NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING

LECTURE

Subject:

VERILOG

Hardware Design Language

Chapter9: Testbench and Verification

Lecturer: Lam Duc Khai

Agenda

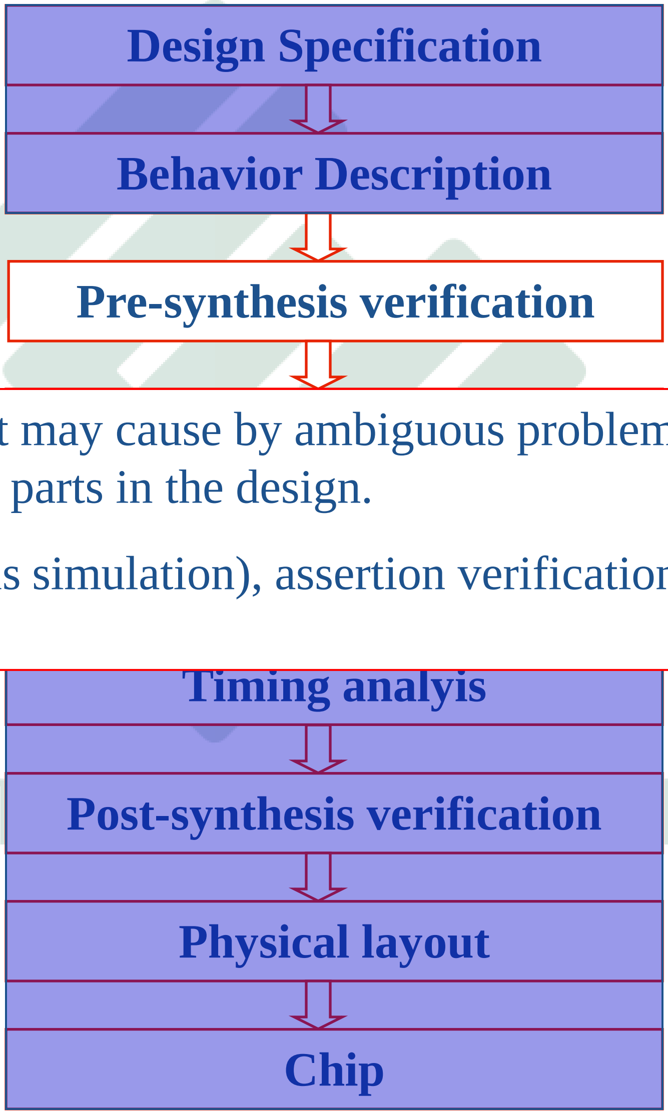
1. Chapter 1: Introduction (Week1)
2. Chapter 2: Fundamental concepts (Week1)
3. Chapter 3: Modules and hierarchical structure (Week2)
4. Chapter 4: Primitive Gates – Switches – User defined primitives (Week2)
5. Chapter 5: Structural model (Week3)
6. Chapter 6: Behavioral model – Combination circuit & Sequential circuit (Week4 & Week5)
7. Chapter 7: State machines (Week6)
8. Chapter 8: Testbench and verification (Week7)

9 Chapter 9: Tasks and Functions (Week8)

Agenda

CAD flow reminder

❖ Pre-synthesis verification:

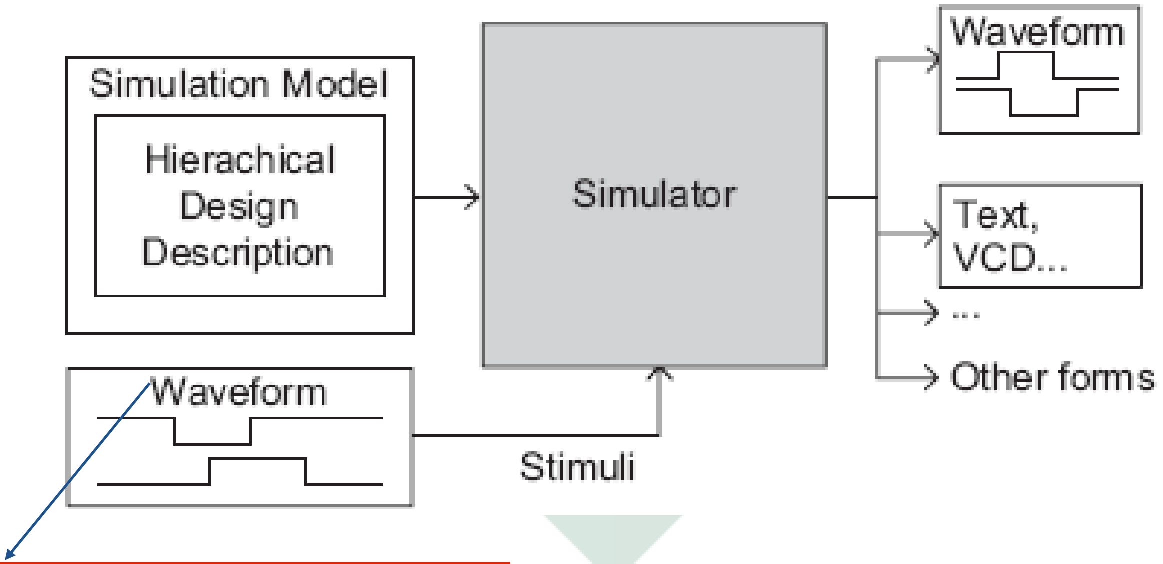


+ Check design function flaws that may cause by ambiguous problem specification, designer errors, or incorrect use of parts in the design.

+ Done by simulation (presynthesis simulation), assertion verification with **testbench** definition or **input waveform.**

CAD flow reminder (Cont'd)

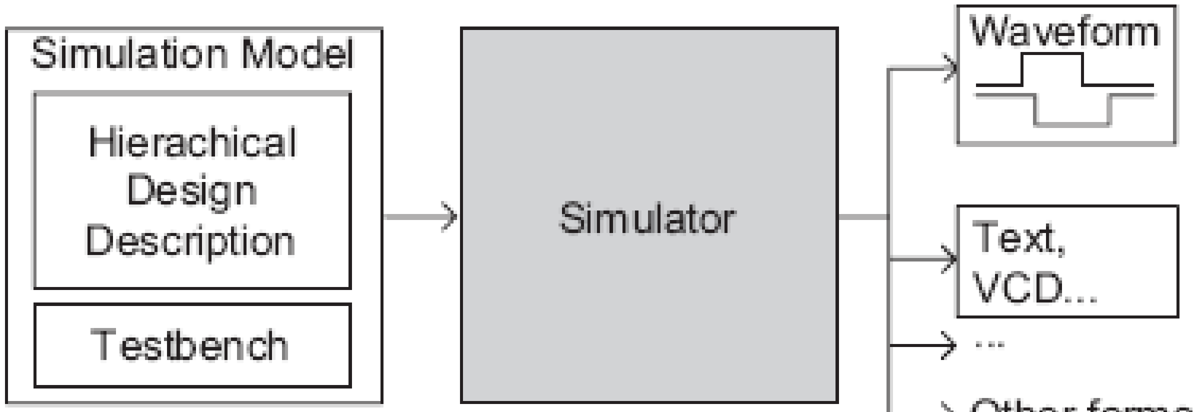
❖ Function verification with input waveform:



Define input waveform manually

CAD flow reminder (Cont'd)

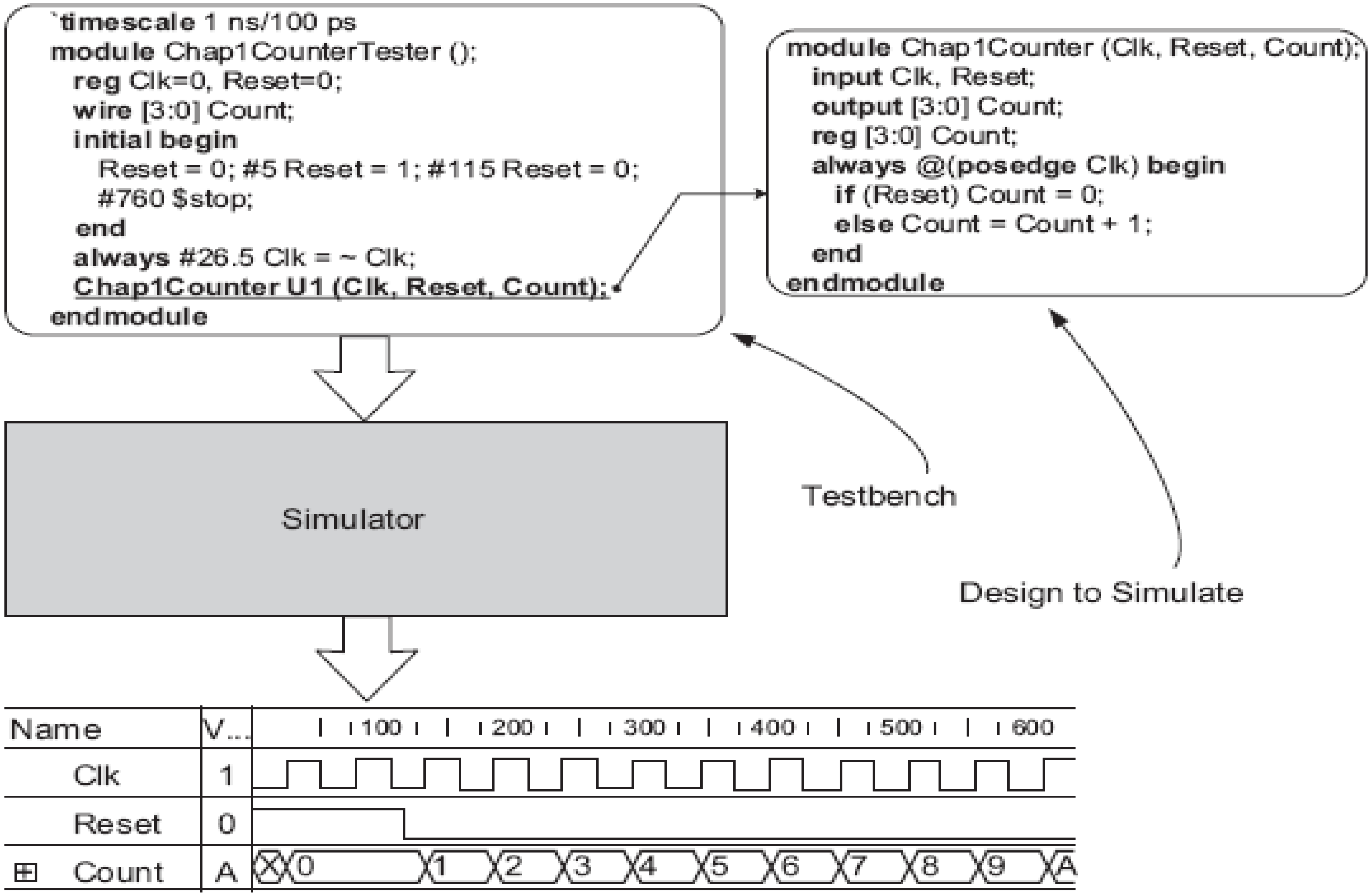
❖ Function verification with testbench:



- A testbench is used to verify that the logic is correct. The testbench instantiates the logic under test. It reads a file of inputs and expected outputs called test -vectors, applies them to the module under test, and logs mismatches.
- Testbench is a code for test, not a part of final design.

