The Verilog Hardware Description Language – Testing the Design

Overview

In this lesson we will

- ✓ Move from design to test
- ✓ Introduce the test bench
- ✓ Examine several of the system tools that support testing
- ✓ Learn how to build a test module
- ✓ Identify the critical elements of a test module
- ✓ Introduce testing of combinational and sequential circuits

Testing and Verifying the Circuit

- Once the circuit is designed and modeled in Verilog
  - We move into the next phase
- First
  - Need to verify that the model functions properly
- Next step is to use it for its intended purpose
  - To that end we perform any necessary
    - Functional, parametric, and stress tests on the design
      - Confirm the design before committing to hardware

- At this point
  - Will review general structure for
    - Test bench
    - Tester
- We will illustrate the verification phase
  - Using the *NandNor* circuit
- To do this we create a *test bench*

The Test Bench

- Test bench models the electronics workbench
- Electronics workbench
  - Comprises the measurement and stimulus instruments
  - Circuit to be tested
    - The UUT
      - Needs to be single top-level module
- Verilog test bench

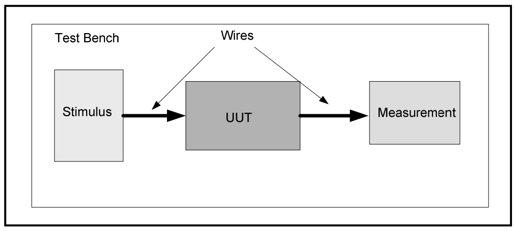
Comprises modules used for stimulus and measurement

These go in a test module

System modeled in Verilog

The UUT

Following diagram gives schematic representation



High-level pseudocode model for the test bench

Has the following general structure

```
module MyTest bench;
   parameter declarations
   wires

   circuit module declarations
   test module declaration
endmodule
```

Test bench plays the same role as does

main() function

In C or C++ and t

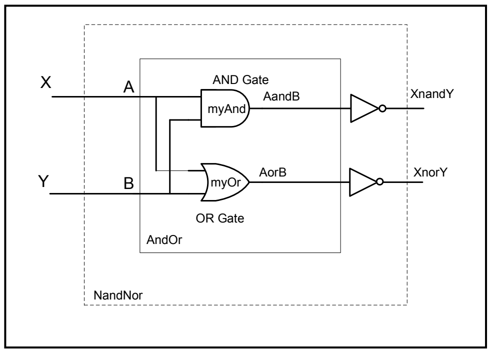
Top level class

In Java

Acts as the outermost container in the program

Let’s look at the pieces

The Unit Under Test



The unit under test or UUT

Is system, subsystem, module

That we are testing

This is our design

We will use the gate level *NandNor* circuit as the UUT

To be tested and verified

During different phases of development life cycle

Testing for different reasons
Formulating and executing
    Different kinds of tests
At any stage
    Must have complete and accurate specification

- Early stages confirming
    - High level functionality
    - Overall behaviour of design
        - Behavioural model works well

- Middle stages testing
    - Data and control flow through system
        - Dataflow / RTL model works well
    - Timing
    - Mainly high level but will examine
    - Critical low-level timing aspects

- Later stages verifying
  It performs according to specification
    - Normal operation
    - Boundary operation
        - Inside, at, and outside boundaries

  Each path through the logic circuit is functional
  Timing
    - All high and low-level timing constraints met

System Tasks and Functions

As we bring the tester together we find
    Verilog language provides standard system tasks
        To aid in performing routine operations

Let’s look at several of these

Time

- Verilog simulations executed with respect to *simulation time*
- Value stored in special register data type
  - Time variable declared using keyword *time* data type
- Value of time variable can be retrieved
  - Using system function $time

```
//  declare variable to hold current time
time simTime;
initial
   begin
      simTime = $time;
   end
```

Displaying Information

- Given purpose of developing Verilog program
  - Model a design
    - To confirm that it conforms to specification
- During test of module under design and when test complete
  - Want to be able to
    - Annotate results
    - Observe values of
      - Input and output signals
      - Intermediate and final values of internal variables

$display and $monitor statements

- The *$display* and *$monitor* are standard system tasks
  - Enable one to see the states of certain signals in text form
- Output is typically directed to the screen (or window)
- Difference between the two statements is
  - *$display*
    - Only evaluated when the directive is encountered during execution
  - *$monitor*
    - Evaluated every time *any* of the signals being monitored changes state

Syntax for the two directives is given as

```
Syntax
    $display (["formatrString"], variableList);

    $monitor (["formatString"], variableList);
```

The *formatString*

Optional for both statements

Both follow the C *printf*

Is a text string containing format variables that are to be instantiated

From the values specified in *variableList*

More commonly used format variables given in following table.

