Lab 6 : Introduction to Verilog

Name:

Your Name

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Type Name

1 Objective

The main objective of this lab is to give hands on experience with Verilog HDL and the Synopsys VCS simulation environment.

2 Pre-requisite

For this lab you are expected to know some basic Verilog programming and understand VCS. You are advised to go through the Verilog tutorial posted on your course website.

3 Using Synopsys VCS

VCS is a hardware development and simulation environment designed by Synopsys. In our lab we will be using the simulator only and will not use the synthesis tools. VCS has its own built-in graphical verification environment, DVE. We will use these tools on the Linux workstations in our labs.

For those of you unfamiliar with Linux (or Unix) and the command line interface (CLI), you should review the basics on the following website:

http://www.ee.surrey.ac.uk/Teaching/Unix/index.html

Before you begin, you must setup the correct environment variables to run VCS. Perform the following steps to setup a terminal with the correct environment to run VCS:

1. Open up a "Terminal" window (Right click on desktop and select "Open Terminal").
2. Determine what type of shell you are running (see the tutorial for details of what this means). Type the following:

   ```
   echo $SHELL
   ```

   (a) If the answer contains "bash", then type the following to setup VCS:

   ```
   source /softwares/setup/synopsys/setup.synopsys.bash
   ```

   (b) If the answer contains "tcsh", then type the following to setup VCS:

   ```
   source /softwares/setup/synopsys/setup.synopsys.tcsh
   ```

3. Test that it works by entering the following at the command prompt:

   ```
   vcs
   ```

   You should see the following response:

   ```
   VCS MX compilation command help:
      (1) Unified use model for all design topologies :
          vcs [libname.]<Top Module_Or_Entity_Or_Config> [compile opts]

      (2) Two step use model for pure verilog design only :
          vcs <source_files> [compile opts]

      where frequently used [compile opts] are,
         [-debug][-debug_all][-o <log_file_name>][+rad][-cm][-sdf][-P <pli tab>]

      For more information,
          About the use model (1), Please refer chapter [4] in VCS MX User Guide or
          About the use model (2), Please refer chapter [3] in VCS User Guide or
          Type vcs -help
   ```

   If you see "command not found" instead, see your TA.

4 Short Tutorial

1. Create a working directory for the lab (see Unix tutorial). Type the following on the terminal:

```
mkdir Lab06
cd Lab06
```

2. Create a Verilog module file for the basic gates tutorial. Verilog modules are written as plain text files in a text editor. You may use any text editor you like, "emacs" and "vim" are the most popular and powerful. "gedit" is usable and similar to notepad. You may open the editor from the command line by typing:

   ```
   gedit Gates.v
   ```

   Note: emacs and vim have powerful Verilog modes that can assist in writing your Verilog assignments. See course webpage for information about these editor modes.

3. Enter the following Verilog text into the editor, save the file and close the editor:

   ```
   module Gates(in, out);
   input [0:1] in;
   output [0:2] out;

   and and0(out[0], in[0], in[1]);
   or or0(out[1], in[0], in[1]);
   xor xor0(out[2], in[0], in[1]);

   endmodule
   ```

   This creates a module which has one input of 2 bits, and one output of 3 bits. The first bit of the output is set to the AND of the two input bits, the second to the OR, and the third to the Exclusive OR.

4. Create a testbench for your verilog module. Create the file "GatesTest.v" in the text editor of your choice:

   ```
   gedit GatesTest.v
   ```

5. Enter the following Verilog for the testbench, save the files and exit the editor.

   ```
   module GatesTest();
   reg [0:1] in;
   wire [0:2] out;

   Gates DUT (.in(in), .out(out));

   initial
   begin
     #0      in=0;
     #10     in=2'b01;
     #10     in=2'b10;
   ```

```
    #10         in=2'b11;
    #10         in=0;
    $finish;
  end
  endmodule
```

6. Compile the Verilog simulation executable:

   `vcs -gui GatesTest.v Gates.v`

   Note: VCS first must compile the Verilog code into an executable simulation binary. This binary is then executed to simulate the Verilog.

7. Start up the simulator in VCS interactive mode:

   `./simv`

   The VCS DVE (Design Verification Environment) window should open up as shown in Figure 1:

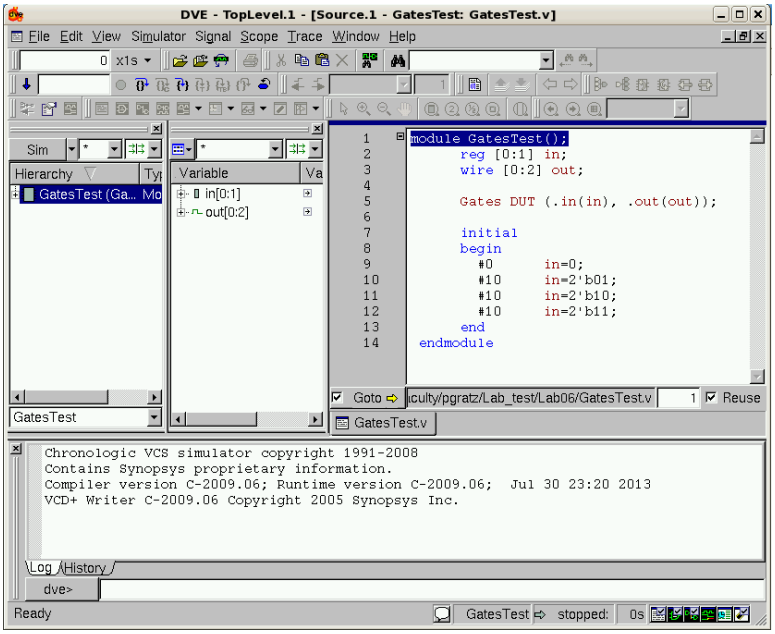

Fig. 1: VCS DVE Window

8. From the DVE window you must now open the waveform viewer.

(a) Expand the "GatesTest" instance in the "Hierarchy" pane of the DVE window (far right pane).