CAD flow reminder (Cont'd)

❖ Function verification with testbench:

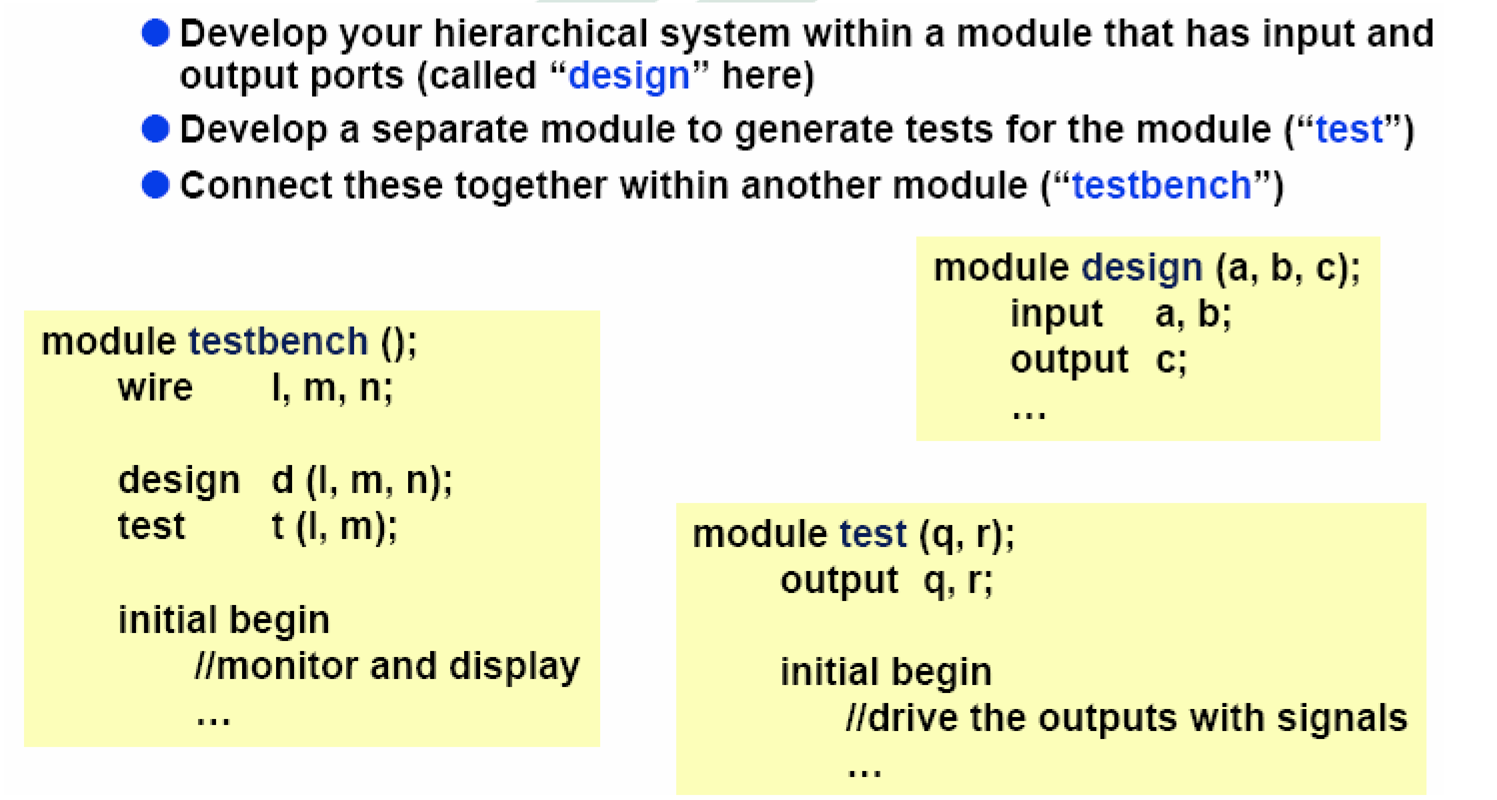


```
`timescale 1 ns/100 ps
module Chap1CounterTester ();
  reg Clk=0, Reset=0;
  wire [3:0] Count;
  initial begin
    Reset = 0; #5 Reset = 1; #115 Reset = 0;
    #760 $stop;
  end
  always #26.5 Clk = ~ Clk;
  Chap1Counter U1 (Clk, Reset, Count);
endmodule
```

```
module Chap1Counter (Clk, Reset, Count);
  input Clk, Reset;
  output [3:0] Count;
  reg [3:0] Count;
  always @(posedge Clk) begin
    if (Reset) Count = 0;
    else Count = Count + 1;
  end
endmodule
```

Structure of a testbench

- Structure1



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

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

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

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

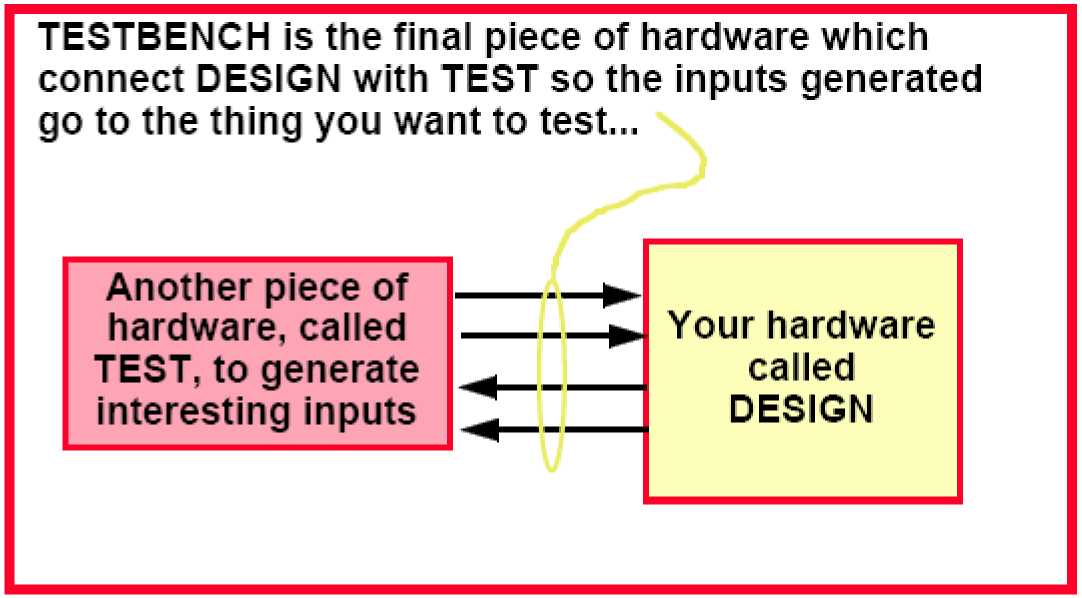
```
module test (q, r);
      output  q, r;

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

Structure of a testbench (Cont’d)

- Structure1 (Cont’d)

Another view of this

- 3 chunks of verilog, one for each of:


TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...
Another piece of hardware, called TEST, to generate interesting inputs
Your hardware called DESIGN

Structure of a testbench (Cont'd)

- Structure1 (Cont'd)

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

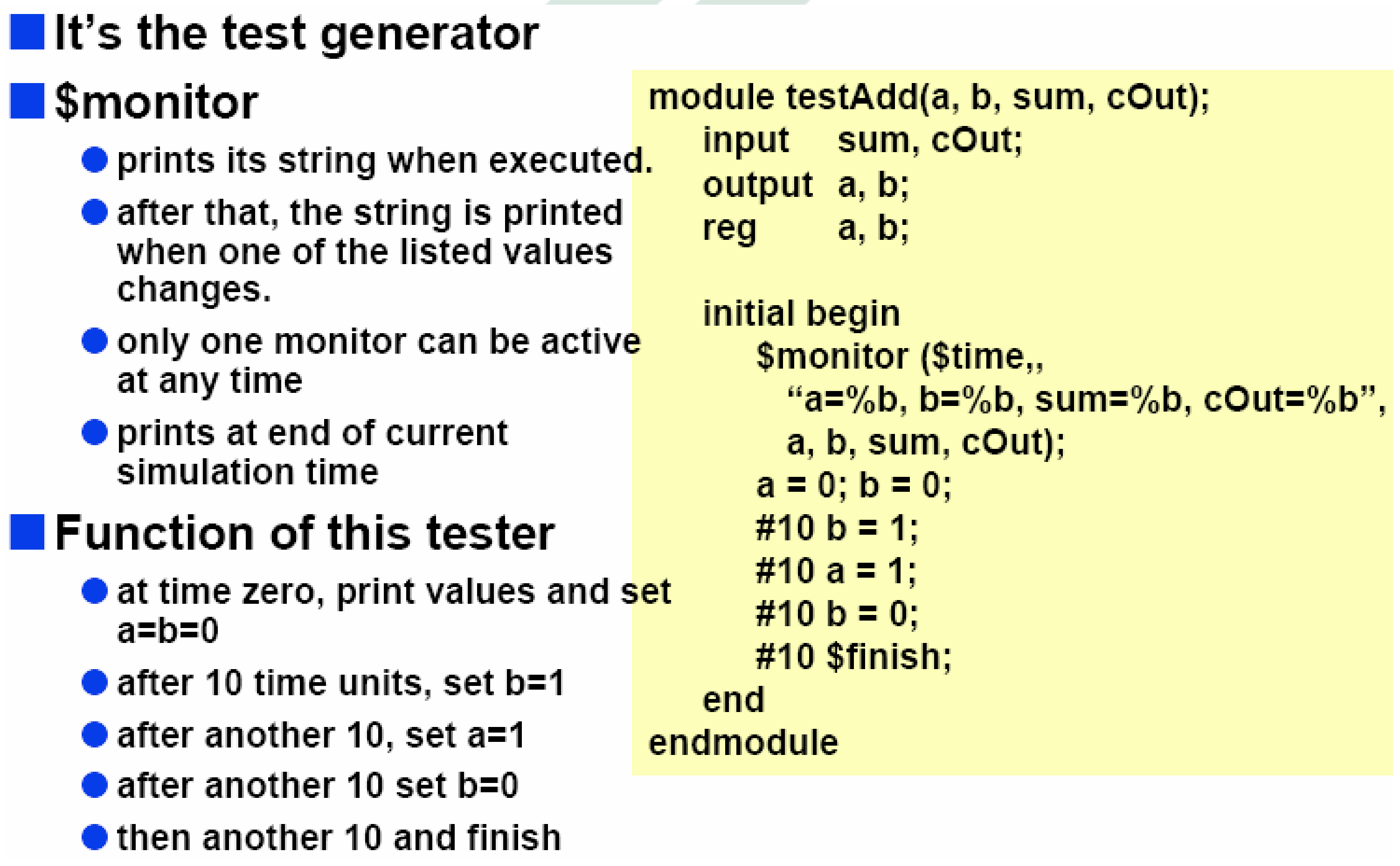
```
module tBench;
    wire      su, co, a, b;

