ECE 337: ASIC Design Lab 3

Week of 9/2/2013

SVN Usage: Initial Setup

- SVN is purely there for revision control of files
- Setup up SVN for a lab directory right after creating directory structure (mkdir LabX, dirset, setupX)
- How to setup up svn for a lab:
  - cd ~/ece337
  - svn import LabX $rdir/LabX/Trunk
  - rm –r LabX
  - svn co $rdir/LabX/Trunk LabX

SVN Usage: Adding New files

- Add source files right after creating with “ch” script
  - ch <source_filename>.sv
  - svn add   source/*.sv
  - svn ci
- Add script files right after creating
  - <make a .do file>
  - svn add scripts/*.do
  - svn ci

SVN Usage: Updating Modified Files

- Update SVN after filling a blank source file
- Update SVN after each big change
- Update SVN after getting a section of the design finished
- Update SVN after source file cleanly compiles
- Update SVN after clean LEDA run for source file
- Update (check in) with:
  - cd ~/ece337/LabX
  - svn ci

SVN Usage: Changing Versions

- If you want to revert back to your last check in
  - svn revert . : Reverts your current directory
  - svn revert <filename> : Reverts a specific file
- List the versions and notes
  - svn log : Change log for the current directory
  - svn log <filename> : Change log for specified file
- Checkout a previous version of you project
  - svn co $rdir/<project name>/Trunk@<Rev #> <Destination>

Structural Coding

```
module computeCksum
(
  input wire [15:0] i,
  output wire [1:0] ck_sum
);

  wire [7:0] midStep1;
  wire [3:0] midStep2;
  genvar k;

  xorbit U10(.a(i[0]), .b(i[1]), .o(midStep1[0]));
  ...
  xorbit U17(.a(i[14]), .b(i[15]), .o(midStep1[7]));
```

Structural Coding Continued

```
  generate
    for(k = 0; k <= 3; k = k + 1)
    begin
      xorbit U2X(.a(midStep1[k*2]),
        .b(midStep1[k*2+1]), .o(midStep2[k]));
    end
  endgenerate

  xorbit U30(.a(midStep2[0]), .b(midStep2[1]),
         .o(ck_sum[0]));
  xorbit U31(.a(midStep2[2]), .b(midStep2[3]),
         .o(ck_sum[1]));
...
endmodule
```

Test Bench

```
`timescale 1ns / 1ns

module tb_sensor_b
();

  wire [3:0] test_input;
  wire test_output;

  localparam NUM_TEST_CASES = 16;

  reg [3:0] tmp_input;
  reg [4:0] i;
  reg expected_output;
```

Test Bench Continued

```
sensor_b dut(.sensors(test_input), .error(test_output));

assign test_input = tmp_input;

initial
  begin
    for(i = 0; i < NUM_TEST_CASES; i = i + 1)
    begin
      // Send test input to design
      tmp_input = i[3:0];
      // wait to allow design to process input
      #10ns
```

Test Bench Continued

```
// Calculate the expected result
expected_output = 1'b0;
if(tmp_input[0] == 1'b1)
begin
  expected_output = 1’b1;
end
else if (tmp_input[1])
begin
  if((tmp_input[2] | tmp_input[3]) == 1’b1)
  begin
    expected_output = 1’b1;
  end
end
```

Test Bench Continued

```
      // Check expected with actual output from design
      if(test_output == expected_output)
        $display(“Test case passed!”));
      else // Test case failed
        $display(“Test case failed!”));
      end
  end
endmodule
```

Timescale

- `timescale <reference_time_unit>/<time_precision>
  - Allows specification of default time units for simulation
  - Allows specification of the level of precision for specified values
- Modelsim requires it to either being all files simulated or none of them
  - Cell libraries and Mapped versions have it
  - Thus, you must have it in your test bench to simulate mapped designs
- Standard usage for this class: `timescale 1ns / 100ps
  - Default units in nanoseconds
  - Time calculation resolution of 100ps

Assertions

- Great for simplifying testing
  - Use them in test benches to aid in identifying problems
  - Use them inside designs to aid in identifying the true source of a problem
- The following syntax must be used in a procedural block
- Immediate Assertion General Syntax:

```
assert(<condition>)
        [code for if true]
else
        [code for if false]
        <$error/$fatal/$warning/$info>(<error message>);
```

Internal Assertions

- Great for simplifying debugging of Source Code
- Use for checking for proper inputs to designs
- Use for checking for functional correctness of components
- Design Compiler Ignores them
  - Not synthesizable
  - Not carried over to mapped version

Code Coverage

- How do you check if you test bench is thorough?
  - Coverage Testing
- Types of Coverage Tests
  - Statements
  - Expressions
  - Toggles
  - States (FSM)
- Will be doing Statement, Expression, & Toggle (0/1) coverage in this Lab

Using GNU Make

- If you encounter errors stating it cannot open a folder in a work library:
  - Run “make clean <earlier target>”
- In Lab3 your 16-bit adder is a hierarchical design
  - populate the TOP_LEVEL_FILE and COMPONENT_FILES variables in the makefile
  - use “make sim_full_source” or “make sim_full_mapped” to simulate it