(b) Right click on DUT (Gates), select "Add to Waves", "New Wave View", as shown in Figure 2.

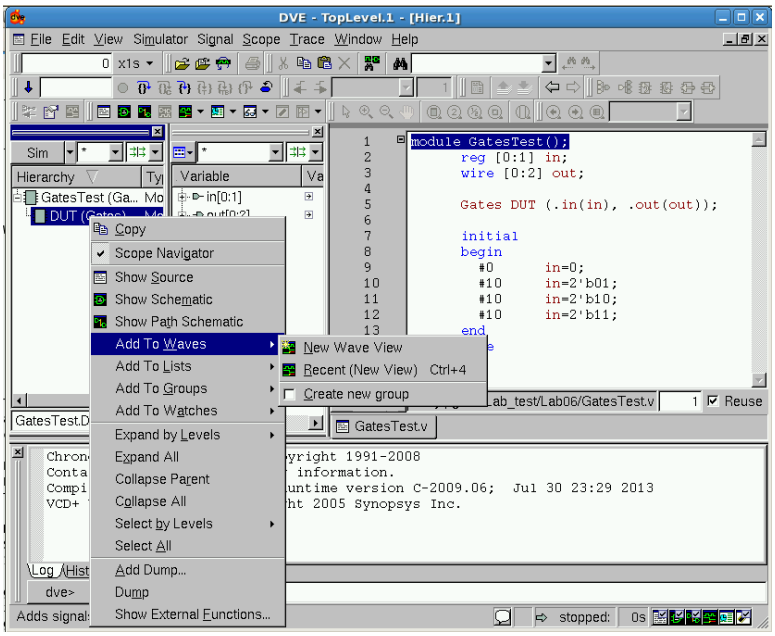

Fig. 2: VCS DVE Window, opening waveform viewer window.

(c) This should open the "Wave.1" window. Note that the waveforms should be blank since the simulator has not run yet.

(d) Go back to the "Heir.1" window and select "Simulator" then "Start/Continue" from the top menus to run the simulation.

(e) When simulation is complete the "Wave.1" window should now show the data from the Gates module during the test.

(f) Expand the output so you can see individual bits. The 0th bit should behave like an *and* gate, the 1st as an *or* gate, and the 2nd as an *xor* gate, as shown in Figure 3:

9. At this point, you must demo the simulation waveform to the TA.

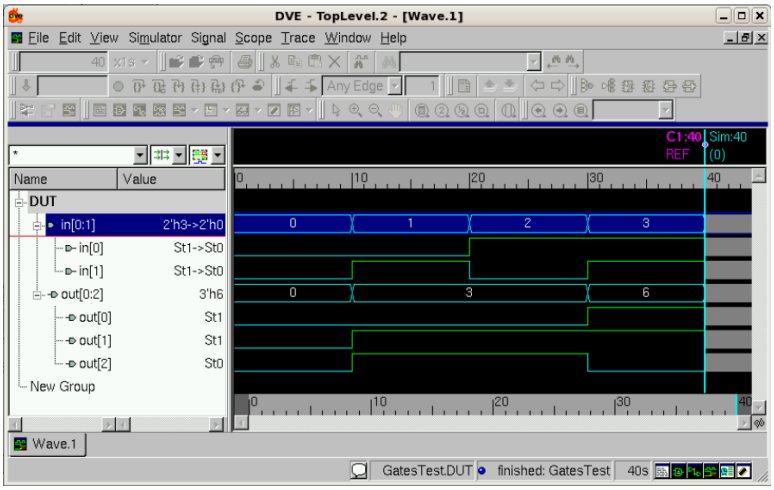

Fig. 3: Simulation

5 Testing Verilog Designs

For all modules listed below:

- Write down a Verilog implementation (if you not done so in the prelab).
- Test your code using the provided testbenches and verify that your code is correct.
- Demonstrate the correct functionality to the TA.

Note: Comment your code properly, in order to get the credit you deserve.

5.1 Half adder

Use structural model designed in the prelab with the following module definition:

```
module HalfAdd(Cout, Sum, A, B);
input A, B;
output Cout, Sum;
```

5.2 2-to-1 MUX designed (structural model)

Use the structural model designed in the prelab with the following module definition:

```
module Mux21(out, in, sel);
input [1:0] in;
input sel;
output out;
```

5.3 2-to-4 decoder

Write a behavioral model of a 2-to-4 decoder with the following module definition:

```
module Decode24(out, in);
input [1:0] in;
output [3:0] out;
```

Note: A 2-to-4 decoder produces a "One-Hot" output for each possible encoded input, e.g. an input of 00_2 produces an output of 0001_2, an input of 01_2 produces an output of 0010_2, and so on.

6 Deliverables

At the end of the lab you are supposed to turn in:

- Your Verilog HDL for each module.
- A screen capture (.png or .jpg) of the waveform taken from simulation of each module. (Your TA will demonstrate how to take a screen capture on Linux).