    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 a testbench (Cont'd)

•Structure1 (Cont'd)

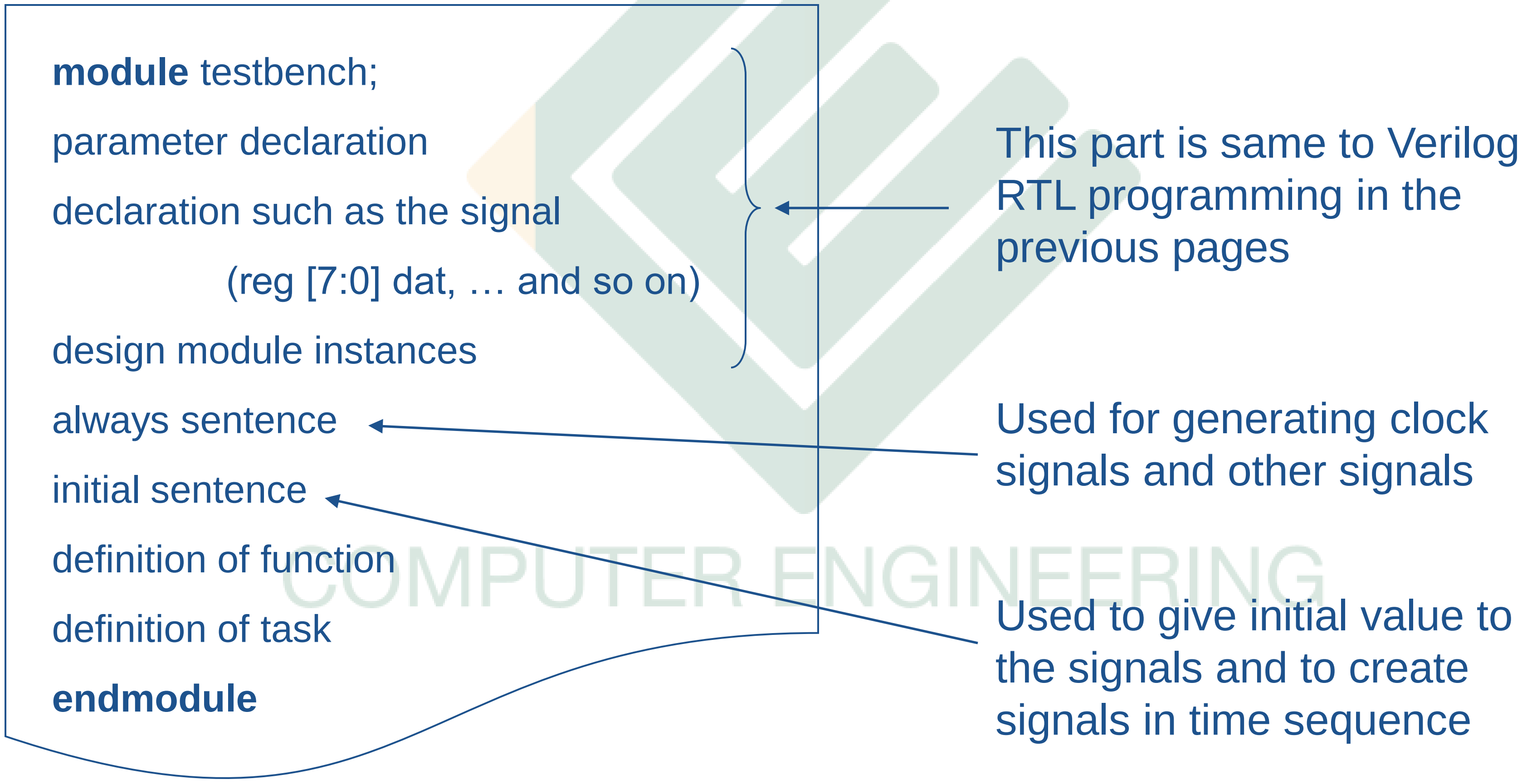


- It's the test generator
- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current simulation time
- Function of this tester
  - at time zero, print values and 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 a testbench (Cont'd)

- Structure2



module testbench;

parameter declaration

declaration such as the signal

(reg [7:0] dat, … and so on)

design module instances

always sentence

initial sentence

definition of function

definition of task

endmodule

This part is same to Verilog RTL programming in the previous pages

Used for generating clock signals and other signals

Used to give initial value to the signals and to create signals in time sequence

Structure of a testbench (Cont'd)

•Structure2 (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 a testbench (Cont'd)

- Structure2 (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 a testbench (Cont’d)

- Structure2 (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 vectors 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 data	Typical cases	→	Use data which will be applied most usually.
	All possible state change	→	Use data which will be applied most usually.
	Corner cases	→	Use data which will be applied most usually.

Test vectors definition (Cont’d)

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.

Without a state transition matrix, we can not verify our logic.

Test vectors definition (Cont'd)

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.

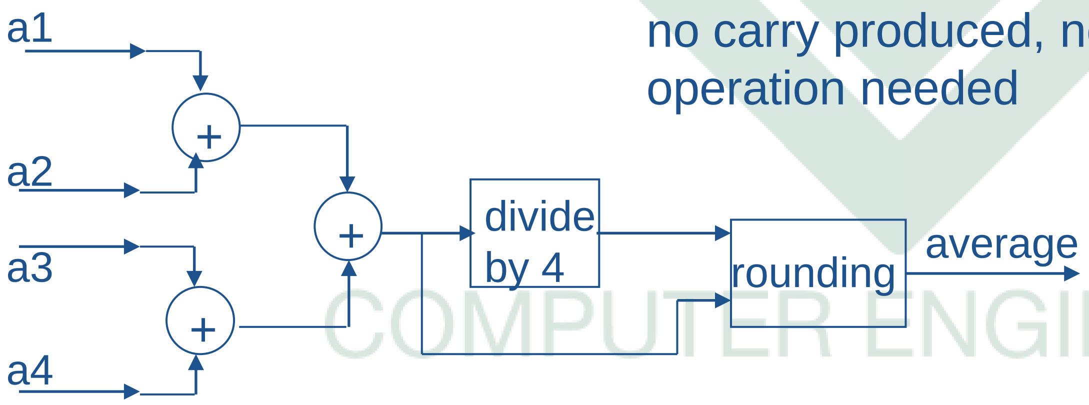
Test vectors definition (Cont'd)

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: $\underbrace{\text{4'b0010, 4'b0101, 4'b0100, and 4'b0001,}}$

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



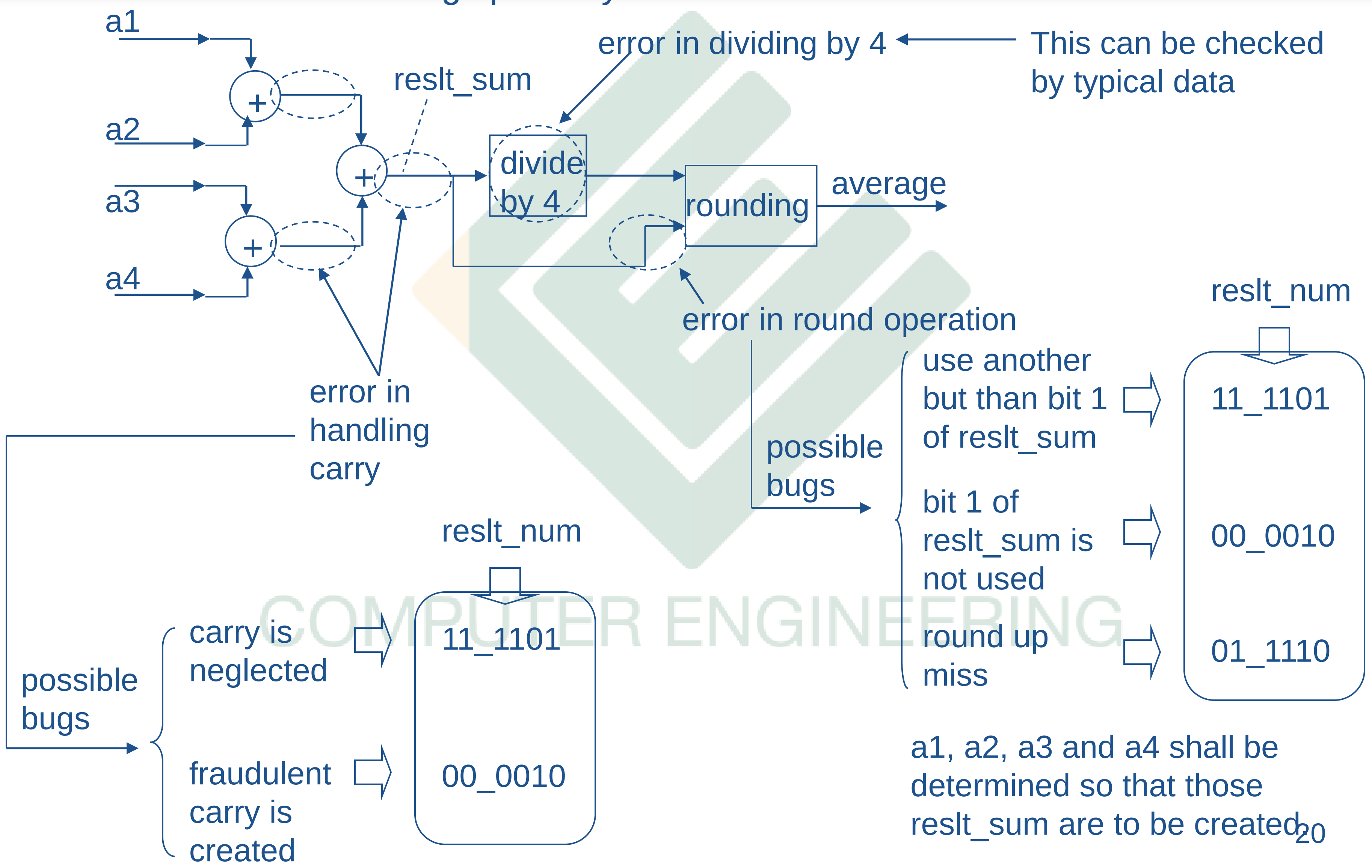
There are several ways to implement the logic. A diagram shown on the left may be one possible implementation

Where bugs possibly sneak in?

Your imaginative power is needed!!