Format Variable	***Display***
%b	Binary
%d	Decimal
%o	Octal
%h	Hexadecimal
%c	Character

By convention

Logic *high* is denoted as a 1

Logic *low* is denoted as a 0

Unknown state is denoted as an x

$display and *$monitor* output statements

Must be placed within an *initial* or *always* block

Stopping and Finishing

There are several system tasks

Can be used to terminate or temporarily suspend a simulation

These are the

$stop and *$finish* statements

As names suggest

Used to either stop or finish a simulation

$stop directs the simulation to the interactive mode

Used when the designer wishes to suspend the simulation prior to exit

To examine the state of signal values

$finish terminates the simulation.

The syntax for the two directives is given as

```
Syntax
    $stop;
    $finish;
```

Time

Verilog simulations executed with respect to *simulation time*

Uses built in variable

- Value stored in special register data type

Time variable declared using keyword *time* data type

Value of time variable can be retrieved

- Using system function $time
  - Returns the current time
- Syntax is given as,

```
Syntax
    $time
```

Can be included in a $display or $monitor statement as

```
Syntax
    $display ($time,  ["formatString"], variableList);
    $monitor ($time, ["formatString"], variableList);
```

```
//  declare variable to hold current time

time simTime;

initial
   begin
      simTime = $time;
   end
```

Simulated Time

Important to note

- Simulated time is simulation of
  - Actual time required for design to run when implemented
  - Although value in simulation
    - Has direct relation to physical (in fabricated system)
    - Is not measured in seconds
      - Implemented as unitless integer

- Common to map or interpret units as nanoseconds
- When modeling design
  - Must think in terms of simulation time
  - Implication
    - Several Verilog statements may be executed
      - Without $time advancing

Sequence

- Important to distinguish
  - Sequence and $time
- Consider following code fragment
  - Note two initial blocks running in parallel
  - Assignments to both variables – a and b
    - Will occur at simulation time 0

```
module SimTime0;

  integer a,b;

  initial
    begin
        a = 1;
        $display("a is %d", a, $time);
    end

  initial
    begin
        b = 2;
        $display("b is %d", b, $time);
    end
endmodule
```

Time Control -

We used # symbol when modeling prop delay through part

Can also utilize to control when statements executed

Modifying above example

```
module SimTime1;

  integer a,b;

  initial
    begin
        #4
        a = 1;
        $display("a is %d at $time = %d", a, $time);
    end

  initial
    begin
        #3
        b = 2;
        $display("b is %d at $time = %d ", b, $time);
    end
endmodule
```

