ECE 337 Lab 3:
Introduction to Hierarchical Designs and Verification, Code Coverage, and Flexible/Scalable Designs

In this lab, you will:

- Create, debug and verify functionality of a 16-Bit Ripple Carry Adder built from your 1-bit Full Adder design from Lab 2
- Use parameters to create flexible and scalable versions of your Serial-to-Parallel and Parallel-to-Serial Shift Register designs
- Use code coverage analysis to evaluate and improve your 16-bit Ripple Carry Adder test bench
- Design and Verify a flexible counter design

1. Setup for the Lab

The first thing you are going to need to do is create your Lab 3 directory structure as you did for the previous lab. To do so, issue the following commands at your UNIX terminal prompt:

```
mkdir ~/ece337/Lab3
cd ~/ece337/Lab3
dirset
```

Now that the Lab 3 directory has been created and its structure has been setup, run the setup command for this lab:

```
setup3
```

This command is actually an alias to a script file that that will check your Lab 3 directory structure and give you file needed for starting the lab. **If you have trouble with this step please ask for assistance from your TA.**

Make sure to import this new workspace into your 337 Repository, like you did in Lab1. This way, you will always have the original copy in storage. Remember to delete and then check out the Lab3 folder after you import it, so you have a working copy.

2. 16-Bit Adder Circuit

2.1. Connecting 1-Bit Adder components to make a 16-Bit Adder

The next step is to create the 16-Bit Ripple Carry Adder. Figure 1 illustrates how a 3-Bit ripple carry adder can be constructed from three 1-Bit full adders.

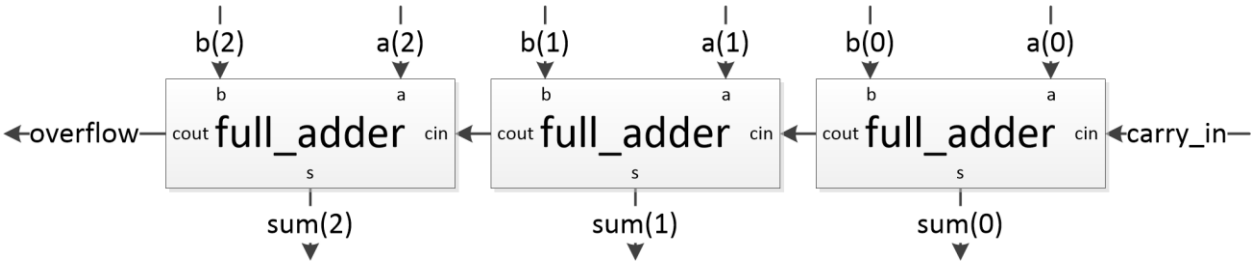

Figure 1: 3-bit Ripple Carry Adder Diagram

The filename you will use for the 16-Bit adder is **adder16bit.sv** and the module is:

```
module adder16bit
(
  input  wire [15:0] a,
  input  wire [15:0] b,
  input  wire carry_in,
  output wire [15:0] sum,
  output wire overflow
);
```

We would like you to use the structural style of coding to create the 16-Bit adder from your 1-bit Full Adder design you made and tested in Lab 2. *Note: Your full_adder.sv file has already been copied from you 'Lab2/source' folder into your 'Lab3/source' folder by the setup3 script*.

At simulation time, ModelSim will look for matching module declarations in the work library used and any additional libraries specified with the vsim command. The makefile will take care of specified the additional libraries for mapped version gates, gold/reference models, and provided course IP modules. Although it would still be a good idea to investigate how the makefile does that for you.

Declare any intermediate signal(s) you want to use. For this lab, an intermediate signal you will use is the one that connects the carrys between adders and/or output pins. Declare this signal and give it an appropriate name (we will use the name carrys, 16 bits in size, to illustrate our point). After you declare the intermediate signal(s) you will need, you can start connecting the components.

