Lab 0
Writing basic Testbenches and Introduction to Synopsys VCS

Module ID : CX-301

Design Verification

Instructor : Dr. Abid Rafique

Version 1.2

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Jan, 2024	Qamar Moavia
1.1	Modified with new exercises	feb, 2025	Qamar Moavia

Table of Contents

Objectives

The objective of this lab is to

- To familiarize yourself with Synopsys VCS
- To use SystemVerilog procedural constructs to model a simple register.
- To understand and apply provided directed and randomized self checking testbenches for the simple register design.
- To develop and apply directed and randomized testbenches for an ALU design using SystemVerilog.

Tools

- SystemVerilog
- Synopsys VCS

Instructions for Lab Tasks

The required files for this lab can be found on the

```
share_folder/CX-301-DesignVerification/Labs/Lab0
```

Each lab task must be placed in its own directory. The submission must be a zip folder with the following hierarchy as given below, with the folder and the file names exactly as listed below.

```
./Lab0/
      ├── Task1/
      │    ├── design/
      │    │    ├── register.sv
      │    ├── testbench/
      │    │    ├── register_directed_test.sv
      │    │    ├── register_random_test.sv
      │    ├── // screenshots of outputs/waveforms are must
      ├── Task2/
      │    ├── design/
      │    │    ├── alu.sv
      │    ├── testbench/
      │    │    ├── alu_directed_test.sv
      │    │    ├── alu_random_test.sv
      │    ├── // screenshots of outputs/waveforms are must
```

Along with that you also need to upload your solution on the github as well, and share the link.

1. Basic SystemVerilog for Verification

In this lab, we'll cover some basic SystemVerilog concepts. These are the building blocks for creating effective testbenches and simulations.

1.1 SystemVerilog Testbench Modules

SystemVerilog testbench top module is built using module constructs. A simple module looks like this:

```
module testbench;
    initial begin
        $display("Hello, SystemVerilog!");
    end
endmodule
```

- `initial` blocks execute once at the start of simulation.
- `$display` prints output to the terminal.

Compiling and Running Simulation using Synopsys VCS

To compile and run your testbench using synopsys vcs you need to do the following:

First compile the testbench using the following command:

```
vcs -full64 -sverilog -timescale=1ns/10ps testbench.sv -o simv
```

Then run the simulation by executing the binary file simv:

```
./simv
```

Note: Make sure you successfully run the above example first before moving forward to the next part of the lab.

1.2 Simulation Control Commands

Following are the simulation control commands that can be used to control the flow of simulation:

- `$stop;` pauses the simulation, allowing debugging before continuing.
- `$time;` returns the current simulation time.
- `#10;` Introduces a delay of 10 time units (based on the timeunit setting).
- representation as 100 picoseconds.
- `wait(~rst);`: Halts execution until the `rst` is dropped (deserted).
- `$finish;` finishes the simulation.

To specify the length of a single time unit and minimum precision for your simulation, use the following directives:

- `timeunit 1ns;` Sets the time unit for simulation to 1 nanosecond.
- `timeprecision 100ps;` Defines the precision of the time

```
module simple_module;
    timeunit 1ns;
    timeprecision 100ps;
    initial begin
        wait(~rst);               // Wait for reset signal to drop
        #10;                      // Wait for 10 time units
        $display("Current time: %t", $time); // Display current simulation time
        @(posedge clk);           // Wait for positive clock edge
        $stop;                    // Pause for debugging
        $display("Current time: %t", $time); // Display current simulation time
        $finish;                    // finishes the simulation
    end
endmodule
```

1.3 Generating Random Numbers

When testing designs, **random inputs** help ensure thorough verification. SystemVerilog provides built-in functions for generating random numbers:

$random (Signed 32-bit Random Number)

- Generates a **signed 32-bit** random number.
- **Not preferred** due to inconsistencies across simulators.

Example:

```
int random_number = $random;
$display("Random number: %d", random_number);
// outputs
Random number: -1456363
```

To ensure different results each run, we can use the following flag while running the simulation:

```
./simv +ntb_random_seed_automatic
```

The above statement ensures that the tool uses a different seed (random seed) on each run.

$urandom (Unsigned 32-bit Random Number)

- Generates an **unsigned 32-bit** random number.

Example:

```
int positive_random_number = $random;
$display("Random number: %d", positive_random_number);
//output
Random number: 647934
```

$urandom_range(min, max) (Random Number in a Range)

- Generates a random number between min and max.
- The order of min/max does not matter.

