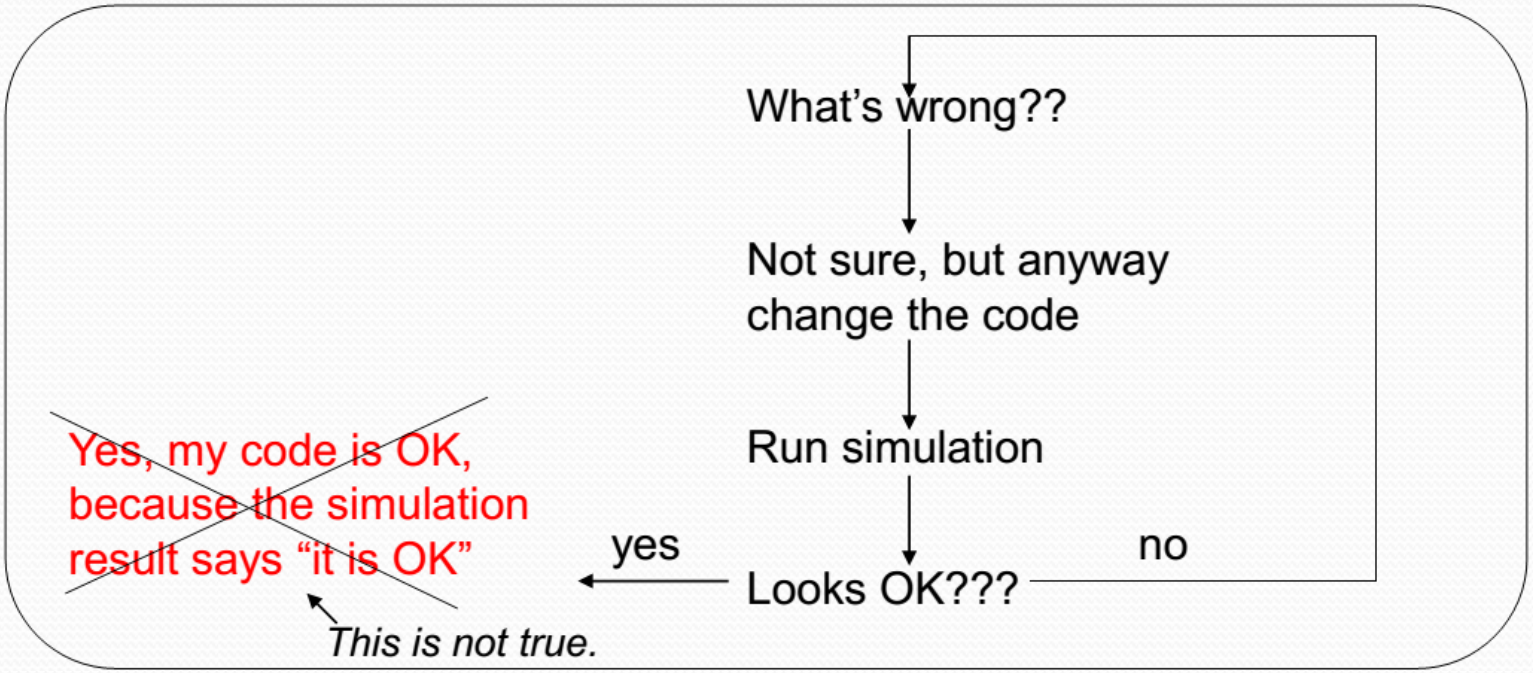
There are several ways to implement the logic. A diagram shown on the left may be one possible implementation

Where bugs possibly sneak in?

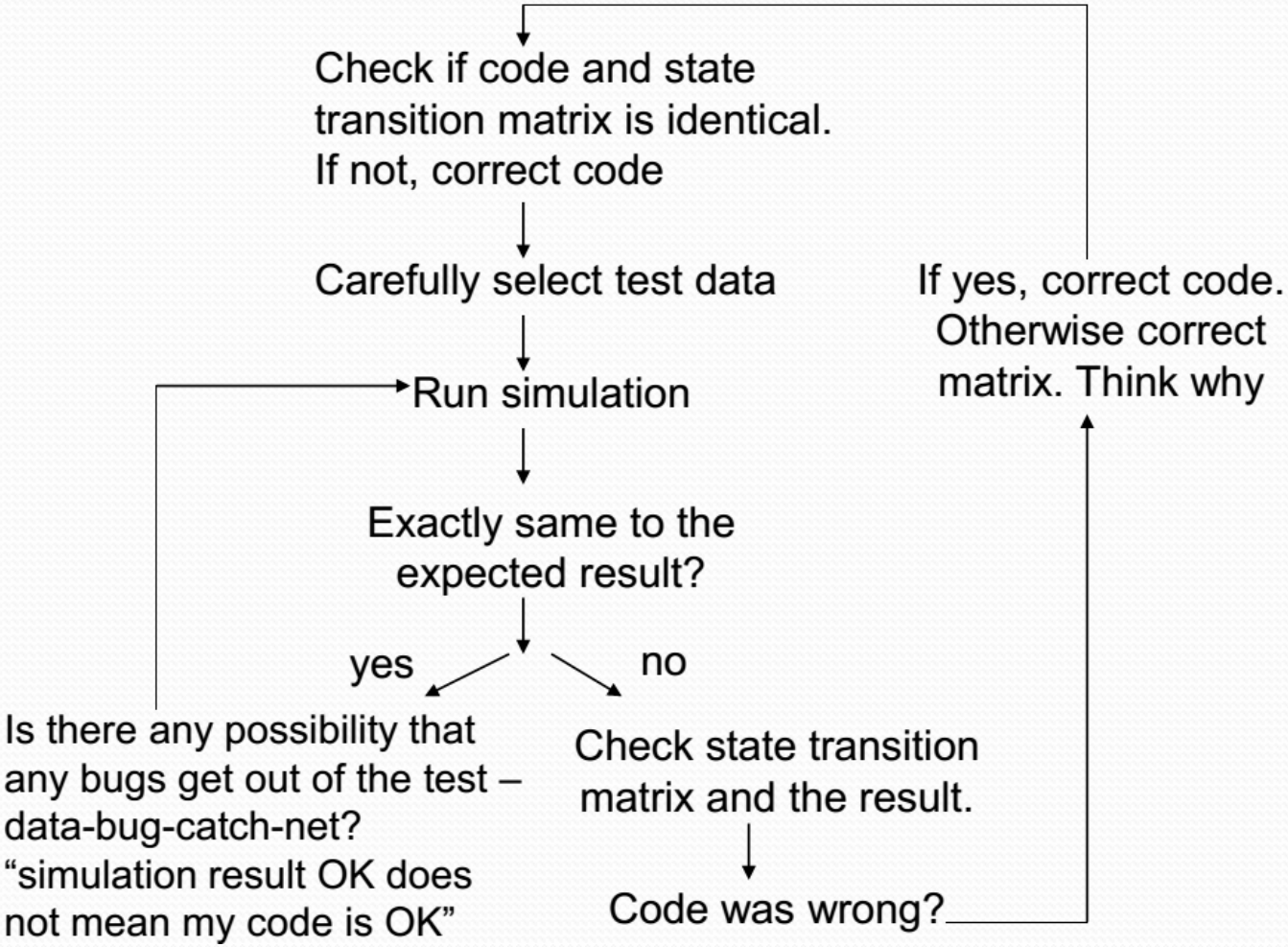
Your imaginative power is needed!!