The following code is an example of how you would port map the first three 1-bit adder instances:

```
full_adder I00 (.a(a[0], .b(b[0], .c_in(carry_in), .s(sum[0]),
  .c_out(carrys[0]));
full_adder I01 (.a(a[1], .b(b[2], .c_in(carrys[0]), .s(sum[1]),
  .c_out(carrys[1]));
full_adder I02 (.a(a[2], .b(b[2], .c_in(cars[1]), .s(sum[2]),
  .c_out(carrys[2]));
```

Now based on the limited Verilog syntax that shown so far, one may think that it is necessary to directly type out (or copy, paste, and modify) each 1-bit adder instance. However, there is a powerful syntax to simplify repetitive structural/dataflow tasks such as this. It's known as a generate loop and results in far more efficiently written code, and fewer port mapping typographical errors. Below is an example of how one can use the generate syntax to exploit the iterative pattern for the 1-bit adder port maps to save a lot of time and frustration.

```
wire [16:0] carrys;
genvar i;

assign carrys[0] = carry_in;
generate
  for(i = 0; i <= 15; i = i + 1)
  begin
    full_adder IX (.a(a[i]), .b(b[i]), .c_in(carrys[i]),
      .s(sum[i]), .c_out(carrys[i+1]));
  end
endgenerate
assign overflow = carrys[16];
```

After you have finished creating your generate based 16-bit adder version, create and run a simple test bench that simulates it. Once you have done this, **show your TA to get checked off.**

2.2. Testing/Verification with Internal Assertion Statements

Another useful feature for testing is the use of Assertion Statements inside designs, in addition to their use externally in test benches. Assertions can be used to check for functional correctness internally during larger scale testing to simplify debugging. Two main uses of assertions internal to a design are to check for valid inputs and to check for correction functionality of components. This allows you to have messages sent to the terminal to more accurately describe the situation when design behavior doesn't match expected behavior during testing, instead of having to fully figure out what's not working and where it starts from just looking at the waveforms.

Please note that Design Compiler ignores assertions since they are not synthesizable and thus internal assertions will not be carried over into the synthesized/mapped file.

The syntax for the assert command is as follows:

```
assert(<DUT's output> = <expected output>)
  $display("<Correct behavior message>");
else
  $error("<Error message>");
```

Where:

- *<DUT's output> is the name of the signal that is being output from the DUT*
- *<expected output> is the name of the signal that is being output from a GOLD Model or a constant logic state, `0' or `1', indicating the expected value of the output.*
- *<Correct behavior message> is a user defined message used to indicate that the correct behavior was observed. An Example is: "DUT's output matches the expected output". This message will be displayed in the main Modelsim transcript window when you run the test bench simulation.*
- *<Error message> is a user defined error Message. An Example is: "DUT's output does not match the expected output". This error message will be displayed in the main Modelsim transcript window when you run the test bench simulation.*

Note on usage of assert: If you would like to only have a message displayed during an error then you simply don't include the $display() code and the semicolon after it.

An example of input value checking for a 1-bit adder:

```
always @ (a)
begin
  assert((a == 1'b1) || (a == 1'b0))
  else $error("Input 'a' of component is not a digital logic
    value");
end
```

An example of function component checking for the sum output of the first 1-bit adder:

```
Always @ (a[0], b[0], carrys[0])
begin
  assert(((a[0] + b[0] + carrys[0]) % 2) == sum[0])
  else $error("Output 's' of first 1 bit adder is not correct");
end
```

Now update your 1-bit adder to use internal assertions to check its inputs and update your generate based structural 16-bit adder design to use assertions to check that each 1-bit adder is functioning correctly. Once you have done this, **show your TA to get checked off.**

2.3. 16-Bit Adder Simple/Functional Test Bench Design

Since the 16-Bit adder essentially has 33 bits on input, it is not reasonable to create an exhaustive for-loop to test the design. It would take too long. So what should be the approach to verifying the functionality of this circuit? The key is to determine unique cases which cover specific subsets of the functionality and have relatively little overlap. This way maximum functionality can be tested using as few test cases as needed.

To help you with this, below are a number of requirements that your test bench must satisfy.