Code fragment will evaluate *b* prior to *a* by one $time unit

We have forced the unambiguous evaluation order

Event Control - @

We can utilize event specified by @ symbol

To control when statements executed

Examine two cases

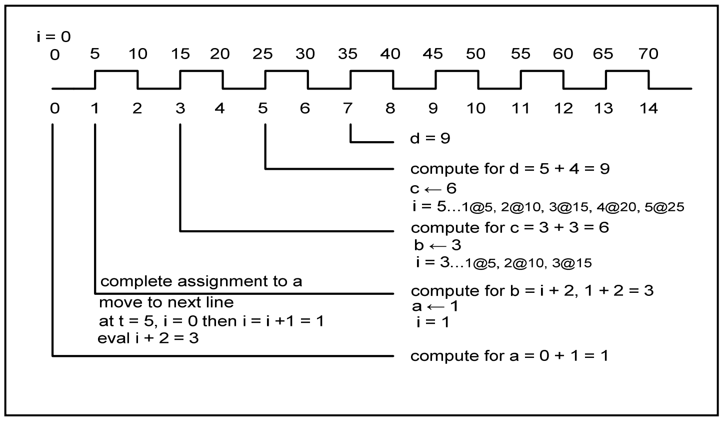
Blocking - =

Non-blocking <=

Modifying above code fragment

To utilize blocking assignment

```
module SimTime2;
   integer a,b, c, d;
   integer i;
   reg sysClk;
   parameter delay = 5;

   //  initial blocks executed in parallel

   initial
      begin
         sysClk = 0;                        //  blocking
         i = 0;
      end

   always
      begin
         #delay sysClk = ~sysClk;     //  blocking → delay then eval
         i = i + 1;
       end

//  blocking assignment
   initial
   begin
      a = @(posedge sysClk) i+1;                       //  i = 0    eval RHS then wait for rising edge
      $display("a is %d at $time = %d", a, $time);

      b = @(posedge sysClk) i+2;                          //  i = 1    from t = 5, but now i ← 2
      $display("b is %d at $time = %d", b, $time);

      c = @(posedge sysClk) i+3;                          //  i = 3    from t = 5, but now i ← 2
      $display("c is %d at $time = %d", c, $time);

      d = @(posedge sysClk) i+4;                          //  i = 5    from t = 5, but now i ← 2
      $display("d is %d at $time = %d", d, $time);

      #(40*sysClk)
      $stop;
      $finish;
   end

endmodule
```



Test Module

Let’s now move on to the tester or test module

This is most critical element in test process

Test module will have

- Initialization sequence
  - To place UUT into known state
    - If start test from unknown state
      - Cannot make any statement about its behaviour
  - This is an essential step
- A set of inputs and a set of outputs
  - Inputs to the test module
    - Will be the outputs of the UUT
    - Will model the measurement equipment
  - Outputs from the test module
    - Will be the inputs of the UUT
    - These will model the stimulus equipment
- Set of test vectors
  - These will be outputs of test module
  - Give careful thought to selecting these
  - Will be
    - Individual vectors
    - Sets of vectors
  - Provide known signals or patterns into UUT
- Set of known responses
  - Will be outputs of UUT
    - Known responses to applied stimuli

Combinational Logic – A First Look

The tester module for the *NandNor* combinational logic

Given in following code fragment

Opening lines of the test module

Identify the sets of *inputs* and *outputs*

These signals will

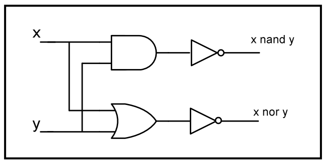
Come from the UUT

Send stimulus vector to the UUT

Following declaration of inputs and outputs

Find definition of *reg* type

```
module Tester (X, Y, XnandYin, XnorYin);
   input          XnandYin, XnorYin;
   output         X, Y;

   reg            X, Y;
   parameter      stimDelay = 10;

   initial        //  set initial conditions
   begin
      x = 0; y = 0;
   end

   initial        // Stimulus – Test Vectors
   begin
                  X = 1; Y = 1;
      #stimDelay  X = 0;
      #stimDelay  Y = 0;
      #stimDelay  X = 1;
   end

   initial
   begin          // Response
      display("\t Time, \t \tX, \t Y, \t XnandYin, \t XnorYin");
      monitor($time, "\t \t %b, \t %b, \t %b, \t \t%b", X, Y, XnandYin, XnorYin);
   end
endmodule
```



There are two signal types in test bench

Used to drive and monitor signals

During the test of the UUT

These two types of signals are *reg* and *wire* types

The *reg* data type holds a value

Until a new value is driven onto it

In an initial or always block.

The reg type

Can only be assigned a value in an always or initial block

Is used to apply stimulus to the inputs of UUT

The *wire* data type is a passive data type

Holds value driven on it by

Port, assign statement or reg type

Wires can not be assigned values

Inside always and initial blocks

Wires not used in this design

- Following reg declaration
  - Parameter *stimDelay* specifies
    - Delay between the applications of successive test vectors
- *Initial* block declared next
  - Initial blocks start executing sequentially at simulation time 0
    - Starting with the first line between begin end pair
    - Each line executes from top to bottom
      - Until a delay is reached
  - When / if a delay is reached
    - Execution of this block waits
      - Until the delay time has passed and then picks up execution again.
  - Each initial and always block executes concurrently
    - So if a block is stalled
      - Other blocks in the design would execute

- If *always* block included
  - An always block does not continuously execute
    - Instead only executes on change in items in the sensitivity list
      - For example posedge clock or negedge reset
      - Recall events discussed earlier
  - Means when there is a
    - low to high on the clock signal or
    - high to low on reset
  - always block will execute.