Test vectors definition (Cont’d)

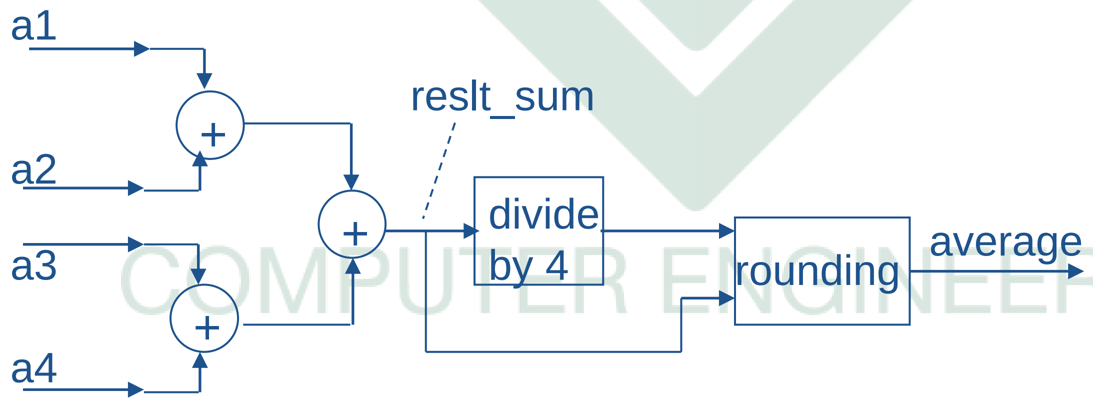
Where bugs possibly sneak in?



a1, a2, a3 and a4 shall be determined so that those reslt_sum are to be created

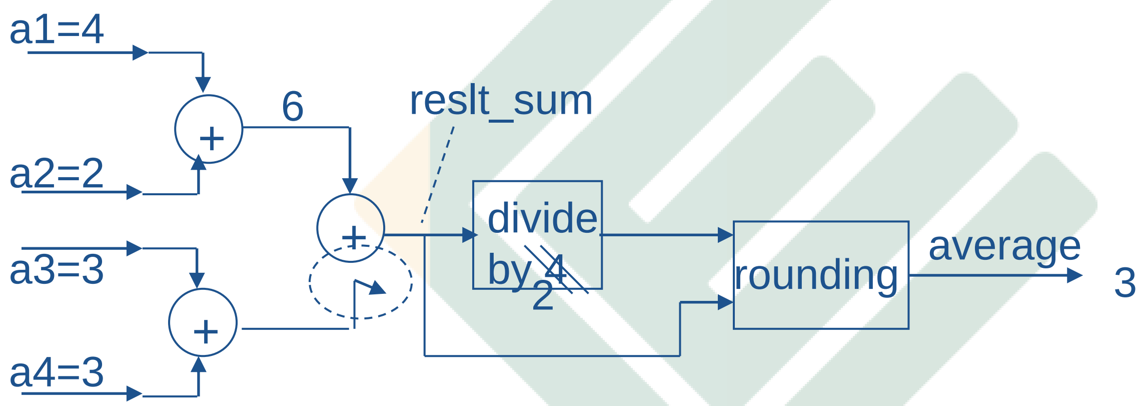
Test vectors definition (Cont'd)

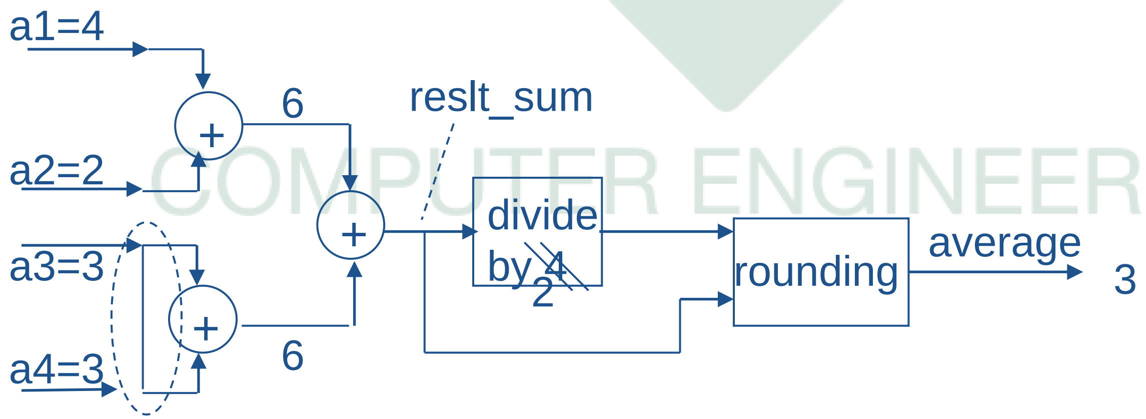
Question: For the logic below, we got output average 3, for the input a1=4, a2=2, a3=3, and a4=3. The result is looks OK. However what kind of bugs may exist in the logic. Imagine!!!



Test vectors definition (Cont'd)

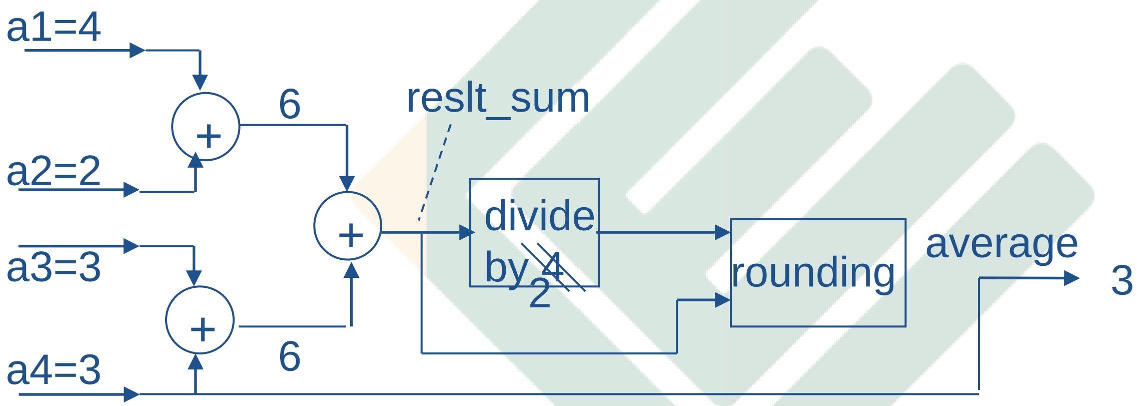
A sample answer.





Test vectors definition (Cont'd)

A sample answer (continued)

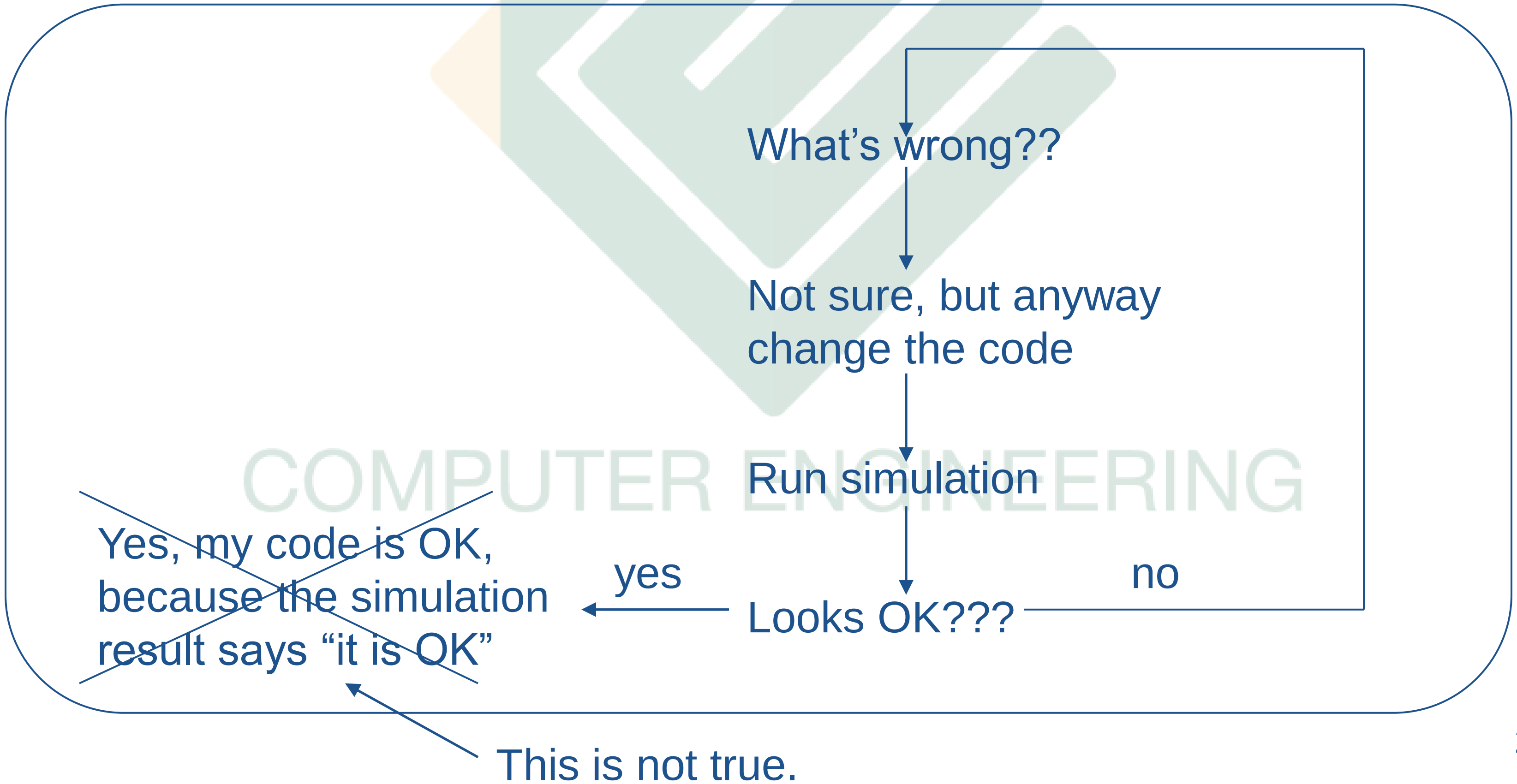


Use **your imaginative power,** **there is no limitation of absurdity of bugs. And they always try to go beyond your imaginative power**

Test vectors definition (Cont'd)

Many kind of bugs escape through RTL simulation test. This means that we can not be sage from bugs even if simulation test looks OK.