- Your first test should be A = 0x0000, B = 0x0000
- Your second test should have A be a large number and B be a small number
- Your third test should have B be a large number and A be a small number
- Your fourth test should have A be a large number and B be a large number
- Your fifth test should have A be a small number and B be a small number
- Along with visual inspection, you need to calculate the expected result using a different method that what is used inside the module. For example, you can simply directly calculate the expected sum and overflow values for the 16-bit adder in a similar way as you did within the 1-bit adder functional correctness assertions earlier.
- You must have a signal called no_match which is asserted high when the DUT does not match the expected output values.
- You must have an assertion after each of your test cases. The assertion should be that the DUT's outputs equal the expected outputs.

In order to properly simulate your hierarchical 16-bit Ripple Carry Adder design (as well as provide the grading system the information it needs) you must populate the following variables in your makefile.

- The "TOP_LEVEL_FILE" variable in your makefile must contain the filename of your 16-bit adder design (not including the source folder)
- The "COMPONENT_FILES" variable in your makefile must contain the filename of your 1-bit adder design (not including the source folder)

Once you have simulated your design under these unique functional test cases and feel confident in the correctness of your design (even after synthesis), **submit your 16-bit adder design to be autonomously graded via the 'submit Lab3A' command.**

3. Creating Flexible and Scalable Designs with Parameters

3.1. Verilog Parameters

Verilog parameters are effectively constants that are specific to a module instead of being a global constant like "`define" constants. They are rather similar to the 'const' in C. One difference between Verilog parameters and C 'const' is that there are two classes of parameters in Verilog. The first class is 'localparam' which is pretty much the same as a C 'const', as it is a constant local to namespace (the module) in which it's declared/defined and can't be modified. The second class is 'parameter' which is a constant that is local to the namespace (the module) in which it's declared/defined, but its value can be modified on a per instance level during the via the instance's port map. This allows us to be able to design the code of a module around a 'parameter' and then simply choose the value of the parameter at the instance's port map or use the 'default' value if we don't want to set the parameter's value during the port map. Both types of parameters are declared/defined in the same way, as shown below.

```
parameter <name> = <default value>;
localparam <name> = <value>;
```

However, since parameters are intended to have their value modified and will often be used to scale internal data sizes and corresponding port sizes, they should be declared inside the module declaration as follows.

```
module <module name>
#(
  <parameter declaration>,
  …
  <parameter declaration>
)
(
  <port declaration>,
  …
  <port declaration>
);
```

3.2. Flexible and Scalable Serial-to-Parallel Shift Register Design

3.2.1. Parameterizing your 4-bit Serial-to-Parallel Shift Register Design

Using the above discussion of parameters, modify your Serial-to-Parallel Shift Register design to have the following behavior:

- Have a parameter called 'NUM_BITS' that determines the number of bits in both the internally stored value and the 'parallel_out' port, with a default value of 4.
- Have a parameter called 'SHIFT_MSB' that determines the shifting direction of the register such that a Boolean value of 'true' results in shifting most significant bit first and 'false' results in shifting least significant bit first.

3.2.2. Testing your flexible Serial-to-Parallel Shift Register Design

To assist you in testing your new flexible design, the setup3 script you ran earlier has provided you with a test bench called 'tb_flex_stp_sr.sv' which creates a couple different instances of your flexible shift register and simultaneously tests each one. In order to be able to test out the synthesized versions of your flexible design 'wrapper' files are needed and were also provided. These 'wrapper' files simply contain an instance of your flexible design with set values for the parameters. When the wrappers are synthesized the result will be a gate net list for the desired design, without needed to maintain separate code that is only different in size or shift direction. Additionally it overwrote the default makefile provided by dirset with one customized for this lab so that it would have the following special targets just for compiling and simulating this flexible design test bench.

```
make tbsim_flex_stp_sr_source
```

and

```
make tbsim_flex_stp_sr_mapped
```

In order for the above commands to properly simulate your design (as well as provide the grading system the information it needs) you must populate the following variables in your makefile.