Test vector definition

- Many kind of bugs escape through RTL simulation test. This means that we can not be safe from bugs even if simulation test looks OK.
- Therefore the following method is the most terrible way to fix bugs. Never do this way.



Test vector definition



Example 1

```
module mux2 (f,a,b,s);
  output f;
  input a,b,s;
  reg f;
  always @ (a or b or s)
    if (s) f=a;
    else f=b;
endmodule // mux2

module testbench (in1,in2,select,out);
  wire in1,in2,select,out;
     testmux test (.a(in1), .b(in2), .s(select), .f(out));
     mux2 mux  (.f(out), .a(in1), .b(in2), .s(select));
endmodule
```

Example 1

```
module testmux (a,b,s,f);
   output a, b, s;
   input f;
   reg a, b, s;
   reg expected;
   initial
      begin
            s=0; a=0; b=1; expected=0;
            #10 a=1; b=0; expected=1;
            #10 s=1; a=0; b=1; expected=1;
             end

  initial
     $monitor("select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
         s, a, b, f, expected, $time);
  endmodule // testmux
```

Example 2

- Encoder 8 to 3

Input								Output		
D7	D6	D5	D4	D3	D2	D1	D0	A2	A1	A0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Example 2

```
module encoder8_3( output reg [2:0] encoder_out, input enable, input [7:0] encoder_in );
    always @ (enable or encoder_in)
    begin
            if (enable)
            case ( encoder_in )
                    8'b00000001 : encoder_out = 3'b000;
                    8'b00000010 : encoder_out = 3'b001;
                    8'b00000100 : encoder_out = 3'b010;
                    8'b00001000 : encoder_out = 3'b011;
                    8'b00010000 : encoder_out = 3'b100;
                    8'b00100000 : encoder_out = 3'b101;
                    8'b01000000 : encoder_out = 3'b110;
                    8'b10000000 : encoder_out = 3'b111;
                    default : $display("Check input bits.");
            endcase
    end
endmodule
```

Example 2

```
module stimulus;
    wire[2:0] encoder_out;
    reg enable;
    reg[7:0] encoder_in;
     reg [2:0] expected;
    encoder8_3 enc( encoder_out, enable, encoder_in );
    initial  begin
            enable = 1; encoder_in = 8'b00000010, expected = 3’b001;
            #1 $display("enable = %b, encoder_in = %b, encoder_out = %b, expected_out = %b", enable, encoder_in, encoder_out, expected);
            #1 enable = 0; encoder_in = 8'b00000001; expected = 3’b001;
            #1 $display("enable = %b, encoder_in = %b, encoder_out = %b, expected_out = %b ", enable, encoder_in, encoder_out, expected);
            #1 enable = 1; encoder_in = 8'b00000001; expected = 3’b000;
            #1 $display("enable = %b, encoder_in = %b, encoder_out = %b, expected_out = %b ", enable, encoder_in, encoder_out, expected);
            #1 $finish;
    end
endmodule
```

More reference

- http://www.asic-world.com/verilog/art_testbench_writing.html
- http://testbench.in/

SystemVerilog		Verilog	OpenVera	Miscellanious
Basic Constructs	Verification Concepts	Verification Concepts	Constructs	Articles
Interface	UVM Tutorial	Switch Example	Switch Example	Specman E Tutorial
OOPS	VMM Tutorial	Basic Constructs	RVM Switch Example	Interview Questions
Randomization	OVM Tutorial		RVM Ethernet Sample	
Functional Coverage	Easy Labs : SV			
Assertion	Easy Labs : UVM			
DPI	Easy Labs : OVM			
VMM Ethernet Example	Easy Labs : VMM			
	Easy Labs : AVM			