- Next test vectors
  - Defined and appear as successive statements
- Four different combinations of X and Y
  - Applied to the circuit input
- Delay is specified between each stimulus application

- Design of the *NandNor* circuit assumes ideal parts
  - Had the logic gates included a delay
  - The *stimDelay* between the applications of successive vectors
    - Would have provided time for the signal to propagate

Through the logic block.

Initialization and test vectors

Written as statements within *initial* blocks

Thus the test suite is applied one time during the simulation

The circuit output in response to the set of test vectors

Presented using the *$display* and *$monitor* system tasks

$display is used to print to a line and enter a carriage return at the end

Variables can also be added to the display

Format for the variables can be

Binary using %b

Hex using %h

Decimal using %d

Another common element used in $display is $time

which prints the current simulation time

To monitor specific variables or signals in a simulation

Every time one of the signals changes value

A $monitor can be used

Only one $monitor can be active at a time in a simulation

But it can prove to be a valuable debugging tool

Sequential Logic – A First Look

Tester for the behavioral sequential *two-bit binary counter* module

Follows the same pattern with several additions

Presented in the code module in following figure

Tester designed to

Reset system

Apply 8 clock pulses to UUT

```
// Test module for two bit binary up counter
module tester(clr, clk, qA, qB);
    input qA, qB;
    output clr, clk;

    reg clk, clr;

    parameter stimDelay = 15;
    parameter clkDelay = 5;

    initial
    begin
        clk = 0;
        clr = 0;
        #stimDelay clr = ~clr;

        repeat(16)
        begin
            #clkDelay clk = ~clk;    //  generate 8 clock periods … 16 transitions
        end
    end

    initial
    begin
        $display("\tTime, \t\tqA, \tqB, \tclr, \tclk");
        $monitor($time,"\t\t%b, \t%b, \t%b, \t%b", qA, qB, clr, clk);
    end

endmodule
```

Clocks and Resets

Synchronous sequential circuit will need

Strobe, enable, or clock in order to operate

As was seen in previous code fragment

Specific number of clock pulses supplied to UUT

Generally test does not have such restrictions

Good designs also include a reset or clear signal

To establish the initial state of the circuit

Typically these signals are supplied

By the tester with a block of code such as

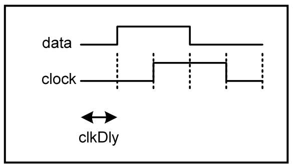
Code fragment in accompanying figure

When designing test vector suite for FSM

```
reg clk, clr;

parameter stimDelay = 15;
parameter clkDelay = 5;

initial
begin
    clk = 0;
    clr = 0;
    #stimDelay clr = ~clr;

    always
        #halfPeriod clk = ~clk;
end
```

Ensure that clock and data synchronized such as in code fragment in following figure

```
reg clk, data;

parameter clkDly = 5;

initial
begin
    clk = 0;
    data = 0;
end

//  generate the clock
always
    #(2*clkDly)clk = ~clk;

initial
begin
    #clkDly data = 1;

    #(2*clkDly)data = 0;
end
```



The Test Bench

Now bring everything together with test bench

In the test bench we instantiate

- One copy of the UUT
    - Here will be NandNor gate,
- One copy of the Tester
    - These are the stimulus and monitoring instruments

Finally, we connect them together using wires

As illustrated in the code fragment in following figure

```
module MyTest bench;
    wire XnandY, XnorY, X, Y;
    NandNor aNandNor (XnandY, XnorY, X, Y);
    Tester aTester (X, Y, XnandY, XnorY);
endmodule
```

Performing the Simulation

If the simulation is now run

- Test vectors are successively applied to the input of the UUT

As the simulation executes

- $monitor system task will display
  - State of the input and output signals
  - System time at which the samples were taken

These appear in following figure

```
Time,        X,     Y,       XnandYin,   XnorYin
0            1,     1,          0,          0
10           0,     1,          1,          0
20           0,     0,          1,          1
30           1,     0,          1,          0
```

If the behavioural results are satisfactory

- Can move on to real work of confirming the design
- We do this by utilizing dataflow or structural models
- Incorporate real world affects and issues

Summary

In this lesson we

- ✓ Moved from design to test.
- ✓ Introduced the test bench.
- ✓ Examined several of the system tools that support testing.
- ✓ Learned how to build a test module.
- ✓ Identified the critical elements of a test module.
- ✓ Introduced testing of combinational and sequential circuits.