- The "STP_SR_FILE" variable in your makefile must contain the filename of your 16-bit adder design (not including the source folder)

3.2.3. Automated Grading of the Flexible Serial-to-Parallel Shift Register

To submit your flexible serial-to-parallel shift register design for grading, issue the following command at the terminal (can be from anywhere).

```
submit Lab3FS
```

3.3. Flexible and Scalable Parallel-to-Serial Shift Register Design

3.3.1. Parameterizing your 4-bit Parallel-to-Serial Shift Register Design

Using the knowledge and experience gained from parameterizing your Serial-to-Parallel Shift Register design above, modify your Parallel-to-Serial Shift Register design to have the following behavior:

- Have a parameter called 'NUM_BITS' that determines the number of bits in both the internally stored value and the 'parallel_in' port, with a default value of 4.
- Have a parameter called 'SHIFT_MSB' that determines the shifting direction of the register such that a Boolean value of 'true' results in shifting most significant bit first and 'false' results in shifting least significant bit first.

3.3.2. Testing your flexible Parallel-to-Serial Shift Register Design

Just like for the serial-to-parallel shift register, the setup3 script has provided a test bench (tb_flex_pts_sr.sv) and wrapper files for testing your flexible parallel-to-serial shift register design. The test bench simulation make targets are as follows.

```
make tbsim_flex_pts_sr_source
```

and

```
make tbsim_flex_pts_sr_mapped
```

In order for the above commands to properly simulate your design (as well as provide the grading system the information it needs) you must populate the following variables in your makefile.

- The "PTS_SR_FILE" variable in your makefile must contain the filename of your 16-bit adder design (not including the source folder)

3.3.3. Automated Grading of the Flexible Parallel-to-Serial Shift Register

To submit your flexible parallel-to-serial shift register design for grading, issue the following command at the terminal (can be from anywhere).

```
submit Lab3FP
```

4. Verification through Coverage Analysis

Coverage is used to check whether the Test bench has satisfactory exercised the design or not. It will measure the efficiency of your verification implementation. There are different types of coverage.

4.1. Statement Coverage

This is the most basic type of coverage which checks how many statements in your design are covered by your test bench. Any moderately complex design will have a lot of conditional statements (Ex. If, when, with case select etc.), which will generally be difficult to fully cover with a small set of test cases.

4.2. Expression Coverage

Any Boolean expression can be expressed as a truth table. Expression coverage measures how many rows in the truth table of the expression have been exercised by your test bench.

Example:

```
a and b are 1 bits
y = a xor b;
```

In this example, there are four rows in the truth table of 'y' – 00,01,10,11. Expression coverage measures how many of those combinations are exercised by your test bench.

4.3. Other Coverage Metrics

There are other types of coverage like State Machine Coverage (How many states in the state machine are covered by tour test bench), toggle coverage (how many times each bit of register, wires and buses toggled during simulation), etc.

4.4. Coverage Reporting

Please follow the steps below to generate the code coverage numbers:

1. Clear your work area in Modelsim
   make clean
   grid vsim -i
2. Compile > compile options(tab)
   Make sure the options are selected as shown in Figure 2 to the right.
3. Close ModelSim and rerun the make simulation target.
   (i.e. make tbsim_%_source)
4. Run the simulation for as long as needed
5. Tools > Coverage Report > Text
   a. Select "All instances" from the top drop down box
   b. > Ok

   Now Modelsim will display the report.
6. View the statements/Expressions that are not covered by double-clicking the Design Name(DUT) in "sim" tab.
7. Take a statement/expression in your code that is not covered, and explain to TA why it is not covered.