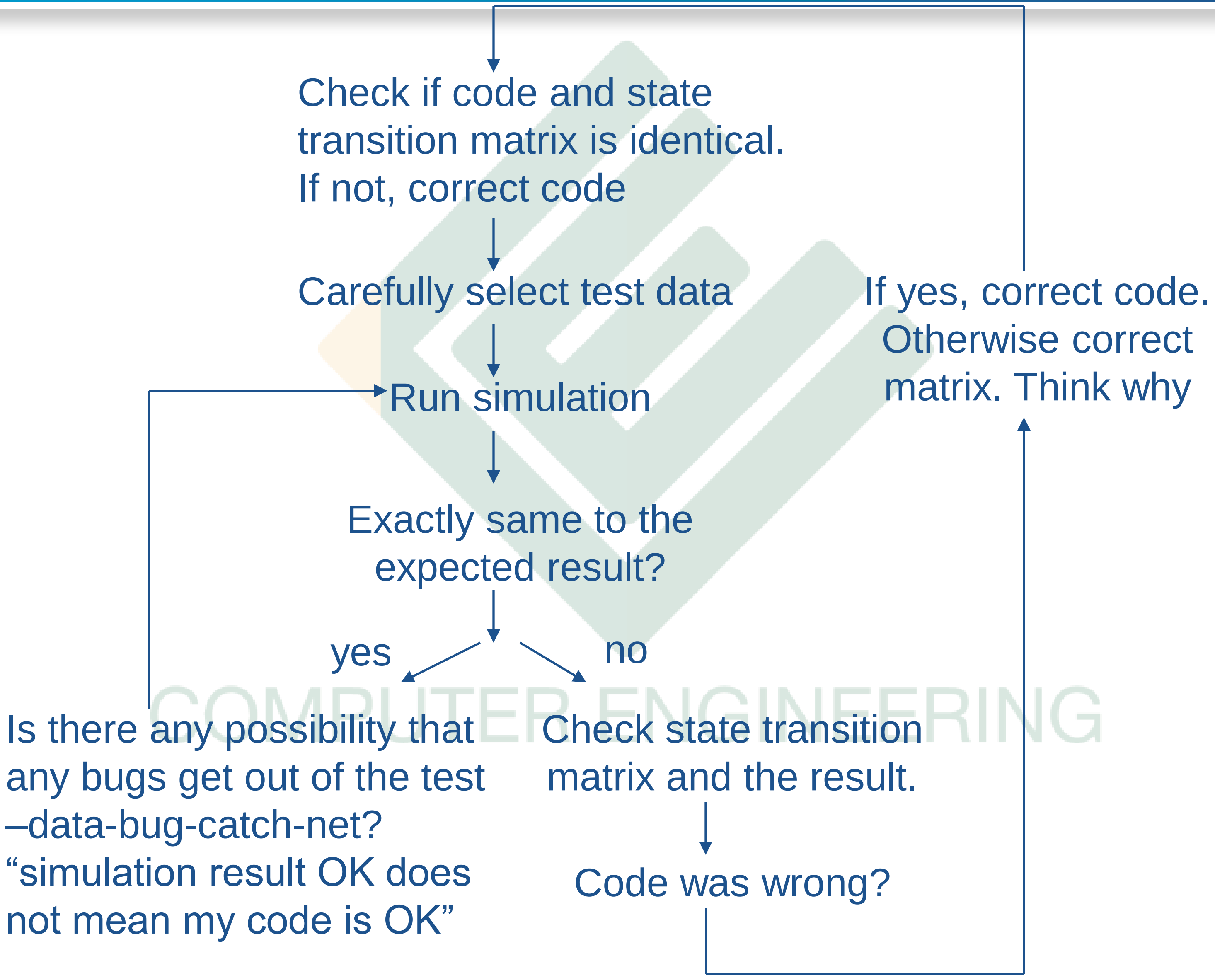
Therefore the following method is the most terrible way to fix bigs. Never do this way.


What's wrong??
Not sure, but anyway change the code
Run simulation
Looks OK???
yes
no
~~Yes, my code is OK, because the simulation result says "it is OK"~~
This is not true.

Test vectors definition (Cont'd)

Do not think "my code is correct, because the simulation result is OK"

Test vectors definition (Cont'd)



Check if code and state transition matrix is identical. If not, correct code

Carefully select test data

Run simulation

Exactly same to the expected result?

yes

Is there any possibility that any bugs get out of the test –data-bug-catch-net? "simulation result OK does not mean my code is OK"

no

Check state transition matrix and the result.

Code was wrong?

If yes, correct code. Otherwise correct matrix. Think why

Example1

```
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

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
```

Example1 (Cont'd)

```
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
```

Example2

```
module multiplexor4_1(out, in1, in2, in3, in4, cntrl1, cntrl2);
         output out;
        input in1, in2, in3, in4, cntrl1, cntrl2;
        reg out;


        always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
                case ({cntrl1, cntrl2})
                                2'b00 : out = in1;
                                2'b01 : out = in2;
                                2'b10 : out = in3;
                                2'b11 : out = in4;
                                default : $display("Please check control bits");
                endcase
endmodule
```

Example2 (Cont’d)

```
module muxstimulus (IN1, IN2, IN3, IN4, CNTRL1, CNTRL2, OUT);
    output IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;
     input OUT;
    reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;

    initial
      begin
            {CNTRL1, CNTRL2}=2’b00; {IN1, IN2, IN3, IN4} = 4’b1010; expected=1;
     #10 {CNTRL1, CNTRL2}=2’b01; {IN1, IN2, IN3, IN4} = 4’b1010; expected=0;
     #10 {CNTRL1, CNTRL2}=2’b10; {IN1, IN2, IN3, IN4} = 4’b1010; expected=1;
    #10 {CNTRL1, CNTRL2}=2’b11; {IN1, IN2, IN3, IN4} = 4’b1010; expected=0;

    end

     initial
       begin
           $display("Initial arbitrary values");
           #0 $display("input1 = %b, input2 = %b, input3 = %b, input4 = %b, control1 = %b,
    control2 = %b, output = %b, expected output = %b, time = %d\n",
                   IN1, IN2, IN3, IN4, CNTRL1, CNTRL2, OUT, expected, $time);
    end
endmodule
```

Example2 (Cont'd)

```
module testbench (a, b, c, d, s1, s2, f);
    wire a, b, c, d, s1, s2, f ;

    muxstimulus stim (.IN1(a), .IN2(b), .IN3(c), .IN4(d), .CNTRL1(s1),
    .CNTRL2(s2), .OUT(f));
    multiplexor4_1 mux (.in1(a), .in2(b), .in3(c), .in4(d), .cntrl1(s1), .cntrl2(s2),
    .out(f));

endmodule
```

Example3

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

Example3 (Cont'd)

```
module encoder8_3( encoder_out , enable, encoder_in );
      output[2:0] encoder_out;
      input  enable;
      input[7:0] encoder_in;
      reg[2:0] encoder_out;
      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
```

Sensitivity list.

Simple behavioral code using the case statement.

Example3 (Cont'd)

```
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
```

Example4

- Let’s have a look on a module named “priority” and its stimulus named “priority_sti”

priority.v:

```
module priority (A, P, F);
// Parameter definition
parameter N = 8;
parameter log2N = 3;
// Input/Output definition
input [N-1:0] A; //Input Vector
output [log2N-1:0] P; // High Priority Index
output F; // Found a one?
// Register definition
reg [log2N-1:0] P;
reg F;
always @(A) begin
            {P, F} <= priority(A);
end
```

```
//Function definition
        function [log2N:0] priority;
            input [N-1:0] A;
            reg F;
            integer I;
            begin
                F = 1'b0;
                priority = {3'b0, F};
               for (I=0; I<N; I=I+1)
                    if (A[I]) begin
                        F = 1'b1;
                        priority = {I, F};// Override previous
                                          //index
                    end
               end
         endfunction
endmodule
```

Example4 (Cont'd)

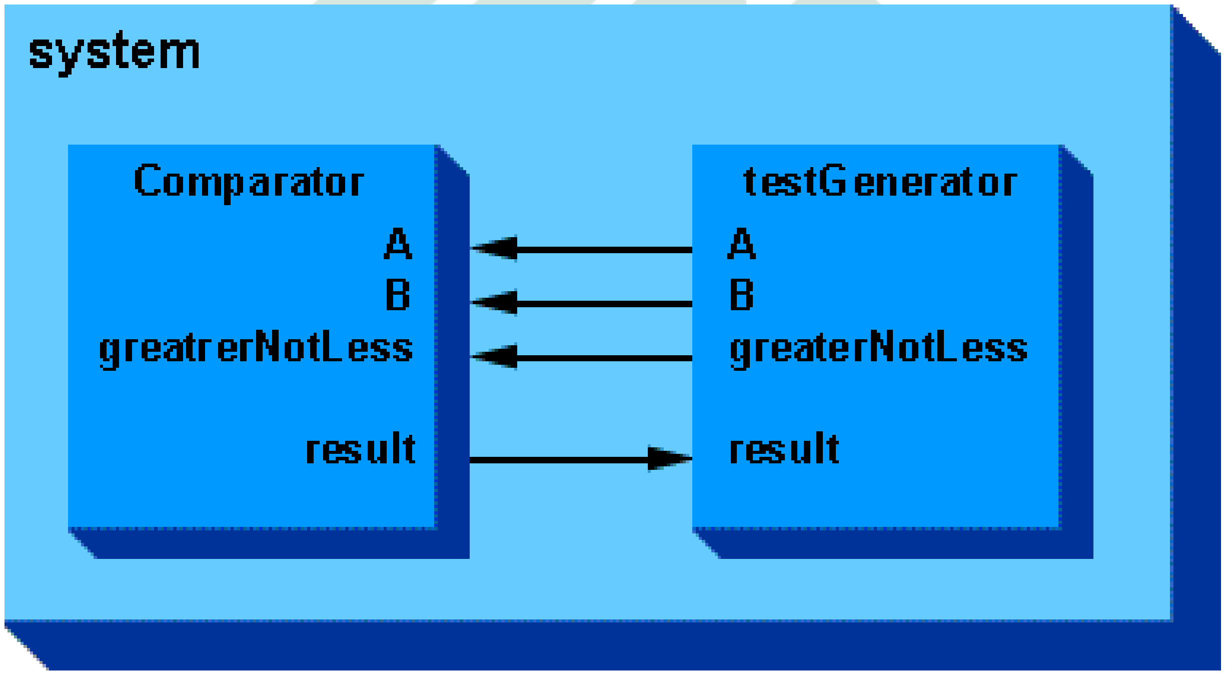
priority_sti.v:

```
`timescale 1ns/1ps
module stimulus;
    reg [7:0] Input;
    wire [2:0] Priority;
    wire Found_one;
    // Instance
    priority priority_inst (Input, Priority, Found_one);
    // Begin test
    initial begin
        test_pri(8'b01100100, 6, 1);
        test_pri(8'b01100100, 6, 0);
        test_pri(8'b01100100, 7, 1);
        test_pri(8'b01100100, 7, 0);
        test_pri(8'b00010010, 4, 0);
        test_pri(8'b00010010, 4, 1);
        test_pri(8'b00010010, 3, 0);
        test_pri(8'b00010010, 3, 1);
    end
```

Example4 (Cont'd)

```
        // Task definition
        task test_pri;
                input [7:0] in;
                input [2:0] pri;
                input f;
                begin
                #5 Input = in;
                #1;
                if((pri !== Priority)||(f !== Found_one))
                        $display("FAIL at time %0d, Input=%b, Priority=%d,
    Priority_exp=%d, Found_one=%b, Found_one_exp=%b\n",$time,in, Priority, pri, Found_one,
    f);
                else
                        $display("PASS at time %0d, Input=%b, Priority=%d,
    Priority_exp=%d, Found_one=%b, Found_one_exp=%b\n",$time,in, Priority, pri, Found_one,
    f);
                End
        endtask
    endmodule
```

Example5


system
Comparator
A
B
greatrerNotLess
result
testGenerator
A
B
greaterNotLess
result

Example5 (Cont'd)

Parameters that may be set when the module is instantiated.

Comparator makes the comparison A ? B where ? Is determined by the input greaterNotLess and returns true(1) or false(0).

```
module comparator (result, A, B, greaterNotLess);
    parameter width = 8;
    parameter delay = 1;
    input [width-1:0] A, B;           // comparands
    input greaterNotLess;              // 1 - greater, 0 - less than
    output result;                   // 1 if true, 0 if false

    assign #delay result = greaterNotLess ? (A > B) : (A < B);

endmodule
```

Example5 (Cont'd)

```
module testGenerator (A, B, greaterNotLess, result);
    output [15:0] A, B;
    output greaterNotLess;
    input result;
    parameter del = 5;
    reg [15:0] A, B;
    reg greaterNotLess;

    task check;
        input shouldBe;
        begin
            if (result != shouldBe)
                $display("Error! %d %s %d, result = %b", A, greaterNotLess?">":"<",
                          B, result);
        end
    endtask
```

Module that generates test vectors for the comparator and checks correctness of output.