Example:

```
int random_num1 = $urandom_range(10, 1);
int random_num2 = $urandom_range(1, 10);
$display("Random numbers between 1 and 10: %d and %d",
random_num1, random_num2);
// output
Random numbers between 1 and 10: 7 and 3
```

1.4 SystemVerilog Functions & Tasks

SystemVerilog Functions:

SystemVerilog functions are similar to C functions. They allow you to group code that can be reused multiple times. Functions can take inputs and return a single value. They execute in zero simulation time and cannot include time delays, such as `#10;` or `@(posedge clk)`.

Key Points:

- Functions return a value using `return`.
- Void functions do not return a value.
- They execute instantly without simulation time delays.

Example:
This example demonstrates a function that calculates the **parity** (even or odd number of 1's) of an 8-bit number:

```
module test;

    // Function to calculate parity (even or odd number of 1's)
    function bit calc_parity(input logic [7:0] data);
        integer i;
        bit parity = 0;
        for (i = 0; i < 8; i = i + 1) begin
```

```
            parity = parity ^ data[i]; // XOR to count 1's
        end
        return parity; // 0 for even, 1 for odd parity
    endfunction

  initial begin
   $display("Parity of 5 (00000101): %0d", calc_parity(8'b00000101));// Odd
   $display("Parity of 8 (00001000): %0d", calc_parity(8'b00001000));// Even
  end
endmodule
```

SystemVerilog Tasks:

SystemVerilog tasks are used to perform a sequence of operations, potentially over multiple simulation time steps. Unlike functions, tasks can contain delays (#), procedural control (`fork-join`), and can execute over time. Tasks can take inputs and give back outputs. Unlike functions, tasks can not return values.

Key Points:

- Tasks can have delay statements.
- Tasks can not return values.
- Tasks can have inputs and outputs.
- Tasks can be used for procedural stuff that need time delay statements.

Example:

Let's say we have a Design Under Test that has a serial input port. Means that it gets input data bit by bit. To send the data bit by bit to the DUT from the Testbench, we can use SystemVerilog Task as you can see below:

```
module example;

    logic clk;
    logic data_in;

     // for simplicity dut instantiation is not shown here
```

```
    // Task to drive a signal with a delay
    task drive_signal(input logic [7:0] data, input int delay_time);
        integer i;
        for (i = 0; i < 8; i = i + 1) begin
            data_in = data[i];     // Drive a bit of data to the DUT input
            #delay_time;           // Delay for the specified time
        end
    endtask

    // for simplicity clock generation logic is not shown here

    // Test the task
    initial begin
       // Drive data with a 5-time-unit delay between bits
        drive_signal(8'b11001100, 5);
    end
endmodule
```

1.5 SystemVerilog Testbench Structure

SystemVerilog basic testbenches follow this structure:

```
module register_test;

// local signals of the testbench
logic [7:0]   out  ;
logic [7:0]   data ;
logic         enable  ;
logic         rst_ = 1'b1;
logic         clk = 1'b1;


// DUT instantiation
 register r1 (.enable(enable), .clk(clk), .out(out), .data(data), .rst_(rst_));

// clock generation
always #5 clk = ~clk;


// generate stimulus
  initial
    begin
      @(negedge clk)
      { rst_, enable, data } = 10'b1_X_XXXXXXXX; @(negedge clk) expect_test ( 8'hXX );
      { rst_, enable, data } = 10'b0_X_XXXXXXXX; @(negedge clk) expect_test ( 8'h00 );
      { rst_, enable, data } = 10'b1_0_XXXXXXXX; @(negedge clk) expect_test ( 8'h00 );
      { rst_, enable, data } = 10'b1_1_10101010; @(negedge clk) expect_test ( 8'hAA );
```

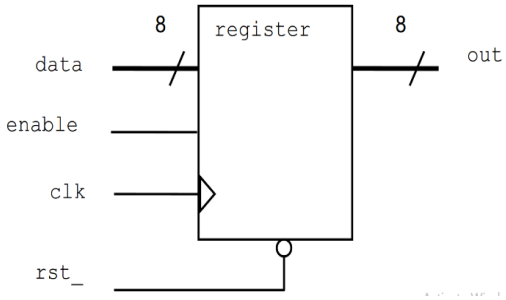
```
    { rst_, enable, data } = 10'b1_0_01010101; @(negedge clk) expect_test ( 8'hAA );
    { rst_, enable, data } = 10'b0_X_XXXXXXXX; @(negedge clk) expect_test ( 8'h00 );
    { rst_, enable, data } = 10'b1_0_XXXXXXXX; @(negedge clk) expect_test ( 8'h00 );
    { rst_, enable, data } = 10'b1_1_01010101; @(negedge clk) expect_test ( 8'h55 );
    { rst_, enable, data } = 10'b1_0_10101010; @(negedge clk) expect_test ( 8'h55 );
    $display ( "REGISTER TEST PASSED" );
    $finish; // finish the simulation at the end
  end

// Separate initial block to Monitor Results and finish the simulation after certain
// time has passed, as sometimes the simulation may get stuck.
initial
  begin
   $timeformat ( -9, 1, " ns", 9 );
   $monitor ( "time=%t enable=%b rst_=%b data=%h out=%h",
              $time,   enable,   rst_,   data,   out );
   #(`PERIOD * 99)
   $display ( "REGISTER TEST TIMEOUT" );
   $finish;
  end


// SystemVerilog Task to Verify Results
task expect_test (input [7:0] expects) ;
// task can access all the signals of the testbench, e.g out is accessible directly
// no need to make out also as the input
  if ( out !== expects )
    begin
      $display ( "out=%b, should be %b", out, expects );
      $display ( "REGISTER TEST FAILED" );
      $finish;
    end
endtask

endmodule
```