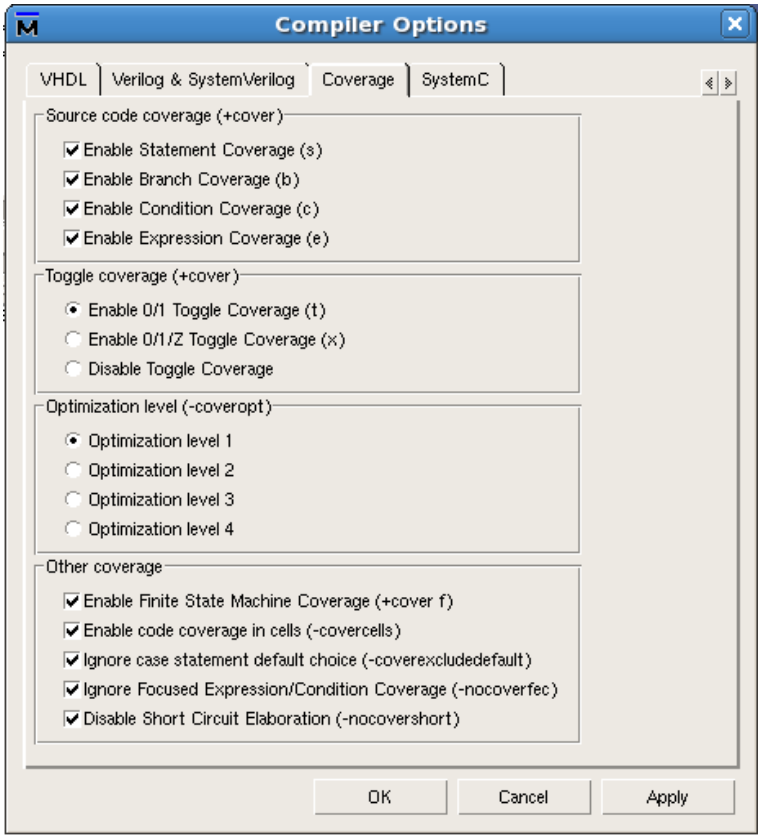

Figure 2: Coverage Compile Options

At this point, **show your TA to get checked off.** 

4.5. Coverage Based 16-bit Adder Test Bench Grading

Once your source and mapped versions of the 16 Bit adder work, your test bench meets all of the requirements listed in section 2.3, and the text file report of coverage shows 100% coverage of the design files during both a source simulation (full_adder.sv & adder16bit.sv) and a mapped simulation (adder16bit.v & gate instances, but not dffsr instances), **submit your design for grading using:**

submit Lab3CC

Note on test bench requirements:
Since both inputs A and B are 16-Bits long, the number of possible test case combinations is on the order of billions so it will be very suspicious if multiple students have similar, let alone exactly the same, test case input values.
Note: Assertions do not need to be fully covered as they are not actual design code and are there to make debugging easier.

5. Designing a Flexible and Scalable Counter with Controlled Rollover

Another common building block used in hardware systems is the counter with controlled rollover. Counters are used to keep track of events that have occurred and how much time has elapsed (in terms of clock cycles or other events that have occurred). In each of the later design labs you will be needing to use a form of a counter for one of these purposes. So to improve reusability of code and minimize wasted time you will design a flexible and scalable counter during this lab based on the knowledge and experience you have gained from the prior sections and tasks of this lab.

The function of a counter is to increment an internal count value every cycle that a desired event occurs until a specified 'rollover' value has been reached at which point it will set the internal value to 1 upon the next desired event occurrence. Additionally it activates the value of a rollover flag output port whenever a rollover has occurred and clears the value upon the next desired event occurrence.

5.1. Flexible Counter Specifications

The required module name is: **flex_counter**
The required filename is: **flex_counter.sv**

The module must be designed around the following parameter:

Parameter NUM_CNT_BITS <# of bits needed to hold the maximum internal count value>

The default value for the NUM_CNT_BITS parameter must be 4.

The module must have only the following ports (case-sensitive port names):

Signal	Direction	Description
clk	Input	The system clock. **(400 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is asserted (logic '0'), all registers/flip-flops in the device must reset to their initial values.
count_enable	Input	This is the active-high enable signal that allows the counter to increment its internal value.
rollover_val[#:0]	Input	This is the N-bit value that is checked against for determining when to rollover. The actual port declaration should use the NUM_CNT_BITS parameter value to determine the value of the '#'.
rollover_flag	Output	This is the active high rollover flag and must be asserted once a rollover has occurred and cleared with the next increment.

5.2. Flexible Counter Automated Grading

When you are ready to **submit your design's code and RTL diagram** for grading, use the following command:

```
submit Lab3FC
```

Make sure to test your design using different values for NUM_CNT_BITS.