Example5 (Cont'd)

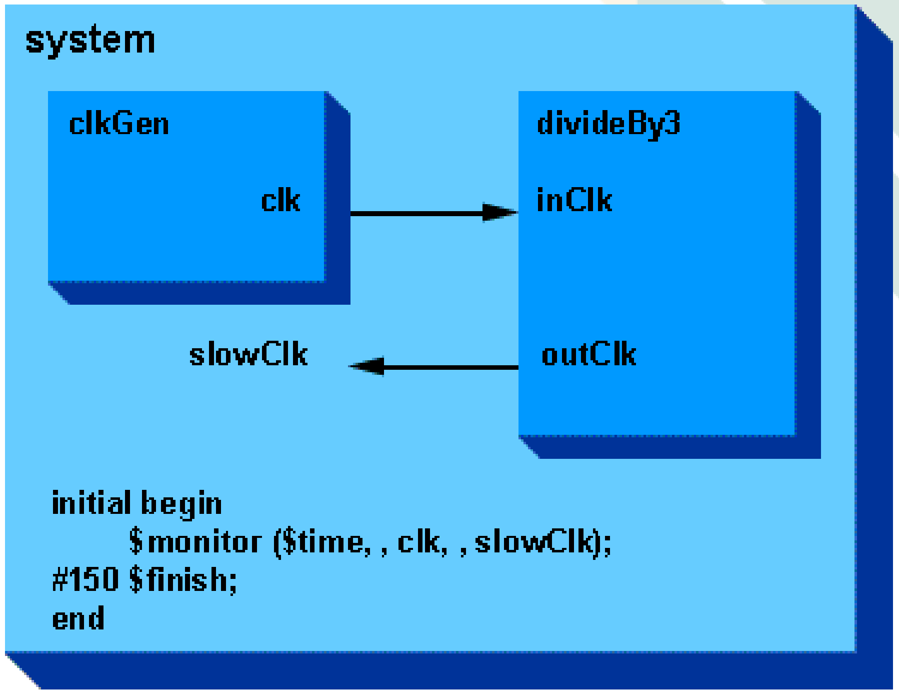
```
initial begin               // produce test data, check results
        A = 16'h1234;
        B = 16'b0001001000110100;   // 16'h1234
        greaterNotLess = 0;
    #del check(0);  ← Task call

        B = 0;
        greaterNotLess = 1;
    #del check(1);

        A = 1;
        greaterNotLess = 0;
    #del  check(0);
        $finish;
    end
endmodule
```

Example5 (Cont'd)

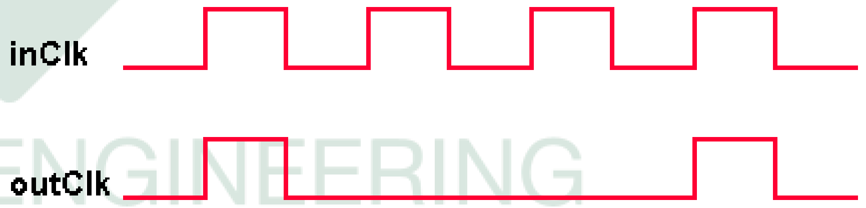
```
module system;
    wire greaterNotLess;        // sense of comparison
    wire [15:0] A, B;           // comparand values - 16 bit
    wire result;                // comparison result

    // Module instances
    testGenerator tg (A, B, greaterNotLess, result);
    comparator #(16, 2) comp (result, A, B, greaterNotLess);

endmodule
```

Example6

Divide-by-3 clock reducer. The input to the module is a reference clock, and the output is a clock signal which has a pulse every three cycles of the reference clock. The pulse width should be the same for both clocks.



```
initial begin
        $monitor ($time, , clk, , slowClk);
#150 $finish;
end
```



Output timing diagram

Example6 (Cont'd)

```
module divideBy3 (outClk, inClk);
    parameter delay = 1;
    input inClk;                  // reference clock
    output outClk;                  // stepped down  clock

    reg ff1, ff2, temp;          // local storage
    initial begin ff1 = 0; ff2 = 1; end

    assign outClk = ff2 & inClk; // output assignment

    always @(posedge inClk)
        begin
            temp = ff1;
            ff1 = ff2;
            ff2 = ~(temp | ff2);
        end
endmodule
```

Example6 (Cont’d)

```
module clkGen (clk);
    parameter period = 2;
    output clk;
    reg clk;

    initial clk = 0;              // start off with 0, so first edge is rising
    always                        // clock loop
        #(period/2) clk = ~clk;
endmodule
```

Example6 (Cont'd)

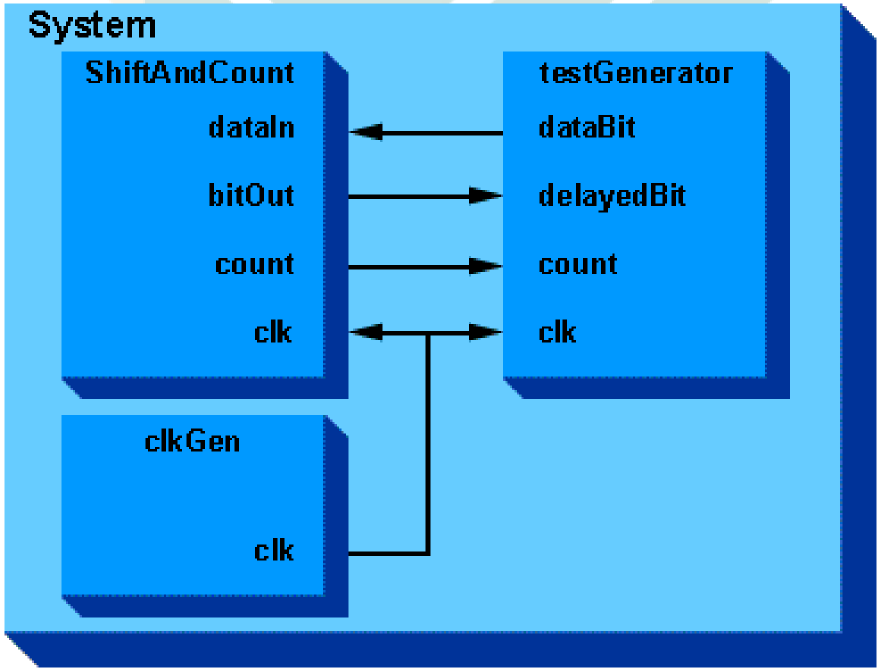
```
module system;
    wire slowClk, clk;      // two clocks

    // Module instances
    divideBy3 d3 (.outClk(slowClk), .inClk(clk));   // clock divider
    clkGen #(10) cg (clk);                          // clock generator

    initial begin
         $monitor ($stime, " clk: %b  slowClk: %b", clk, slowClk);
    #150 $finish;
    end

endmodule
```

Example7

Implement a shift and count register. A data value is shifted in on every clock cycle, on the rising edge, and the oldest value is shifted out. The shift register is fifo. Count the number of ones which are present in the shift register. The counter should be 32 bits wide (admitedly, this is overkill). The depth of the shift register should be parameterized.

The module ports are:

data input (1 bit), clock (1 bit), data output (1 bit), counter (32 bits).



Example7 (Cont'd)

```
module shiftAndCount (bitOut, count, dataIn, clk);
   parameter width = 8;
   output bitOut;                    // data shifted out
   output [31:0] count;              // count of ones
   input dataIn, clk;                // inputs
   integer count;                    // the counter
   reg bitOut;                       // temporary
   reg [width-1:0] lastBits;         // shift register

   initial begin count = 0; lastBits = 0; end

   always @(posedge clk) begin
      bitOut = lastBits[width-1];
      lastBits = (lastBits<<1) | dataIn;
      if (bitOut > dataIn)
         count = count - 1;
      else
      if (bitOut < dataIn)
         count = count + 1;
   end
endmodule
```

Follow the logic of this behavioral code with an example by hand.
Observe that initially lastBits = 00000000 and with each dataIn bit coming in from the right lastBits is shifted left so bitOut is the leftmost bit.

Example7 (Cont'd)

```
module clkGen (clk);
    parameter period = 2;
    output clk;
    reg clk;

    initial clk = 0;              // start off with 0, so first edge is rising
    always                        // clock loop
        #(period/2) clk = ~clk;
endmodule
```

Example7 (Cont’d)

dataIn	lastBits	bitOut	count
x	00000000	x	0
1	00000001	0	1
0	00000010	0	1
1	00000101	0	2
0	00001010	0	2

Example7 (Cont'd)

```
module testGenerator (dataBit, delayedBit, count, clk);
    output dataBit;
    input delayedBit;
    input [31:0] count;
    input clk;
    reg dataBit;

    initial begin              // produce test data, check results
        $monitor($time," dataBit: %b delayedBit: %b", dataBit, delayedBit);
        emitBits(0, 1);          // take care of first cycle
        emitBits('b10010, 5);
        check(0, 2);
        emitBits('b101101, 6);
        check(0, 5);
        emitBits('b01, 2);
        check(1, 5);
        $stop;
    end
```

Module generates test vectors and checks result.

Tasks: emitBits, check

Example7 (Cont'd)

```
task emitBits;  // helper task to emit n bits
      input [7:0] bits, n;         // task inputs
      begin
         repeat (n) begin        // assume clk is at negedge
            dataBit = bits[0];   // take just the low order bit
            bits = bits >> 1;
         @(negedge clk) ;
         end                      // leave at negative edge
      end
   endtask

   task check;
      input bit;
      input [31:0] shouldBe;
      begin
         if (delayedBit != bit)
            $display($time," delayed bit is %b but should be %b",
                        delayedBit, bit);
         if (count != shouldBe)
            $display($time," Count is %d but should be %d",
                        count, shouldBe);
      end
   endtask
endmodule
```