- `logic` is used to define signals.
- `always #5 clk = ~clk;` to generate a clock signal.
- `initial` blocks execute once at the start of simulation.
- `$display;` to display the info or signals value on the console.
- `$monitor;` to continuously monitor a signal and display it whenever it changes.
- `$time;` to get the current simulation time.
- `$finish;` finishes the simulation.

2. Lab Tasks

Task 1: Modeling and Testing a simple Register

Describe a register using SystemVerilog and verify that register with supplied directed and randomized self-checking testbenches.
Read the specification first and then follow the instructions to complete this task.



Specifications:

- data and out are both 8-bit logic vectors.
- rst_ is asynchronous and active low.
- The register is clocked on the rising edge of clk.
- If enable is high, the input data is passed to out.
- Otherwise, out retains its current value.

Modeling Register

1. Implement the register using SystemVerilog or Verilog constructs.

Directed Testing of the Register

2. Use the provided testbench register_test.sv.
3. Simulate the register design and observe the expected results.
4. Capture a screenshot of the test output.

Randomized Testing of the Register

For this Part we have provided randomized testbenches for testing your register model by generating random inputs to create test cases that we might have missed in previous directed testbench. We have provided a self-checking testbench. So, the test will automatically check if the register adheres to the specification and display test passed or failed status automatically.

5. Use the provided randomized testbench register_random_test.sv to test the register design.
6. Testbench register_random_test uses **$urandom** to generate random values for the data input of the register.

7. Observe how testbench register_random_test.sv validates the register's functionality under randomized conditions.
8. Run the simulation. Make sure you see the **Register Test Passed** output in the simulation log.

Task 2: Testing an ALU Design

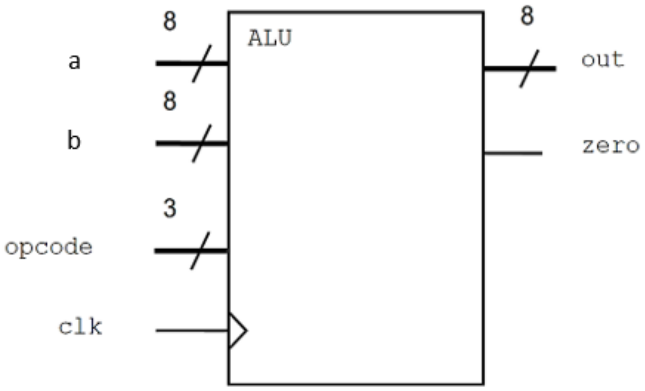
Create stimulus tasks using SystemVerilog subprogram enhancements and verify the supplied ALU design.

Read the specification first and then follow the instructions in the lab section, Completing the ALU Testbench.

ALU Specifications:

- Inputs a and b are 8-bit logic vectors.
- a, b and out are all 8-bit logic vectors. opcode is a 3-bit logic vector for the ALU operations.
- zero is a single bit, asynchronous output with the value of 1 when out equals 0 Otherwise, zero is 0.
- out is synchronized to the negative edge of clk and takes the following values depending on opcode.

Opcode	Encoding	Output
ADD	000	a + b
SUB	001	a – b
MUL	010	a * b
OR	011	a | b
AND	100	a & b
XOR	101	a ^ b
SLL	110	a << b
SRL	111	a >> b



Directed Testbench for ALU

Write a directed SystemVerilog testbench for the ALU design.

1. Study the ALU's functionality from alu.sv and typedef.sv.
2. Develop a directed testbench in file alu_directed_test.sv to verify the ALU features mentioned in its specification.

Randomized Testbench for ALU

Develop a randomized SystemVerilog testbench for the ALU design.

3. Create a randomized testbench alu_random_test.sv.
4. Validate the ALU's functionality under various conditions.

Good Luck 🙂