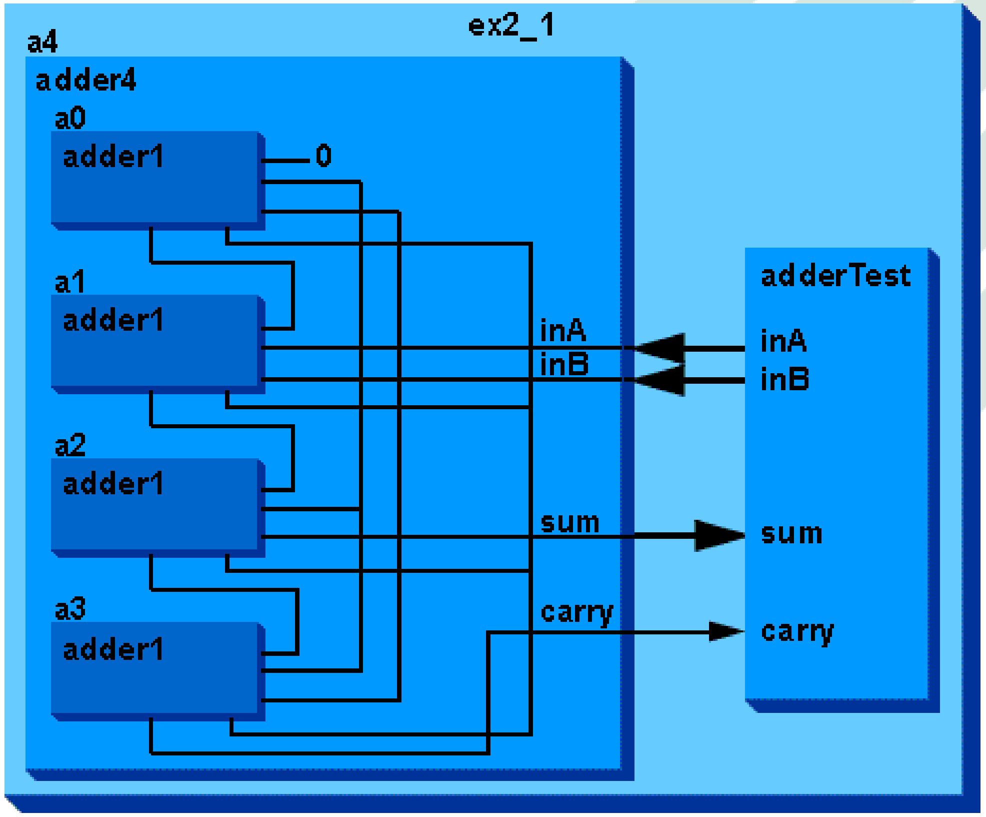
Task definitions.

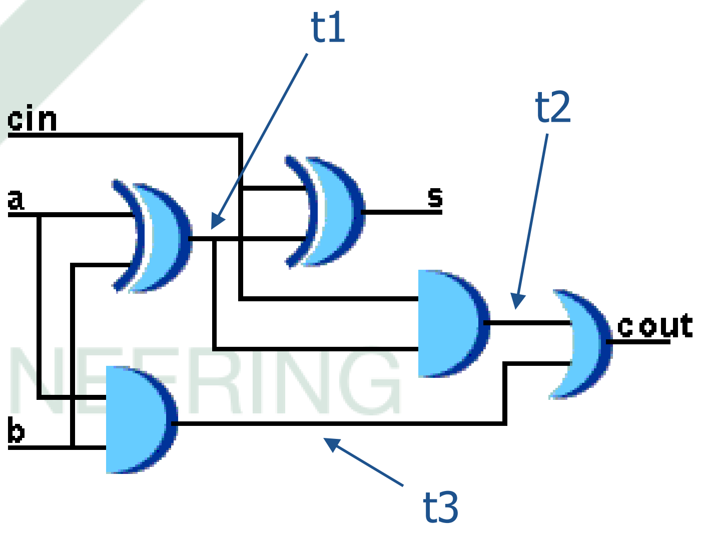
Example7 (Cont'd)

```
module system;
    wire data, clk;                // nets to connect up the pieces
    wire delayedData;              // data out of the fifo
    wire [31:0] nOnes;             // number of ones contained in fifo

    // Module instances
    shiftAndCount SandC (delayedData, nOnes, data, clk);    // shift register
    clkGen #(10) cg (clk);                                  // generate the clock
    testGenerator tg (data, delayedData, nOnes, clk);       // create data, check result

endmodule
```

Example8

4-bit adder. It takes two 4-bit operands, inA and inB, and produces a 4-bit result, sum, and a 1-bit carry. It is composed of four 1-bit adders, each of which has a carry in as well as the two operand inputs





Full 1-bit adder

Example8 (Cont'd)

```
module adder1 (s, cout, a, b, cin);
    output s, cout;
    input a, b, cin;

    xor (t1, a, b);
    xor (s, t1, cin);
    and (t2, t1, cin),
          (t3, a, b);
    or  (cout, t2, t3);
endmodule
```

1-bit full adder module. Refer the circuit diagram before.

Example8 (Cont'd)

```
module adder4 (sum, carry, inA, inB);
   output [3:0] sum;
   output carry;
   input [3:0] inA, inB;

   adder1 a0 (sum[0], c0,    inA[0], inB[0], 1'b0);
   adder1 a1 (sum[1], c1,    inA[1], inB[1], c0);
   adder1 a2 (sum[2], c2,    inA[2], inB[2], c1);
   adder1 a3 (sum[3], carry, inA[3], inB[3], c2);
endmodule
```

4-bit adder module composed of 4 1-bit adders modules. Structural code.

Example8 (Cont’d)

```
module adderTest (A, B, sum, carry);
    output [3:0] A, B;
    input [3:0] sum;
    input carry;
    reg [3:0] A, B;

    integer i, j;
    initial begin
       $monitor ("A: %d  B: %d sum: %d  carry: %d", A, B, sum, carry);
       for (i=0; i<16; i=i+1)
          for (j=0; j<16; j=j+1)
                begin
                   A = i;
                   B = j;
                #1 ;
                end
       $finish;
    end
endmodule
```

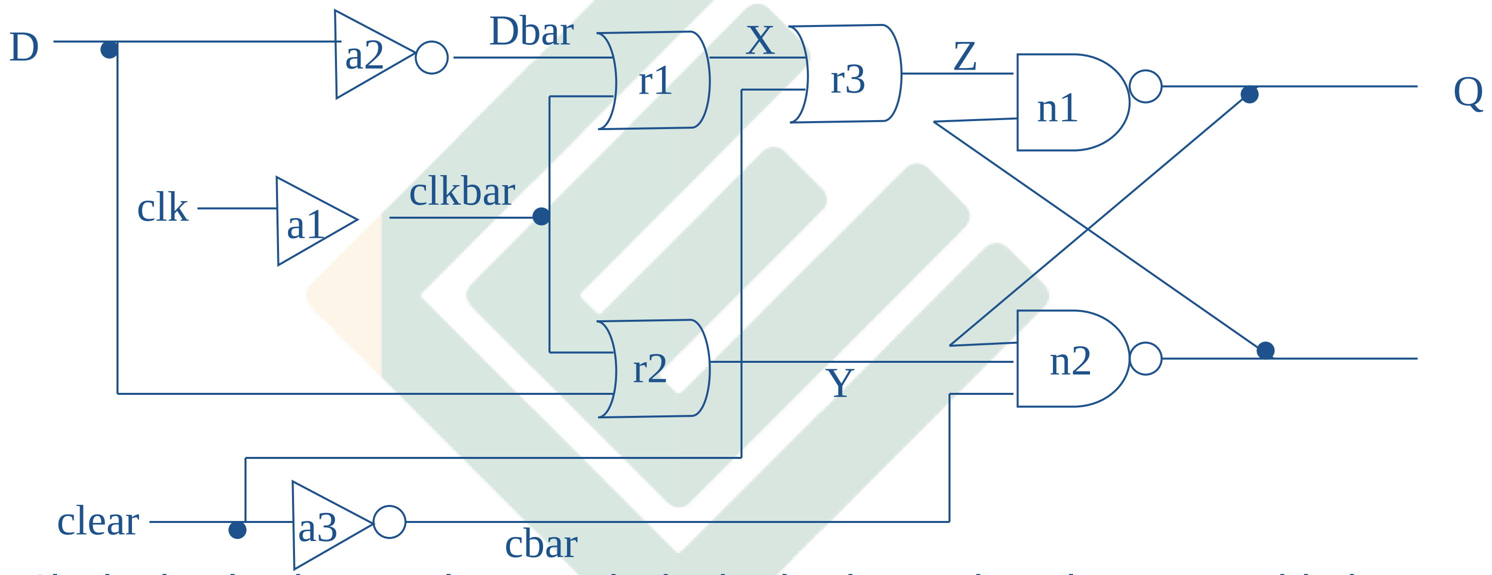
Module generates test vectors.

Example8 (Cont'd)

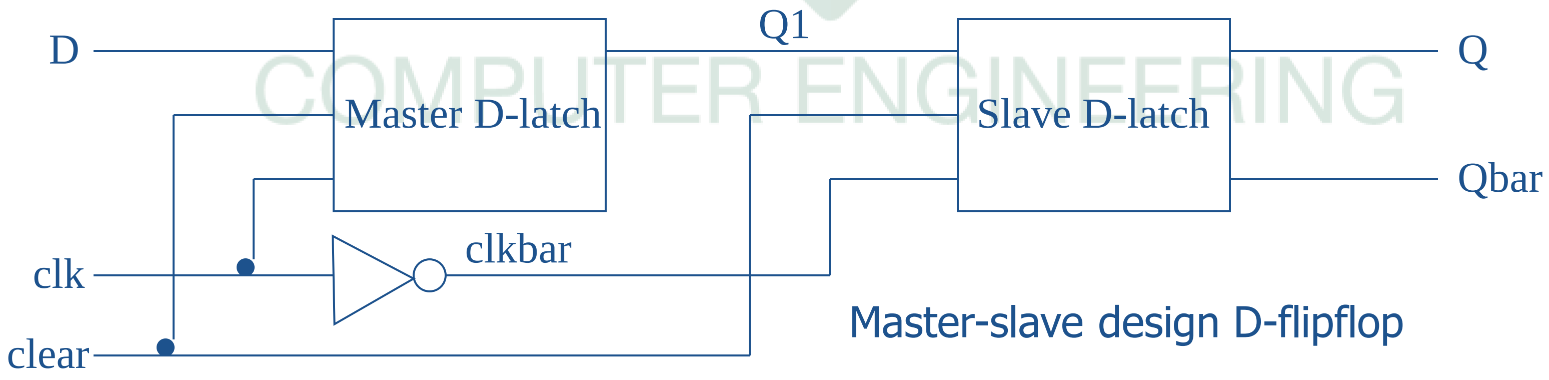
```
module ex2_1;
    wire [3:0] sum, inA, inB;
    wire       carry;

    adder4 a4 (sum, carry, inA, inB);
    adderTest at (inA, inB, sum, carry);
endmodule
```

Stimulus.

Example9



Clocked D-latch, exactly as in clockedD_latch.v with a clear wire added



Master-slave design D-flipflop

Example9 (Cont'd)

```
// Clocked D-latch as in clockedD_latch.v with a clear signal added
module clockedD_latch(Q, Qbar, D, clk, clear);

output Q, Qbar;
input D, clk, clear;
wire X, Y, Z, clkbar, Dbar, cbar;

not a1(clkbar, clk);
not a2(Dbar, D);
not a3(cbar, clear);
or r1(X, Dbar, clkbar);
or r2(Y, D, clkbar);
or r3(Z, X, clear);
nand n1(Q, Z, Qbar);
nand n2(Qbar, Y, Q, cbar);
endmodule

// Negative edge-triggered D-flipflop with 2 D-latches in master-slave relation
module edge_dff(q, qbar, d, clk, clear);

output q, qbar;
input d, clk, clear;
wire q1;

clockedD_latch master(q1, , d, clk, clear); // master D-latch
not(clkbar, clk);
clockedD_latch slave(q, qbar, q1, clkbar, clear, writeCtr); // slave D-latch
endmodule
```

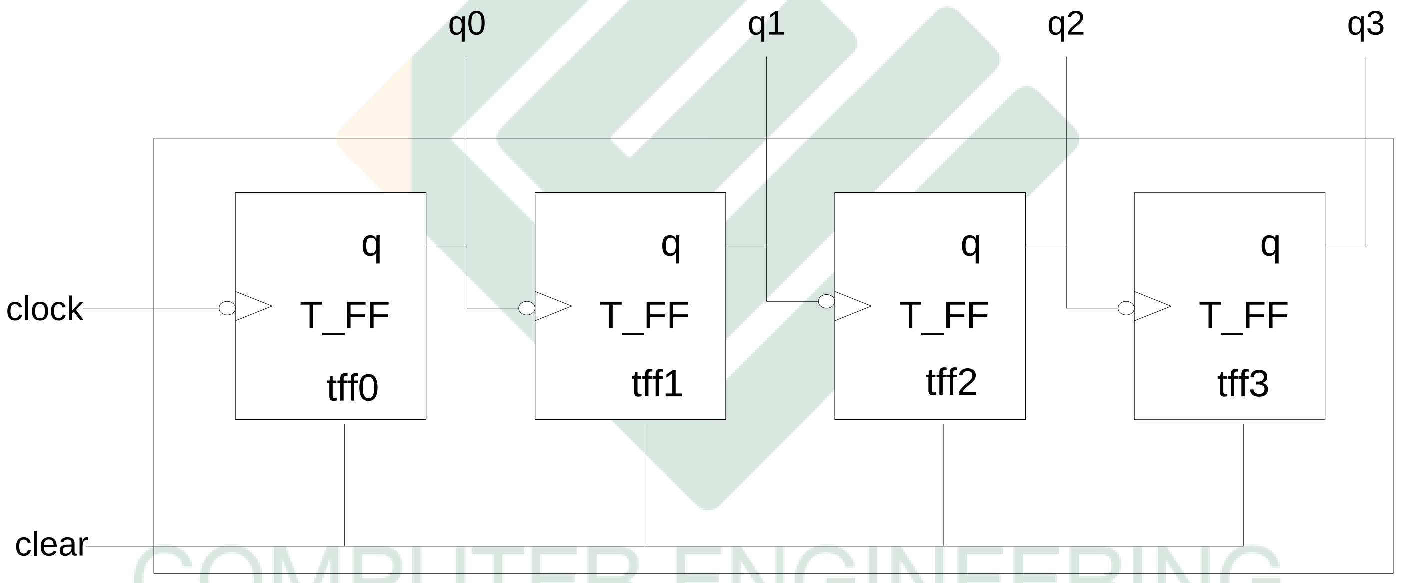
Use same stimulus code as edge_dffGates.v!

Example9 (Cont'd)

```
module Top;
wire q, qbar, clk;
reg d;

edge_dff ff1(q, qbar, d, clk, clear);
clockGenerator cg(clk);

initial
begin
      clear = 0;
  #2 d = 0;
  #0 $display(" time = %d, clk= %b, d = %b, q= %b\n", $time, clk, d, q);
  #4 d = 1;
  #0 $display(" time = %d, clk= %b, d = %b, q= %b\n", $time, clk, d, q);
  #2 d = 0;
  #0 $display(" time = %d, clk= %b, d = %b, q= %b\n", $time, clk, d, q);
  #4 d = 1;
  #0 $display(" time = %d, clk= %b, d = %b, q= %b\n", $time, clk, d, q);
  #10 d = 0;
  #0 $display(" time = %d, clk= %b, d = %b, q= %b\n", $time, clk, d, q);
end

initial
begin
   #100 $finish;
end
endmodule
```

Explain output!

Example10



4-bit ripple counter made from a series of T-flipflops

Example10 (Cont'd)

```
module counter(Q , clock, clear);
output [3:0] Q;
input clock, clear;

// Instantiate the T flipflops
t_ff tff0(Q[0], clock, clear);
t_ff tff1(Q[1], Q[0], clear);
t_ff tff2(Q[2], Q[1], clear);
t_ff tff3(Q[3], Q[2], clear);

endmodule
```

Example10 (Cont'd)

```
module stimulus;
reg CLOCK, CLEAR;
wire [3:0] Q;

initial
	$monitor($time, " Count Q = %b Clear= %b",
	Q[3:0],CLEAR);

counter c1(Q, CLOCK, CLEAR);

initial
begin
	CLEAR = 1'b1;
	#34 CLEAR = 1'b0;
	#200 CLEAR = 1'b1;
	#50 CLEAR = 1'b0;
end
```

```
initial
begin
		CLOCK = 1'b0;
		forever #10 CLOCK =
~CLOCK;
end

initial
begin
		#400 $finish;
end
endmodule
```

Timing check

- A timing check is a system task that performs the following steps:
  - ➢Determines the time between two events
  - ➢Compares the elapsed time to specified minimum or maximum time limits
  - ➢Reports a timing violation whenever the elapsed time occurs outside the specified time limits.

Timing check

- **Using Timing Check:**

  To verify the timing characteristics of your design, you can invoke the following timing check system tasks in specify blocks:

  $hold(<clk_event>, <data_event>, <hold_limit>{, <notifier>});

  $period(<clk_event>, <period_limit> {, <notifier>});

  $setup(<data_event>, <clk_event>, <setup_limit>{, <notifier>});

  $skew(<clk_1>, <clk_2>, <skew_limit> {, <notifier>});

  $width(<edge_clk>, <min_limit> {,<threshold>{, <notifier>}});

  ...

Timing check (Cont'd)

- $hold

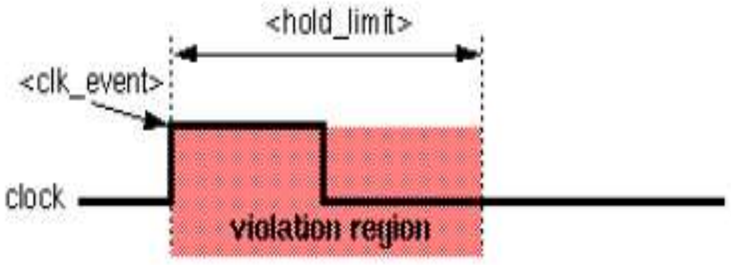
  The $hold system task determines whether a data signal remains stable for a minimum specified time after a transition in an enabling signal, such as a clock signal that latches data in a memory.

  The $hold system task has the following syntax:

  ```
          $hold(<clk_event>, <data_event>, <hold_limit> {,
  <notifier>});
  ```

  The following example illustrates how to use the $hold system task:

  ```
  specify
          specparam hold_param=1
          $hold( posedge clk, data, l
          $hold( posedge clk, data, l
  endspecify
  ```



Timing check (Cont’d)

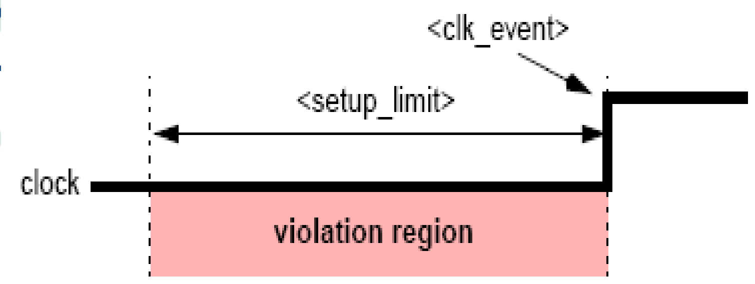
- $setup

  The $setup system task determines whether a data signal remains stable before a transition in an enabling signal, such as a clock signal that latches data in memory.

  The $setup system task has the following format:

  ```
  $setup(<data_event>, <clk_event>, <setup_limit> {, <notifier>});
  ```

  In the following example, the $setup system task reports a violation if the interval from <data_event> to <clk_event> is less than <setup_limit> (10). The second specification of the $setup sy
  notifier, flag, to indicate a timing violation.

  ```
  specify
      specparam setup_param=10;
      $setup( data, posedge clock, setup_
      $setup( data, posedge clock, setup_param, flag ) ;
  endspecify
  ```

Timing check (Cont'd)

- $width

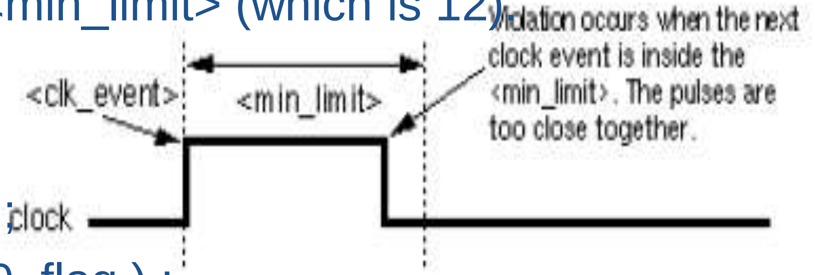
  The $width system task specifies the duration of signal levels from one clock edge to the opposite clock edge. A violation occurs when signals are too close together. If you use edge specifiers, all edges must be of the same direction.

  The syntax for the $width system task is as follows:

  $width(<edge_clk>, <min_limit> {, <threshold>{, <notifier>}});

  The following example shows how to use the $width system task to report a violation if the interval from <edge_clk> (negedge clr) to the implicit <data_event> (posedge clr) is less than <min_limit> (which is 12).

```
specify
        specparam width_param=12;
        $width( negedge clr, width_param ) ;
        $width( negedge clr, width_param, 0, flag ) ;
endspecify
```


Violation occurs when the next clock event is inside the <min_limit>. The pulses are too close together.
<clk_event>
<min_limit>
clock

Timing check - Example

dff.v:

```
module dff (D, Q, CLK);
        input D;
        input CLK;
        output Q;
        reg Q1;
        wire Q = Q1;
        always @(posedge CLK)
                Q1 <= D;
        specify
                specparam t_setup = 5;
                specparam t_acc = 10;
                specparam t_pwh = 20;
                specparam t_pwl = 20;
                $setup( D, posedge CLK, t_rise_setup);
                $width(edge[01,x1]CLK,t_pwl);
                $width(edge[10,1x]CLK,t_pwh);
                (CLK => Q) = t_acc;
        endspecify
endmodule
```

Timing check – Example (Cont’d)

dff_sti.v:

```
`timescale 1ns/1ps
module stimulus;
reg din, clock;
reg exp;
wire dout;
parameter clk_cycle = 60;
// Instance DFF
dff instance_1 (.D(din), .), .Q(dout));
// Clock generator
always
   begin
   #(clk_cycle/2) clock = ~clock;
   end
```

Timing check – Example (Cont’d)

```
// Input waveform and Expected output
initial
begin
    clock   = 1'b0;
    din     = 1'b0;
    fork
                    #24 din = 1'b1;
                    #41 exp = 1'b1;
                    #84 din = 1'b0;
                    #101 exp = 1'b0;
                    #144 din = 1'b1;
                    #161 exp = 1'b1;
                    #206 din = 1'b0;
                    #221 exp = 1'b0;
                    #250 $finish;
              join
end
```

Timing check – Example (Cont'd)

```
//Compare output
    always @(exp)
    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
```